TypeScript Succinctly

by Steve Fenton

Syncfusion | Technology Resource Portal

TypeScript Succinctly

By
Steve Fenton

Foreword by Daniel Jebaraj

Important licensing information. Please read.

Edited by

This publication was edited by Praveen Ramesh, director of development, Syncfusion, Inc.

Table of Contents

The Story behind the *Succinctly* Series of Books

Daniel Jebaraj, Vice President
Syncfusion, Inc.

Staying on the cutting edge

As many of you may know, Syncfusion is a provider of software components for the Microsoft platform. This puts us in the exciting but challenging position of always being on the cutting edge.

Whenever platforms or tools are shipping out of Microsoft, which seems to be about every other week these days, we have to educate ourselves, quickly.

Information is plentiful but harder to digest

In reality, this translates into a lot of book orders, blog searches, and Twitter scans.

While more information is becoming available on the Internet and more and more books are being published, even on topics that are relatively new, one aspect that continues to inhibit us is the inability to find concise technology overview books.

We are usually faced with two options: read several 500+ page books or scour the web for relevant blog posts and other articles. Just as everyone else who has a job to do and customers to serve, we find this quite frustrating.

The *Succinctly* series

This frustration translated into a deep desire to produce a series of concise technical books that would be targeted at developers working on the Microsoft platform.

We firmly believe, given the background knowledge such developers have, that most topics can be translated into books that are between 50 and 100 pages.

This is exactly what we resolved to accomplish with the *Succinctly* series. Isn't everything wonderful born out of a deep desire to change things for the better?

The best authors, the best content

Each author was carefully chosen from a pool of talented experts who shared our vision. The book you now hold in your hands, and the others available in this series, are a result of the authors' tireless work. You will find original content that is guaranteed to get you up and running in about the time it takes to drink a few cups of coffee.

Free forever

Syncfusion will be working to produce books on several topics. The books will always be free. Any updates we publish will also be free.

Free? What is the catch?

There is no catch here. Syncfusion has a vested interest in this effort.

As a component vendor, our unique claim has always been that we offer deeper and broader frameworks than anyone else on the market. Developer education greatly helps us market and sell against competing vendors who promise to “enable AJAX support with one click,” or “turn the moon to cheese!”

Let us know what you think

If you have any topics of interest, thoughts, or feedback, please feel free to send them to us at succinctly-series@syncfusion.com.

We sincerely hope you enjoy reading this book and that it helps you better understand the topic of study. Thank you for reading.

Please follow us on Twitter and “Like” us on Facebook to help us spread the word about the *Succinctly* series!

About the Author

Steve Fenton is an Agile .NET developer with a keen interest in the web stack. He has been creating business applications since 1997 and writing web applications since 2001. He is currently keeping an eye on emerging standards and techniques for web development and was an early adopter of the HTML 5 standards, releasing his first HTML 5 website in 2009.

Steve is well versed in JavaScript and has dabbled with alternatives such as Flash, Silverlight, CoffeeScript, and Dart, but always preferred writing raw JavaScript—until he tried TypeScript in 2012.

In his spare time, Steve attends conferences, runs brown-bag learning lunches, reads books in his local coffee shop, and attends ice hockey matches.

TypeScript is a trademark of Microsoft Corporation.

Introduction

Whenever the word JavaScript is mentioned to a room full of .NET developers, there are visible shudders and uncomfortable fidgeting at the prospect of writing anything in such a sloppy language. I actually love JavaScript, partly because it was the first curly-braced language that I used, but also because I have learned to use it an appropriate way. However, I can understand the reasons for its lack of popularity amongst the .NET community. If you spend most of your time writing code in C#, VB.NET, or F#, the prospect of using a language that lacks sensible autocompletion, type checking, and object-orientation is not a pleasant one—and this is where TypeScript fits perfectly into the tool belt of a .NET programmer.

Is TypeScript the Answer?

There are no golden bullets in the world of software development. What is certain is that JavaScript is one of the most widely adopted languages on Earth, and it isn't likely to be disappearing any time soon, especially given its recent emergence as a high-volume web-server language under the Node.js moniker.

TypeScript eases the pain of JavaScript development by adding some of the features that .NET developers take for granted. It is already smoothly integrated into Visual Studio, which makes it easy to use without switching development tools.

I envisage a future where developers don't need to write boilerplate JavaScript, not because they are using a framework that includes everything they might need to use, but because they can compose a number of small and reusable modules that take care of specific areas, like AJAX, SVG, and Canvas.

Who is This Book For?

I have written this book primarily for professional .NET developers. TypeScript isn't exclusively for the .NET domain, as Microsoft has released the language under an open source license and there are plug-ins for Sublime Text, Vim, Emacs, and WebStorm, as well as Visual Studio, which has a fully featured extension. You don't have to be a JavaScript expert to read this book, but if you would like to learn more about it, you can download *JavaScript Succinctly* by Cody Lindley from the Syncfusion Technology Resource Portal:

http://www.syncfusion.com/resources/techportal/ebooks

What is Missing?

Currently, TypeScript is available as a preview and the language specification is labeled version 0.9. It is likely that the list of features will grow as the TypeScript compiler becomes smarter. Some of the current features haven't been finalized and are likely to change, and there are further planned features that haven't yet been written. I have tried to make it clear where a feature is experimental or likely to change, as well as noting alternatives to features that can harm the readability of your code.

Code Samples

All of the examples in this book were created in Visual Studio 2012 using the freely available *TypeScript for Visual Studio 2012* extension. Many of the examples can also be tested in the online TypeScript Playground at http://www.typescriptlang.org/Playground/.

In this book, the code samples are shown in code blocks such as the following example.

Sample1.ts Code Samples

```
function log(message) {
    if (typeof window.console !== 'undefined') {
        window.console.log(message);
    }
}
```

The code samples in this book can be downloaded from https://bitbucket.org/syncfusion/typescript-succinctly.

Notes

There are notes that highlight particularly interesting information about a language feature, including potential pitfalls you will want to avoid.

Note: An interesting note about TypeScript.

Chapter 1 Concepts in TypeScript

Code organization

I will begin by drawing some parallels between .NET and TypeScript in respect to code organization and language features. In some cases the conventions are nearly identical, but there are also some interesting differences in both the naming conventions and the meaning of the various components used to compose your application.

Figure 1 demonstrates the structure of a TypeScript program, which is organized into a number of modules, each containing interfaces, classes, and variables. Each module should group related classes together and should have minimal dependencies on other modules in your program. Although you can create multidirectional dependencies and circular dependencies in TypeScript, I recommend that you avoid doing so, as it will make your program harder to maintain.

This program structure allows you to put your object-oriented programming skills into practice and makes it easy to use common design patterns to organize your program.

I will use the TypeScript naming conventions throughout this book, but feel free to read *method* when I use the term *function*, for example, if that is what you are more used to.

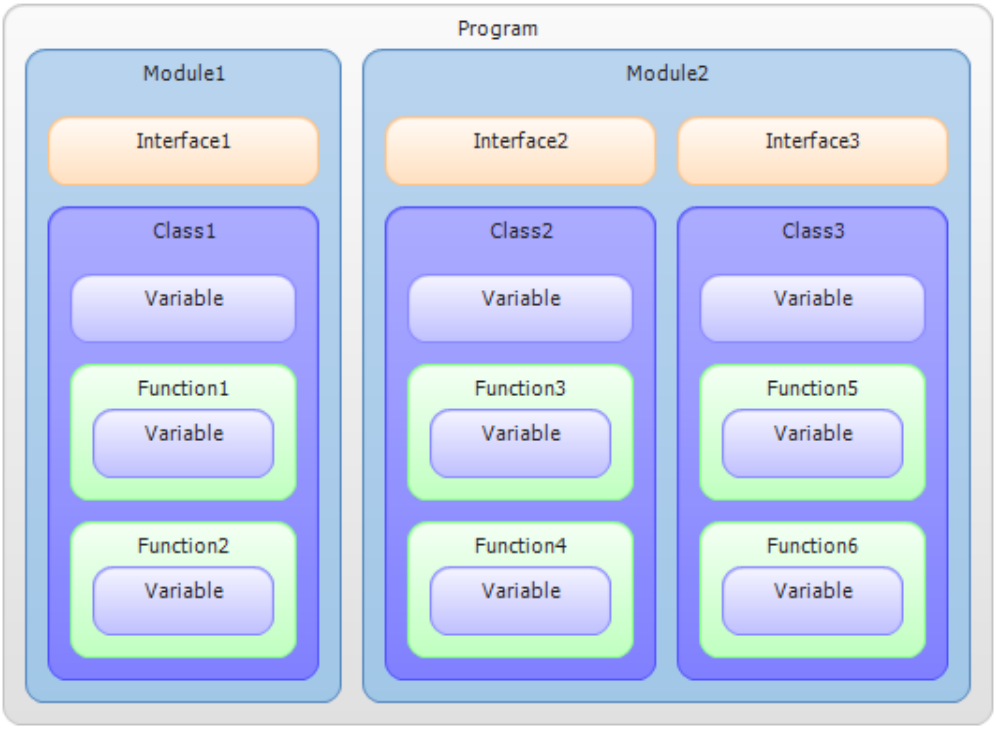

Figure 1: A TypeScript program

TypeScript Glossary

Program

A TypeScript program is a collection of source files. Most of these source files will contain your code implementation, but they can also contain declarations that add static types to external code. I will talk about writing you own code in Chapter 4, "Creating new modules," and consuming existing code and third-party libraries in Chapter 6, "Working with existing JavaScript."

You can think of your program as both the TypeScript .ts files and the compiled JavaScript .js files.

Module

TypeScript modules are similar to .NET namespaces, and can contain a number of classes and interfaces. The structure of your modules can vary from very flat structures to highly nested ones depending on the kind of program you are writing.

Interface

Interfaces in TypeScript work exactly like .NET interfaces, allowing the development tools and the compiler to emit errors if a class fails to implement an interface correctly. Because the interface exists solely for the benefit of the design-time tools and the compiler, no JavaScript code is generated from a TypeScript interface.

A TypeScript interface can extend multiple interfaces.

Class

A TypeScript class is similar to a .NET class and can contain functions and variables with varying levels of visibility. In TypeScript you can use `public` and `private` visibility keywords on functions and variables. A TypeScript class can extend one class and implement multiple interfaces.

Function

A TypeScript function is like a .NET method, accepting arguments and returning a typed value or void. This is where the real logic will live in your program.

Variable

In TypeScript all properties and fields are variables, but with optional typing. By default, these variables behave in a similar manner to those declared in .NET using the `var` keyword, as the compiler will attempt to infer the type and then treat the variable as statically typed. In cases where the type cannot be inferred, the compiler doesn't show an error as it would in .NET, but instead treats the variable as a dynamic type, which in TypeScript is indicated with the `any` keyword. I will discuss this in more detail in Chapter 3, "Type safety."

Enumeration

Enumerations in TypeScript are similar to .NET enumerations. The experimental implementation in version 0.8 of TypeScript has been amended slightly in version 0.9.

Scope

Because functions, classes, and modules are compiled into various flavors of JavaScript functions, everything nested within them is locally scoped, and will enjoy the benefits of JavaScript's lexical scoping. This means a nested function has access to its own variables and also the variables in the outer function. If no variable exists in the local scope, the JavaScript runtime walks up the chain to find the variable in the outer function, and if no variable exists there, it continues to the next level. This continues right up until JavaScript checks the global scope.

Compilation

TypeScript is converted into JavaScript at compile time. JavaScript is an implementation of the ECMAScript standard, and it isn't the only one–ActionScript is another well-known language based on the ECMAScript standard. There are three versions of ECMAScript that you will come across when dealing with TypeScript.

ECMAScript 3

ECMAScript 3 was published in 1999, and it is the version supported in most browsers. When TypeScript compiles into JavaScript, it uses the ECMAScript 3 standard by default. A very small number of features are unavailable in TypeScript when targeting ECMAScript 3, such as property getters and setters, but it offers the widest possible support for your program.

If you try to use a feature that isn't available in the version of ECMAScript you are targeting, the compiler will warn you to either avoid the feature or change the target version. I have included instructions on how to target ECMAScript 5 in Chapter 2, "Visual Studio."

ECMAScript 5

ECMAScript 5 was published in 2009 and is supported in all modern browsers. You can instruct the TypeScript compiler to target ECMAScript 5, which makes additional language features available, but your program may encounter problems in older browsers. The following browsers support all of the important features of ECMAScript 5; older versions of these browsers have only partial support:

- Internet Explorer 9 and above.
- Firefox 4 and above.
- Opera 12 and above.
- Safari 5.1 and above.
- Chrome 7 and above.

ECMAScript 6

ECMAScript 6, also known as ECMAScript Harmony, is currently being drafted and has experimental support in some of the latest browsers. This version is interesting, because it will make modules and classes part of the JavaScript language. TypeScript gives you immediate access to these proposed new features in practically all browsers. At some point in the future you will be able to target the ECMAScript 6 standard from TypeScript, which will mean the compiled JavaScript will more closely match your TypeScript code.

There is currently no release date planned for ECMAScript 6, and it doesn't support all of the concepts in the TypeScript language.

TypeScript Life Cycle

The TypeScript life cycle consists of three distinct stages. When you are writing TypeScript code, you are participating in the design time stage along with any development tools you are using. The compiler represents the second stage, converting TypeScript into JavaScript, and raising errors and warnings it discovers. The final stage involves the runtime executing the generated JavaScript.

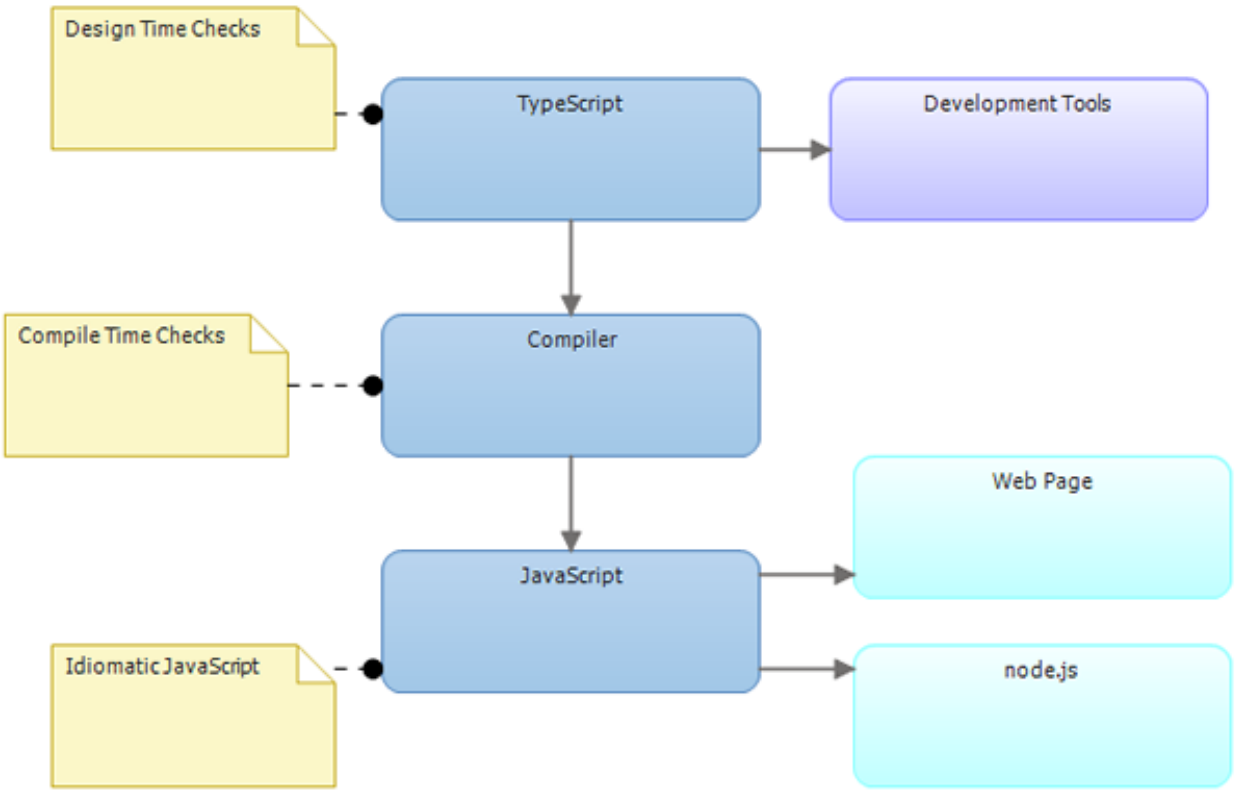

Figure 2: TypeScript life cycle

Chapter 2 Visual Studio

This chapter describes how to set up Visual Studio 2012 for TypeScript. If you are using a different development environment, you can skip this chapter. There are more details on using TypeScript with other tools in *Appendix A: Alternative Development Tools*.

Visual Studio Extension

The first task is to download and install the *TypeScript for Visual Studio 2012* plug-in from the TypeScript language website at http://www.typescriptlang.org/.

This will add TypeScript language support as well as new project and file templates. You can add TypeScript files to an existing project, but for the examples in this book you can simply create a new project using the *HTML Application with TypeScript* template, which can be found under the Visual C# Templates—this is something of a misnomer as the project won't contain C# or VB.NET.

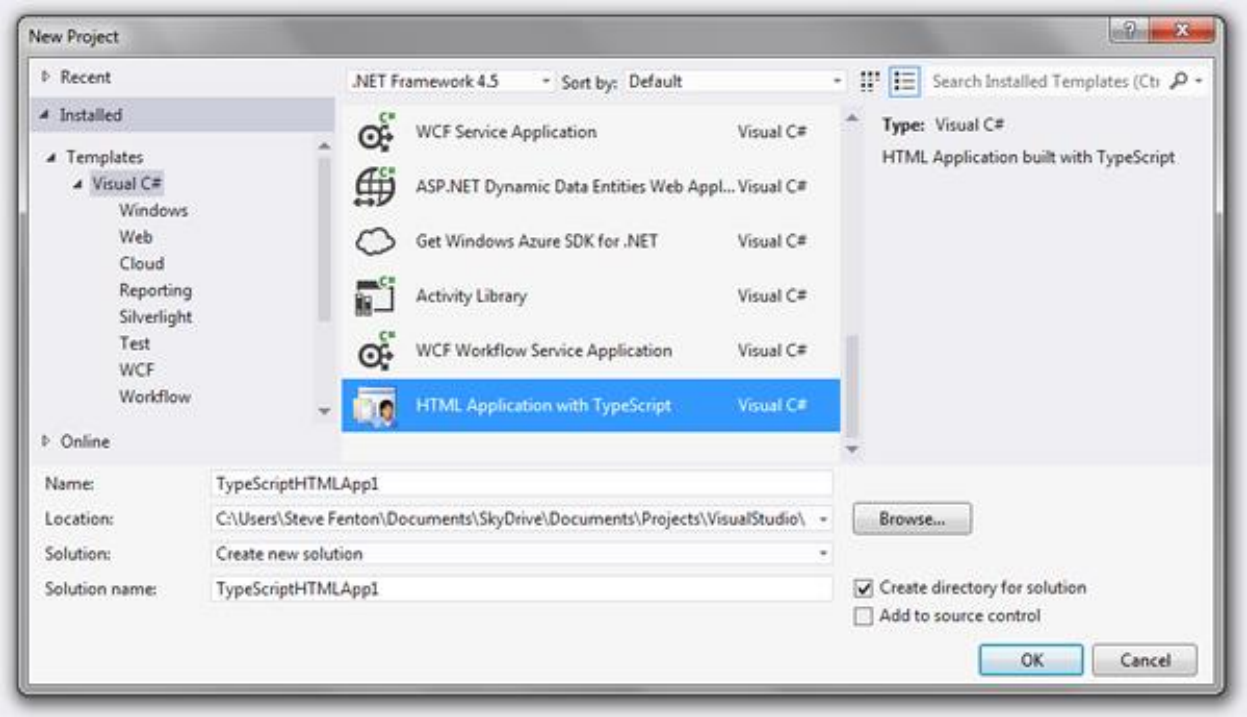

Figure 3: New project types in Visual Studio

The TypeScript file template can be accessed via the Add New Item menu, and also appears as a shortcut in the project context menu once you have added a TypeScript file to your project. The default template contains an example interface, module, and class, but you can change this template by navigating to the extensions zip file and editing the file named *file.ts*. The part of the path shown in bold is likely to be different on your computer, but you can perform a search for *f.zip* in the Extensions folder if you get stuck.

C:\Program Files (x86)\Microsoft Visual Studio 11.0\Common7\IDE\Extensions\
msfz1qy5.oca\~IC\IT\CSharp\1033\f.zip

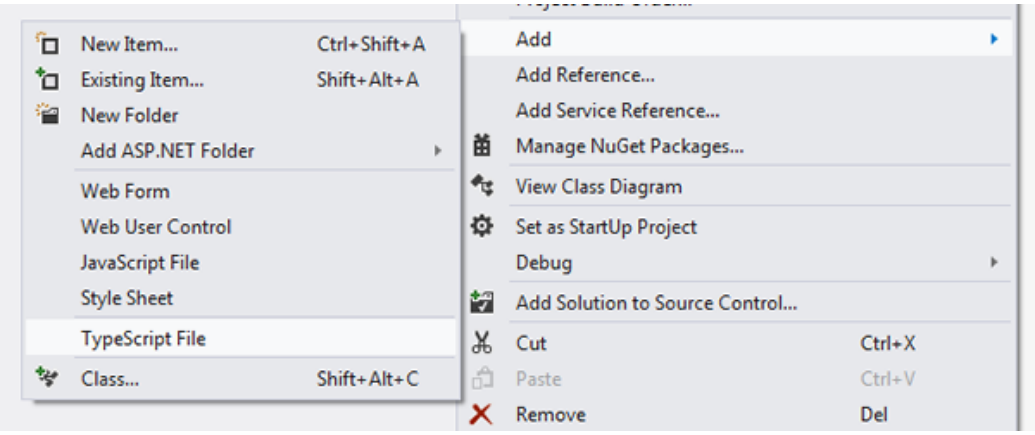

Figure 4: Add TypeScript file shortcut

The plug-in also installs the TypeScript compiler, which can be used from the command line, Visual Studio, and your build server.

C:\Program Files (x86)\Microsoft SDKs\TypeScript\tsc.exe

You can also use the Web Essentials 2012 extension, which adds side-by-side editing to Visual Studio.

Pre-Build Event

If you are using a project with automatic TypeScript support, such as the *HTML Application with TypeScript* project template, all of your TypeScript files will automatically be compiled into a paired JavaScript file each time you build the project, or each time a file is saved if you are using the latest version of the TypeScript Visual Studio extension.

If you want to add TypeScript to an existing project, for example, to an *ASP.NET MVC Web Application*, you can add the required sections to your project file. To edit your project file, select **Unload** from the context menu. Then re-open the context menu and select **Edit**.

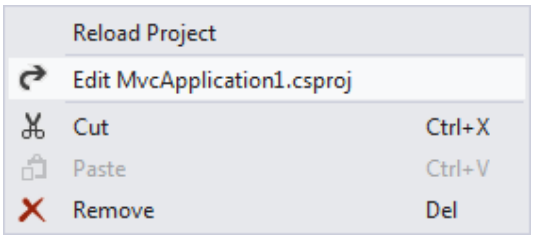

Figure 5: Project context menu

There are two sections to add. The `ItemGroup` element defines the files to be compiled (everything in your project with a .ts file extension). The `Target` element contains the build step that runs the TypeScript compiler against these files. The exact entries are shown below and can be pasted directly into your project.

```
  <ItemGroup>
    <TypeScriptCompile Include="$(ProjectDir)\**\*.ts" />
  </ItemGroup>
  <Target Name="BeforeBuild">
    <Exec Command="&quot;$(PROGRAMFILES)\Microsoft SDKs\TypeScript\tsc&quot;
@(TypeScriptCompile ->'&quot;%(fullpath)&quot;', ' ')" />
  </Target>
```

If you want the compiler to target ECMAScript 5, you can adjust this example to include the `--target` flag, as shown in the following example. You should only use this if you are certain that you don't need to support older browsers.

```
  <ItemGroup>
    <TypeScriptCompile Include="$(ProjectDir)\**\*.ts" />
  </ItemGroup>
  <Target Name="BeforeBuild">
    <Exec Command="&quot;$(PROGRAMFILES)\Microsoft SDKs\TypeScript\tsc&quot;
--target ES5 @(TypeScriptCompile ->'&quot;%(fullpath)&quot;', ' ')" />
  </Target>
```

Trying it Out

Once you have performed these steps, Visual Studio is ready to use. Build your solution to make sure that there are no problems with any changes you have made to the project file.

Chapter 3 Type Safety

Static, Dynamic, and Optional Types

A statically typed language gives you compile-time checking for type safety. This doesn't mean you have to specify all types explicitly, as a smart compiler can infer the types in many cases. TypeScript is no exception and the compiler will intelligently determine types for you even if you don't explicitly declare the types in your code.

TypeScript is referred to as optionally statically typed, which means you can ask the compiler to ignore the type of a variable if you want to take advantage of dynamic typing in a particular circumstance. This mix of static and dynamic typing is already present in .NET; for example, C# is statically typed but allows dynamic types to be declared with the `dynamic` keyword.

As well as compile-type checking, the language constructs allow static analysis that makes it possible for development tools to highlight errors in your code at design time without the need for compilation.

Throughout this chapter, I will give practical examples that will show you how to take advantage of TypeScript's static analysis and compilation.

Inferred Types

Before I start introducing the TypeScript language features, here is a plain and simple JavaScript logging function. This is a common paradigm in JavaScript development that checks for the presence of a console before attempting to write a message to it. If you don't perform this check, calling the console results in an error when one isn't attached.

Sample2.ts Logging Function

```
function log(message) {
    if (typeof window.console !== 'undefined') {
        window.console.log(message);
    }
}

var testLog = "Hello world";

log(testLog);
```

Even though this example is plain JavaScript, the TypeScript compiler is smart enough to infer that `testLog` is a string. If you hover over the variables and functions in your development environment you'll see the following tooltips.

Table 1: Types in Sample2.ts

Name	Tooltip
`log`	`(message: any) => void`
`message`	`any`
`window.console`	`Console`
`testLog`	`string`

There are several benefits to this type knowledge when it comes to writing your code. Firstly, if you try to assign a value to `testLog` that isn't a `string` type, you will get a design-time warning and a compilation error.

Because TypeScript is aware of the types, development tools are able to supply more precise autocompletion than they can for JavaScript. For example, if you didn't know the type of `testLog`, you would have to supply an autocompletion list that either covered all possible suggestions for all types or no suggestions at all. Because TypeScript knows the type is `string`, the autocompletion can contain just the properties and operations relevant to a string.

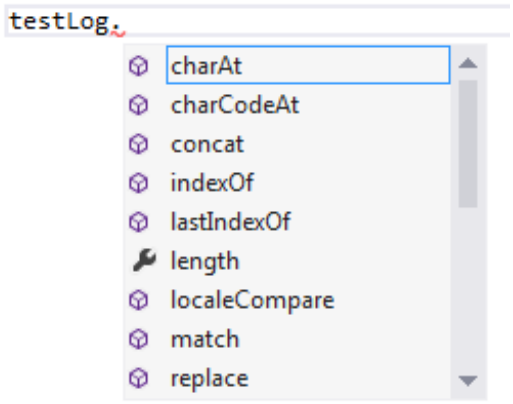

Figure 6: Precise autocompletion

This is a demonstration of type inference, where the tools work out the types based on the values you assign in code. It isn't always possible or desirable to infer the types automatically like this, and that is the reason the `message` parameter has the dynamic `any` type, which means it is not statically typed. You could argue that based on the calling code, which always passes a `string`, it is possible to infer that `message` is also of type `string`. The problem with this kind of inference is that it relies on all calling code being available at compilation time, which is almost certainly not the case–so rather than make a potentially dangerous assumption, TypeScript uses the `any` type.

You don't have to rely on type inference in your code, and you can use type declarations to tell the compiler the intended type of a variable, function, or parameter in cases where it cannot work it out. In the logging example, you only need to tell TypeScript the type of the message parameter in order for everything to be statically typed. To declare a type, append a colon followed by the type name; in this case, you would decorate the `message` parameter with a `: string` declaration

Sample3.ts Typed Parameters

```
function log(message: string) {
    if (typeof window.console !== 'undefined') {
        window.console.log(message);
    }
}

var testLog = "Hello world";

log(testLog);
```

Now when you view the tooltips for this code, you will see that everything is statically typed and the dynamic `any` keyword has been replaced in both the `log` function signature and the `message` parameter:

Table 2: Types in Sample3.ts

Name	Tooltip
`log`	`(message: string) => void`
`message`	`string`
`window.console`	`Console`

Name	Tooltip
testLog	string

You can test the static typing by calling the log method with different values. Your development tools should highlight the erroneous calls that fail to pass a `string` argument. This will work in all cases where there is a type, whether it is inferred by the compiler or explicitly declared in your code.

Sample4.ts Function Call Type Checking

```
// allowed
log("Hello world");

// not allowed
log(1);
log(true);
log({ 'key': 'value' });
```

The level of type declarations you add yourself is a matter of taste; you could explicitly declare the types everywhere in your code, but I personally find the resulting code too verbose, and in many cases adding a type declaration is redundant when the type is already obvious. The next two examples compare two approaches to help illustrate this subject. The first is an entirely explicit version of the logging code and the second applies a more pragmatic level of declarations.

Sample5.ts Entirely Explicit Types

```
function log(message: string): void {
    if (typeof window.console !== 'undefined') {
        window.console.log(message);
    }
}

var testLog: string = "Hello world";

log(testLog);
```

Sample6.ts Pragmatic Explicit Types

```
function log(message: string): void {
    if (typeof window.console !== 'undefined') {
        window.console.log(message);
    }
}

var testLog = 'Hello world';

log(testLog);
```

In the second example, I explicitly state the type for any variable or parameter that cannot be inferred by the compiler and additionally make the return type explicit for functions, which I find makes the function more readable. I haven't specified the type for the `testLog` variable as it is obvious that it is a `string` because it is initialized with a string literal. I will go into more detail in the section *When to Use Types* at the end of this chapter.

Built-in Types

TypeScript has five built-in primitive types as well as the dynamic `any` type and the `void` return type. The `any` type can have any value assigned to it, and its type can be changed at runtime, so it could be initialized with a `string` and later overwritten with a `number`, or even a custom type. The void return type can only be used to declare that a function does not return a value.

The following list contains the five built-in primitive types.

Table 3: Primitive Types

Name	Tooltip
`number`	The `number` type is equivalent to the primitive JavaScript number type, which represents double-precision 64-bit floating-point values. If you are used to the distinction of integer types and non-whole numbers, there is no such distinction in TypeScript.
`boolean`	The `boolean` type represents a value that is either true or false.

Name	Tooltip
`string`	The `string` type represents sequences of UTF-16 characters.
`null`	The `null` type always has the value of the null literal. This is a special sub-type that can be assigned to a variable of any type except for undefined or void.
`undefined`	The `undefined` type always has the value of the undefined literal. This is also a special sub-type, but unlike `null`, it can be assigned to any type.

Sample7.ts Use of Null and Undefined

```
// allowed
var a: string = null;
var b: number = undefined;

// not allowed
var c: null;
var d: undefined;

// has a value of undefined
var e: string;
```

In this example I have demonstrated that you can assign `null` or `undefined` to a variable with an explicit type, but you cannot declare something to be of type `null` or `undefined`.

Custom Types

You are not restricted to the built-in types. You can create your own types using modules, interfaces, and classes, and use these in your type declarations. In this example I haven't even created a definition for the interface, or a body for the class, but they can already be used as valid types within the type system. In addition, the type system works with polymorphism, which I discuss in more detail in *Inheritance and Polymorphism*.

Sample8.ts Custom Types Using Interfaces and Classes

```
interface ILogger {
}

class Logger {
}

var loggerA: ILogger;
var loggerB: Logger;
```

It is technically possible to use a module in a type declaration, but the value of this is limited as the only thing you could assign to the variable is the module itself. It is rare to pass a whole module around your program and much more likely that you will be passing specific classes, but it is worth knowing that it can be done.

Sample9.ts Using a Module as a Type

```
module MyModule {
}

var example: MyModule = MyModule;
```

Advanced Type Declarations

The type definitions I have described so far are unlikely to be enough when it comes to writing a program. TypeScript has advanced type declarations that let you create more meaningful types, like statically typed arrays or callback signatures. There is a distinct cognitive shift from .NET type declarations for these, which I will explain as we look at each case.

The first advanced type declaration allows you to mark the type of an array by adding square brackets to the type declaration. For example, an array of strings would have the following type declaration:

```
var exampleA: string[] = [];
```

Unlike other languages you may be using, the variable is initialized using either the array literal of empty square brackets `[]` or the `new Array(10)` constructor if you wish to specify the array length. The type is not used on the right-hand side of the statement.

The second advanced type declaration allows you to specify that the type is a function. You do this by surrounding the definition in curly braces; for example, a function accepting a `string` parameter and not returning any value would have the following type declaration:

```
var exampleA: { (name: string): void; } = function (name: string) { };
```

If the function has no parameters, you can leave the parenthesis empty, and similarly you can specify a return value in place of the `void` return type in this example.

In all of these cases, the compiler will check assignments to ensure they comply with the type declaration, and autocompletion will suggest relevant operations and properties based on the type.

Here are some examples of advanced type declarations in action:

Sample10.ts Advanced Type Declarations

```
class Logger {
}

// exampleA's type is an array of Logger objects.

var exampleA: Logger[] = [];
exampleA.push(new Logger());
exampleA.push(new Logger());

// exampleB's type is a function.
// It accepts an argument of type string and returns a number.

var exampleB: { (input: string): number; };

exampleB = function (input: string) {
      return 1;
```

```
};

// exampleC's type is an array of functions.
// Each function accepts a string and returns a number.

var exampleC: { (input: string): number; } [] = [];

function exampleCFunction(input: string) : number {
      return 10;
}

exampleC[0] = exampleCFunction;

exampleC[1] = exampleCFunction;
```

Type Inference Mechanism

I explained a little about type inference at the start of this chapter, but the purpose of this section is to clarify exactly how it works in TypeScript. Inference only takes place if you haven't explicitly stated the type of a variable, parameter, or function.

Variables and Parameters

For variables and parameters that are initialized with a value, TypeScript will infer the type from the initial value. For example, TypeScript infers the correct types for all of the following examples by inspecting the initial value even if the initial value comes from another variable:

Sample11.ts Type Inference

```
class ExampleClass {
}

function myFunction(parameter = 5) { // number
}

var myString = 'Hello World'; // string
```

```
var myBool = true; // boolean
var myNumber = 1.23; // number
var myExampleClass = new ExampleClass(); // ExampleClass
var myStringArray = ['hello', 'world']; // string[]


var anotherBool = myBool; // boolean
```

If you don't initialize the variable or parameter with a value, it will be given the dynamic `any` type, even if you assign a value of a specific type to it later in the code.

Sample12.ts Inferred Any Type

```
var example; // any


example = 'string'; // still any
```

Functions

Type inference for functions usually works upwards from the bottom, determining the return value based on what is being returned. When writing a function, you will be warned if you have conflicting return types—for example, if one branch of code returns a string and another branch of code returns a number.

Sample13.ts Incompatible Return Types

```
function example (myBool: boolean) {
      if (myBool) {
            return 'hello'; // string
      }
      return 1; // number
}
```

Contextual Typing

In some situations, TypeScript uses the contexts within which an expression occurs to determine the types. For example, if a function expression is being assigned to a variable whose type is known, the type declaration of the variable will be used to infer the types for parameters and return values.

Sample14.ts Contextual Typing

```
var func: { (param: number): number; };

func = function (a) {
    return a++;
}
```

This example demonstrates that the type of `a` is inferred to be `number`, based on the type declaration of the `func` variable on the first line.

Widened Types

Type inference uses a widened type when the initial value of a variable, parameter, or function return value is deemed to be `null` or `undefined`. In these cases, the dynamic `any` type is used.

Sample15.ts Widened Types

```
var a = undefined; // any
var b = null; // any
var c = [null, null]; // any[]
var d = { a: undefined, b: 1 }; // { a: any, b: number }
```

When to Use Types

Whether to make a type explicit or not is a matter of personal taste, so you should agree on your team's preferred style in this respect. The discussion is similar to the conversations I have heard over the use of the .NET `var` keyword. The important thing to remember is that whether a type is explicit or inferred, it is still statically typed in all cases where the type has not been widened.

The minimum level of explicit typing would be to specify types wherever the compiler would infer the `any` type either because a variable hasn't been initialized, or it has been initialized with a `null` or `undefined` value. I would recommend explicitly declaring types on interfaces, and on function parameters and return values, in addition to this minimum level.

Chapter 4 Creating New Modules

When writing a new TypeScript program, you should start by designing the high-level modules. Each class and interface you add should be placed inside an appropriately named module, and if you don't already have a module that they naturally fit into, you should consider adding a new one. If you are adding TypeScript to an application that has existing JavaScript files, there is more information on integrating the two languages in Chapter 6, "Working with existing JavaScript."

In this chapter, I will use a practical and usable example to demonstrate the features of the TypeScript language and associated tools. I will create a utilities module to house cross-cutting concerns, such as the logging example from the previous chapter.

Modules

Declaring a Module

Modules are declared with the `module` keyword. A module is a container to group together a set of related interfaces and classes. The interfaces and classes can be internal only, or made available to code outside of the module. Everything inside of a module is scoped to that module, which is preferable to adding items to the global scope.

If you are writing a large-scale application, you can nest your modules just like you nest namespaces in .NET. The aim should be to make the components of your program easy to discover using autocompletion. In this chapter I will name the program `Succinctly` and the first module `Utilities`.

Sample16.ts Utilities Module

```
module Succinctly {
    module Utilities {


    }
}
```

Typically you would write your program using one module per file, so the `Utilities` module would live inside of Utilities.ts and be compiled into Utilities.js. It is possible to add more than one module to the same file, but it may make your program more difficult to comprehend. Additionally, you can declare a module across multiple files, which is useful for creating a namespace, or for extending a third-party library.

Note: Module declarations in TypeScript can vary depending on your module loading strategy. You can read more about how this affects your program in Chapter 5, "Loading Modules."

Adding a Class to a Module

To add a class to a module, you use the `class` keyword. You can then place variables and functions inside the class.

Sample17.ts Logger Class (Utilities.ts)

```
module Utilities {


    export class Logger {
        log(message: string): void {
            if (typeof window.console !== 'undefined') {
                window.console.log(message);
            }
        }
    }
}
```

The interesting part of this example is the `export` keyword. This makes the `Logger` class visible outside of the `Utilities` module, which means you can create new instances of the `Logger` elsewhere in your code. If you omit the `export` keyword, you will only be able to use the `Logger` from other classes in the `Utilities` module.

The `log` function is public by default, but I'll go into more detail about functions later in this chapter.

Assuming you have placed this module inside a file named Utilities.ts, you can now make use of the `Logger` class elsewhere in your program by referencing the module file in a special TypeScript `reference` comment. The reference is removed from the compiled JavaScript and simply helps the development tools understand what modules and classes are available for autocompletion and type checking. You can use multiple `reference` comments, but should try to make each module depend on as few other modules as possible.

Sample18.ts Calling the Logger

```
///<reference path="Utilities.ts" />

window.onload = function () {
    var logger = new Utilities.Logger();
    logger.log('Logger is loaded');
};
```

You create a new instance of a `Logger` using the `new` keyword. The `Logger` must be referenced via its module. This gives you powerful code organization tools that closely match those available in .NET—you can imagine what the entire TypeScript program might look like:

- Utilities.Logger
- Utilities.Dates
- Messaging.Ajax
- Messaging.JsonConverter
- Messaging.XmlConverter

To test this simple example, you can run it inside a browser using an HTML file. I have manually added the two scripts to the end of the `body` tag, but I will introduce some alternative techniques in Chapter 5, "Loading Modules."

Sample19.ts HTML Page

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>TypeScript Example</title>
</head>
<body>
    <h1>TypeScript Example</h1>
    <script src="Utilities.js"></script>
    <script src="Program.js"></script>
</body>
</html>
```

If you open the browser's console, you should see the message `Logger is loaded`. To open the console, use the appropriate option for your browser:

Table 4: Browser Console Access

Browser	Tools Access
Internet Explorer	Press F12 to open the developer tools. The console can be found under the **Scripts** tab.
Firefox	Press Ctrl+Shift+K to open the standard Web Console. There are also excellent productivity extensions for Firefox, such as the Firebug add-in, which also hosts the console.
Opera	Press Ctrl+Shift+I to open Dragonfly, Opera's built-in developer tools. The console is one of the main tabs.
Safari	Press Ctrl+Alt+I to open the developer tools.
Chrome	Press Ctrl+Shift+J to open the developer tools and jump straight to the console.

Interfaces, Classes, and Functions

Private Functions

In some console windows, the date and time of the message is automatically displayed, but not in all cases. You could extend the logger to show the time automatically next to the message.

Sample20.ts Private Functions

```
module Utilities {

    export class Logger {
        log(message: string): void {
            if (typeof window.console !== 'undefined') {
```

```
                window.console.log(this.getTimeStamp() + ' - ' + message);
            }
        }

        private getTimeStamp(): string {
            var now = new Date();

            return now.getHours() + ':' +
                now.getMinutes() + ':' +
                now.getSeconds() + ':' +
                now.getMilliseconds();
        }
    }
}
```

I have added a `getTimeStamp` function using the `private` keyword. The `private` keyword tells that this function is not available outside of the `Logger` class. If you attempt to call the method outside of this class, you will get a warning.

Sample21.ts Attempting to call Private Functions

```
///<reference path="Utilities.ts" />

window.onload = function () {
    var logger = new Utilities.Logger();
    logger.getTimeStamp();
};
```

The `private` keyword can be used on functions and variables, but it is important to understand that this only protects them at design time. Once your TypeScript code has been compiled into JavaScript, there is nothing to stop them from being used, so you cannot rely on them if you are making your libraries available to other developers.

Note: Type safety, interfaces, and protection of private variables and functions only occurs at design time and compilation time in TypeScript.

Static Functions

If you run the latest example, you will see that the message is now prefixed with a timestamp, but that the format isn't quite right: `15:39:1:767 - Logger is loaded`. When the hours, minutes, or seconds are less than two digits, or when the milliseconds are less than three digits, the value should be left-padded with zeros to preserve the time stamp format.

Sample22.ts Static Functions

```
module Utilities {

    export class Logger {
        log(message: string): void {
            if (typeof window.console !== 'undefined') {
                window.console.log(this.getTimeStamp() +
                    ' - ' + message);
            }
        }

        private getTimeStamp(): string {
            var now = new Date();

            return Formatter.pad(now.getHours(), 2, '0') + ':' +
                Formatter.pad(now.getMinutes, 2, '0') + ':' +
                Formatter.pad(now.getSeconds(), 2, '0') + ':' +
                Formatter.pad(now.getMilliseconds(), 3, '0');
        }
    }

    class Formatter {
        static pad(num: number, len: number, char: string): string {
            var output = num.toString();
            while (output.length < len) {
                output = char + output;
            }
            return output;
```

```
        }
    }
}
```

I have added a `Formatter` class to the `Utilities` module that contains the function that will format the number by pre-pending zeros and returning the resulting string. The first point to note is that I have omitted the `export` keyword, which means that the `Formatter` class is only available inside of the `Utilities` module. It is not accessible anywhere else in the program.

I have done this deliberately in this case, as I don't want to expose this class outside of the `Utilities` module, but if I change my mind in the future, I can simply add the `export` keyword to expose the class. If something isn't appearing in your autocompletion list when you think it should, you are probably missing the `export` keyword.

The second new keyword I have used in this example is the `static` keyword. This keyword allows me to call the `pad` function without creating an instance of the `Formatter` class, which you would normally do like this: `var formatter = new Formatter();`.

This is now a working module that adds a neatly formatted time stamp to any message being logged. It is now possible to take a step back and re-factor the example to make it cleaner and easier to use. TypeScript has language features that you can use to clean up your use of functions, and I'll introduce them next.

The following code samples are all based on the lLogging example, but I have just included specific sections of code to shorten the examples and to avoid boring you with repetition. You can continue to follow along using the code that was created during this chapter; just slot in the updated sections as you go.

Default Parameters

By specifying a default value for a parameter, calling code can treat the parameter as optional. You may already be familiar with default parameters, which were added to C# in .NET version 4 and offer a clean alternative to overloaded methods in many cases.

To specify a default parameter, you simply add it to your function using an equal sign after the type declaration. I have added defaults to the `pad` function to specify the most common `len` and `char` input values.

Sample23.ts Default Parameters

```
// before
static pad(num: number, len: number, char: string) {


// after
static pad(num: number, len: number = 2, char: string = '0') {
```

You no longer need to pass these arguments to the pad function, unless you want to use a value that is different from the default, so you can update your calling code to make it much cleaner and less repetitive. Specifically, you need to pass the `num` argument in all cases, because it isn't sensible to specify a default value for this parameter. You only need to pass the `len` argument in the case of milliseconds, which should be three characters long, and you don't need to pass the `char` argument at all.

Sample24.ts Calling a function that has default parameters

```
// before
return Formatter.pad(now.getHours(), 2, '0') + ':' +
    Formatter.pad(now.getMinutes, 2, '0') + ':' +
    Formatter.pad(now.getSeconds(), 2, '0') + ':' +
    Formatter.pad(now.getMilliseconds(), 3, '0');


// after
return Formatter.pad(now.getHours()) + ':' +
    Formatter.pad(now.getMinutes()) + ':' +
    Formatter.pad(now.getSeconds()) + ':' +
    Formatter.pad(now.getMilliseconds(), 3);
```

Optional Parameters

Optional parameters are similar to default parameters, but are useful when you don't actually need an argument to be passed to carry out the operation. You mark a parameter as optional by appending a question mark to its name.

Sample25.ts Optional Parameters

```
static pad(num: number, len: number = 2, char?: string) {
    if (!char) {
        char = '0';
    }
```

You can check for the presence of the `char` argument using a simple `if` block. In this example, I set a value if one hasn't been passed, which makes it behave just like a default parameter—but you could implement different logic flows based on whether optional parameters have been passed as arguments by calling code.

Rest Parameters

Rest parameters are placeholders for multiple arguments of the same type. To declare a rest parameter, you prefix the name with three periods. When you call a function that has a rest parameter, you are not limited to the number of arguments you pass, but they must be of the correct type.

Sample26.ts Rest Parameters

```
function addManyNumbers(...numbers: number[]) {
    var sum = 0;
    for (var i = 0; i < numbers.length; i++) {
        sum += numbers[i];
      }
    return sum;
}

var result = addManyNumbers(1,2,3,5,6,7,8,9); // 41
```

The `...numbers` parameter is declared as a rest parameter, so when we call `addManyNumbers` you can pass as many arguments as you need to. Any non-rest parameters should come before the rest parameter.

Sample27.ts Mixed Normal and Rest Parameters

```
function addManyNumbers(name: string, ...numbers: number[]) {
```

Function Overloads

If you can solve your function definition requirements using default parameters and optional parameters, I recommend that you use these instead of function overloads. The function overload syntax in TypeScript works very differently from method overloads in .NET. Rather than specifying multiple methods with different signatures, TypeScript allows a single function to be decorated with multiple signatures.

To illustrate one potential use for function overloads, imagine that the `pad` function could be called either with a `string` to pad, or a `number` to pad.

Sample28.ts Function Overloads

```
static pad(num: number, len?: number, char?: string);
static pad(num: string, len?: number, char?: string);
static pad(num: any, len: number = 2, char: string = '0') {
    var output = num.toString();
    while (output.length < len) {
        output = char + output;
    }
    return output;
}
```

I have added two function overload signatures above the original function. The first specifies that `num` is a `number`, and the second specifies that `num` is a `string`. Both the `len` and `char` parameters are marked as optional to allow calling code to omit them, in which case the default values specified in the original function declaration will apply.

The original `pad` function has been updated to specify a `num` parameter using the `any` type. This could be either a string or a number because of the two overloads, so I have used the `any` type to accept either of these, making `num` a dynamic parameter.

The function body remains untouched, as it was already converting the `num` parameter into a string. It just means that this call is redundant when the supplied argument is already a string. The actual function cannot be called when it is decorated with function overloads, so it is not possible to pass a value that isn't a `number` or a `string`, even though the original function now accepts the dynamic `any` type, so it is more restricted and predictable to use function overloads than it is to simply change a parameter type to `any`.

I personally find this code block harder to read than two separate methods—for example `padString` and `padNumber`—so you may want to consider multiple functions as an alternative solution.

Sample29.ts Alternative to Function Overloads

```
static padNumber(num: number, len?: number, char?: string) {
    return padString(num.toString(), len, char);
}

static padString(input: string, len: number = 2, char: string = '0') {
    var output = input;
    while (output.length < len) {
```

```
        output = char + output;
    }
    return output;
}
```

Constructors

Constructors in TypeScript are similar to .NET constructors and allow you to specify the dependencies and data that your class needs in order to work. Because you can't create a new instance of the class without these things, you can guarantee they will be available in any non-static functions.

I will update the `Logger` class to remove its reliance on the static `Formatter` class and instead require an instance of a `Formatter` to be passed into the constructor.

Sample30.ts Constructor

```
export class Logger{
    constructor (private formatter: Formatter) {
    }

    log(message: string): void {
        if (typeof window.console !== 'undefined') {
            window.console.log(this.getTimeStamp() +
                ' - ' + message);
        }
    }

    private getTimeStamp(): string {
        var now = new Date();

        return this.formatter.pad(now.getHours()) + ':' +
            this.formatter.pad(now.getMinutes()) + ':' +
            this.formatter.pad(now.getSeconds()) + ':' +
            this.formatter.pad(now.getMilliseconds(), 3);
    }
}
```

In the constructor, you can use either the `public` or `private` keyword to initialize the parameters and make them visible externally or internally, respectively. Unlike .NET constructors, you do not need to manually map the argument passed in the constructor onto a separate field as the compiler does this for you. You can then access the class-level variable using the `this` keyword.

To create an instance of the `Logger` class, you are required to pass in a `Formatter`. The development tools and compiler will check that you do.

Sample31.ts Calling a Constructor

```
var formatter = new Utilities.Formatter();
var logger = new Utilities.Logger(formatter);
```

Remember, you'll need to add the `export` keyword to the formatter.

Interfaces

The logging example created throughout this chapter is now in a pretty good state. The next stage is to create an interface for the `Logger` so that different implementations can be supplied, such as one that raises messages to the user if there is no console attached, or one that sends errors back to the server to be logged.

Interfaces are declared using the `interface` keyword and you can make an interface available outside of a module using the `export` keyword, just like you can with a class.

Sample32.ts ILogger Interface

```
export interface ILogger {
    log(message: string): void;
}
```

To implement the interface on the Logger class, you use the `implements` keyword. TypeScript will now check that the class has the required variables and functions that are promised by the interface, and it will raise a warning if anything hasn't been implemented, or if the implementation doesn't correctly match the signature of the interface.

Sample33.ts Implementing the ILogger Interface

```
export class Logger implements ILogger {
```

It is now possible to write new logging classes that also implement the `ILogger` interface safe in the knowledge that they can be used in place of each other wherever an `ILogger` is required. This will not affect the calling code because you can guarantee the concrete classes are compatible.

Sample34.ts New Logger Class

```
export class AnnoyingLogger implements ILogger {
    log(message: string): void {
        alert(message);
    }
}
```

Multiple Interfaces

It is allowable in TypeScript for a class to implement more than one interface. To do this, separate each interface with a comma:

Sample35.ts New Logger Class

```
export class MyClass implements IFirstInterface, ISecondInterface {
```

Duck Typing

An interesting feature of TypeScript interfaces is that they do not need to be explicitly implemented; if an object implements all of the required variables and functions, it can be used as if it did explicitly implement the interface. This is called duck typing, which you may have come across if you have used .NET's iterator pattern.

I will use a new example to demonstrate this feature in isolation. Where an `IPerson` interface requires a `firstName` and `lastName` variable, you can create an anonymous object that TypeScript will accept as an `IPerson` type, even though it doesn't claim to implement the interface. This also works if a class satisfies the requirements of the interface.

Sample36.ts Duck Typing

```
interface IPerson {
    firstName: string;
    lastName: string;
}

class Person implements IPerson {
    constructor (public firstName: string, public lastName: string) {

    }
```

```
}

var personA: IPerson = new Person('Jane', 'Smith'); // explicit
var personB: IPerson = { firstName: 'Jo', lastName: 'Smith' }; // duck typing
```

Inheritance and Polymorphism

I will now return to the logging example in order to demonstrate inheritance and polymorphism. When I added the `AnnoyingLogger` class, it could already be used in place of the original `Logger`. This is the essence of polymorphism, whereby you can substitute any particular implementation of the logging class without affecting the calling code.

You can also achieve polymorphism using inheritance, which is useful if you want to alter only parts of the original class, or if you want to add additional behavior over and above what is offered by the original class. For instance, if you wanted to raise a visible message and write to the console, you could use inheritance, rather than call both loggers in turn.

Sample37.ts Inheritance and Polymorphism

```
export class AnnoyingLogger extends Logger {
    log(message: string): void {
        alert(message);
        super.log(message);
    }
}
```

To inherit from a class, you use the `extends` keyword. The `log` function alerts the user before calling the base `Logger` class which you can access with the `super` keyword. The result of this function is that an alert dialog will be displayed, and when it is closed, a message will be added to the console, if one is running.

Sample38.ts Calling the Logger

```
///<reference path='Utilities.ts'/>

window.onload = function () {
    var logger: Utilities.ILogger = new Utilities.AnnoyingLogger();
    logger.log('Logger is loaded');
};
```

Multiple Inheritance

A class can only inherit from one other class; it is not possible to inherit from multiple classes. This is the same restriction you will find in .NET and it avoids many pitfalls, such as confusion over the base class when `super` is called, and the famous Diamond of Death.

One interesting feature that TypeScript has that you won't find in .NET is the ability for an interface to inherit from another interface. Just like class inheritance, you use the `extends` keyword to do this, but the purpose is slightly different. Rather than attempting to override or extend the behavior of a base class, you would use interface inheritance to compose interface groups. Unlike classes, interfaces can inherit from multiple interfaces.

Sample39.ts Interface Inheritance

```
interface IMover {
    move() : void;
}

interface IShaker {
    shake() : void;
}

interface IMoverShaker extends IMover, IShaker {

}

class MoverShaker implements IMoverShaker {
    move() {
    }

    shake() {
    }
}
```

Because the `IMoverShaker` interface inherits from both `IMover` and `IShaker`, the class must implement both the `move` function and the `shake` function. This is a powerful feature that allows you to compose a number of interfaces into a super interface.

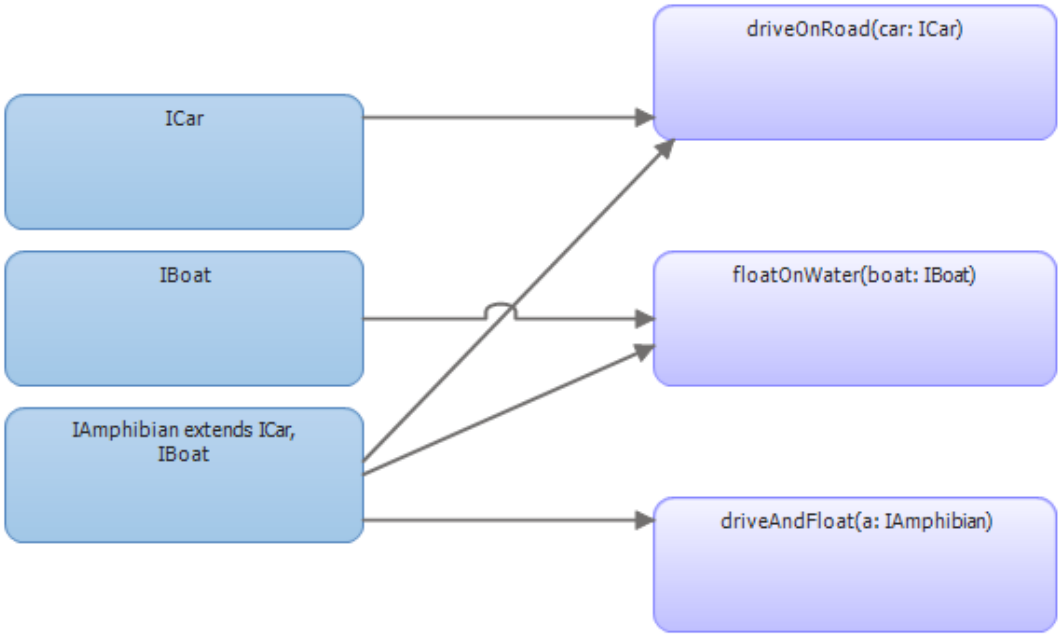

Figure 7: A Super Interface

The super interface in this diagram is the `IAmphibian` interface. A class that implements this super interface can do everything shown on the right-hand side of the diagram. Without the super interface, it would be hard to express that a class needed both `IBoat` and `ICar` interfaces to be valid for a given operation.

Using instanceof

The `instanceof` keyword can be used to compare an instance to a known type. It will return true if the instance is of the known type, or inherits from the known type. For example, the `AnnoyingLogger` class is an instance of `AnnoyingLogger`, and it is also an instance of `Logger`.

Sample40.ts Instance Of

```
var isLogger = logger instanceof Utilities.Logger; // true
var isLogger = logger instanceof Utilities.AnnoyingLogger; // true
var isLogger = logger instanceof Utilities.Formatter; // false
```

Chapter Summary

I know that this has been an epic chapter, but you now know everything there is to know about TypeScript modules, interfaces, classes, and functions, including how to organize your code; set the visibility of classes, functions and variables; and how to use object-orientation in your TypeScript program design.

Chapter 5 Loading Modules

There are different ways of packaging your program depending on where you are planning on running it. It is easier if you decide on your packaging strategy up front as it may lead to a subtly different program structure. So far in this book I have used examples that suit the first of the packaging techniques: bundling. This is the technique you are most likely to use if you are writing applications using ASP.NET MVC (bundling was introduced in version 4).

The main differences packaging can make to your TypeScript program are:

- Module declaration style
- Module referencing style
- Script loading style

I will describe how to do these three things for each of the packaging techniques in this chapter.

Bundling

Bundling is an increasingly common strategy for improving the performance of websites and web applications. When you reference multiple JavaScript files on a webpage, each file is a separate HTTP request that typically spends more time queuing than downloading. This makes the page appear to load slowly. By combining all of your JavaScript files into a single release file, you eliminate most of the queuing time, which makes users perceive that the webpage loads much faster.

Yahoo's Exceptional Performance team puts the goal of minimizing HTTP requests at the top of its *Best Practices for Speeding up Your Web Site* list and suggests bundling as the solution for script files. The list can be found here: http://developer.yahoo.com/performance/rules.html.

Module Declaration Style

To use a bundling strategy in your program, you can declare the modules as you have seen in the examples in this book.

MyModule.ts

```
module MyModule {
    export class MyClass {


    }
}
```

Module Referencing Style

TypeScript has a commenting system that allows you to make modules in different files available throughout your program. In the comment, you specify the path to the file you want to include relative to the directory of the current source file.

Main.ts

```
/// <reference path="MyModule.ts" />


var myClass = new MyModule.MyClass();
```

The `reference` comment promises the compiler that the file will be available at run time, so it allows the use of the module and classes defined inside of the *MyModule.ts* file. You will get all of the autocompletion and type checking at design time based on the `reference` comments you include. You can use these comments throughout your TypeScript program wherever one file depends on another, and you can add multiple `reference` comments if you need to.

Script Loading Style

When using bundling, TypeScript leaves the method of combining scripts to you. This allows you to take advantage of the built-in ASP.NET MVC bundling feature. In ASP.NET MVC 4, you can configure your scripts in *BundleConfig.cs*, which is in the *App_Start* folder of your application. In this file, reference the compiled JavaScript files, which will have a .js file extension.

BundleConfig.cs

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/MyProgram").Include(
            "~/Scripts/MyModule.js",
            "~/Scripts/Main.js"));
    }
}
```

If you want to include all scripts in a given directory, you can use a wildcard character in your bundle registration, although this will not guarantee any ordering like an explicit bundle would.

Bundle Using Wildcard Character

```
public class BundleConfig
{
    public static void RegisterBundles(BundleCollection bundles)
    {
        bundles.Add(new ScriptBundle("~/bundles/MyProgram").Include(
            "~/Scripts/*"));
    }
}
```

To add the bundle to your page, use the built-in `Scripts` helper to render the bundle in your view.

Using the Bundle

```
@Scripts.Render("~/bundles/MyProgram")
```

Summary

TypeScript fits in neatly with ASP.NET MVC bundling. This doesn't mean you have to use MVC to use TypeScript with bundling. You can use a number of free tools to combine your compiled JavaScript files such as YUI Compressor, JSMin, or Google's Closure Compiler.

Whenever you add a `reference` comment, let that trigger a reminder in your head to add the script to the bundle or the webpage if it isn't already included.

CommonJS Modules

CommonJS is intended to provide a common standard library for JavaScript that makes it easier to run as a server-side language, command-line tool, or desktop application. The most famous implementation of CommonJS is the highly scalable Node.js, which runs on Google's V8 JavaScript engine.

http://www.commonjs.org/

If you are targeting a platform such as Node.js, which implements a CommonJS runtime module loader, you can tell the TypeScript compiler to generate code that works out of the box with CommonJS. To do this, add a module switch to the TypeScript compiler command in your project file. You can also use this switch outside of Visual Studio, which I describe in *Appendix A: Alternative Development Tools*.

```
  <Exec
    Command="&quot;$(PROGRAMFILES)\Microsoft SDKs\TypeScript\tsc&quot; --
module commonjs @(TypeScriptCompile ->'&quot;%(fullpath)&quot;', ' ')" />
```

Module Declaration Style

There are currently three draft specifications of CommonJS, and although version 1.0 has been superseded by version 1.1, Node.js hasn't yet signed up to the changes, so I have based my examples on version 1.0.

To make a module available to a CommonJS implementation, you rely on the file name rather than a module declaration. Because the module is implicitly named after the file, you can simply export the classes and functions that you want to expose externally. If you are using CommonJS, you can organize your TypeScript program using folders and files that represent modules.

MyModule.ts

```
export class MyClass {


}
```

Module Referencing Style

Rather than using comments to make a module available, the CommonJS pattern uses an `import` declaration to load the module. The import statement is automatically compiled into a JavaScript `requires` statement, which is how you ask CommonJS to load a module.

Main.ts

```
import myModule = module('MyModule'); // MyModule.ts


var myClass = new myModule.MyClass();
```

Script Loading Style

All of the scripts required by your *main.js* file will be automatically loaded by the CommonJS library.

Summary

TypeScript's CommonJS support is tailored for platforms such as Node.js, although there are many more implementations that you may not have heard of, such as Narwhal and CouchDB.

AMD Modules

AMD stands for asynchronous module definition and is an API that allows your modules to be loaded asynchronously on an as needed basis. This is actually a pattern adapted from CommonJS intended to be used in web browsers.

To use AMD, you will need to add a module-loader to your program, such as RequireJS.

http://requirejs.org/

To get AMD output from the TypeScript compiler, set the module mode to "amd" in your project file. If you aren't using Visual Studio, you can find more information to help you in *Appendix A: Alternative Development Tools*.

```
  <Exec
    Command="&quot;$(PROGRAMFILES)\Microsoft SDKs\TypeScript\tsc&quot; --
module amd @(TypeScriptCompile ->'&quot;%(fullpath)&quot;', ' ')" />
```

Module Declaration Style

Just like CommonJS modules, your files become modules automatically, and the file name without the extension becomes the module name.

MyModule.ts

```
export class MyClass {


}
```

Module Referencing Style

Modules are referenced identically to the CommonJS style, using the `import` declaration. Although your TypeScript file looks identical to the CommonJS TypeScript file, the compiled JavaScript is very different.

Main.ts

```
import myModule = module('MyModule'); // MyModule.ts

var myClass = new myModule.MyClass();
```

Script Loading Style

Because RequireJS loads all of your modules as required, you only need to add a `script` tag for the *require.js* JavaScript file, and specify your main program file using the `data-main` attribute. When the RequireJS script loads, it will automatically load the file you specify—in this case *Main.js*, located in the Scripts folder. As further modules are required, they will be loaded for you behind the scenes.

```
<script src="Scripts/require.js" data-main="Scripts/Main"></script>
```

Chapter Summary

If you are designing a program that will run on the server, as well as in a browser, you can write your modules in the CommonJS style, and use the TypeScript compiler to give you the CommonJS and the AMD flavors of compiled JavaScript without having to change your code. While your TypeScript remains unchanged between the two platforms, the CommonJS output will use the built-in `require` function, while the AMD output will call RequireJS.

Chapter 6 Working with Existing JavaScript

It is inevitable that at some point you will need to work with an existing JavaScript library in your TypeScript program, whether it is your own or from a third party. The good news is that TypeScript has a mechanism for providing a definition of an external script, which allows you to get all of the type safety and autocompletion that you would have if it had been written in TypeScript. The files that describe an external resource are called ambient declarations, and TypeScript comes preloaded with definitions for the Document Object Model and native JavaScript APIs.

If you are referencing a popular open-source library or framework, you may find that there is already a TypeScript definition for it—but for your own existing code or less common external scripts, you may have to roll your own definition. If you are planning to convert your JavaScript to TypeScript, you may find that creating ambient declarations helps you to gradually migrate your code base.

Creating Ambient Declarations

The building block of ambient declarations is the `declare` keyword. TypeScript uses information supplied by a declaration to add design-time support and compile-time checking, but doesn't emit any JavaScript for a declaration. You can start by simply telling TypeScript that a variable is being supplied by an outside source. This will prevent errors and warnings being generated for the variable, but won't receive any additional type checking just yet.

Sample41.ts Simple Ambient Declaration

```
declare var externalLogger;


externalLogger.anything();
```

By using the `declare` keyword, TypeScript will allow you to call any variables or functions on the `externalLogger`, which makes it an ideal first step when integrating an existing JavaScript file in your program. There is no check that the `anything` function exists, or if you are passing the correct arguments, because TypeScript has no way of knowing about these—but as a quick fix, it is now possible to use the `externalLogger` variable without any design-time or compilation-time errors in the program.

When using an ambient declaration, you will need to reference the JavaScript file manually if you are using bundles. If you are using CommonJS or AMD modules, the ambient declaration should be in the same location as the actual library. If you are using bundling, you can add it to the same bundle as your TypeScript output.

Ideally, you should create a more descriptive ambient declaration, which will give you all of the tooling support associated with TypeScript. You can explicitly define the functions and variables that the external script supports.

Sample42.ts Ambient Class Definition

```
declare class ExternalLogger {
    log(message: string): void;
}

declare var externalLogger: ExternalLogger;

externalLogger.anything();
```

You will notice that the `declare` keyword allows you to treat the body of the `ExternalLogger` class as if it were an interface and the `log` function needs no implementation, as this is in the external JavaScript file. By explicitly defining everything you need from the external script, you will have autocompletion and type checking in your development tool.

Using this technique allows you to be tactical about porting existing scripts, as you can hide them behind ambient declarations, and you can choose to convert the files that change most often, leaving the rest for later. You can also opt to declare just the parts of an external class that you want to use in your program, therefore hiding the parts you don't need or want to deprecate.

Sample43.ts Complete Ambient Declaration

```
declare module ExternalUtilities {
    export class ExternalLogger {
        public totalLogs: number;
        log(message: string): void;
    }
}

var externalLogger = new ExternalUtilities.ExternalLogger();

externalLogger.log("Hello World");
var logCount = externalLogger.totalLogs;
```

You can place your ambient declaration in a separate file. If you do so, you should use a .d.ts file extension, for example, *ExternalLogger.d.ts*. This file can then be referenced or imported just like any other TypeScript file.

Sample44.ts Referencing Ambient Declaration File

```
/// <reference path="ExternalUtilities.d.ts" />
var externalLogger = new ExternalUtilities.ExternalLogger();
externalLogger.log("Hello World");
```

Note: The TypeScript Language Specification notes that any code inside a file with a .d.ts file extension will implicitly be treated as if it has the declare keyword, but at the time of writing, Visual Studio requires the declare keyword to be present.

It is possible to declare modules, classes, functions, and variables by prefixing the block of code with the `declare` keyword and by omitting the implementations. The names you use should match the exact names in the JavaScript implementation.

Porting JavaScript to TypeScript

I won't try to dictate a strategy for porting legacy JavaScript into TypeScript, or even tell you that you must do so. I will briefly explain some of the available options and present a candidate plan for porting the code, but your chosen approach will depend entirely on your unique situation.

I will assume that you have decided to use TypeScript for new code, but have a number of existing libraries. Your first step is to create ambient declarations for the parts of the existing code you want to call from your new TypeScript program. Start by defining only the parts of the existing code that you are using—this is your opportunity to remove redundant code as well as convert what you do need to TypeScript.

To select the files to start rewriting, look for the JavaScript files that change most often. The more changes that will be made, the more benefit you will get from design-time tooling and compile-time checking. If a file is stable, leave it where it is for the time being. Another potential driver for the order in which you move your code will be where you find dependency chains. You may find it easier to move slices of code along with the code it depends on.

Ideally, you would ensure that you were calling legacy JavaScript from TypeScript, but not the other way around. If you rewrite your existing JavaScript to call the code generated by the TypeScript compiler, you will find it hard to make changes because subtle structural changes in your TypeScript could easily break the old code that relies on it. When all of your code has been converted to TypeScript, it would be trivial to make these changes because the compiler will validate the calling code, and you will have access to all the standard refactoring tools.

The process for porting a JavaScript file is relatively simple. Start by placing the JavaScript straight into a TypeScript file. You will need to fix any errors to make the code valid. Once you have a working file, look for cases where the `any` type is being used, and try to narrow the type by specifying it explicitly. In some cases, you won't be able to narrow the type because a variable is being used to store different types at run time, or because a function returns different types in different logical branches. If you can re-factor out these usages, do so. If they are genuine candidates for a dynamic type, you can leave them in. You can start adding interfaces at this stage if you need to provide a contract for custom objects in the code.

The next step is to add structure to your file using modules and classes. Finally, you should look for candidates to make less accessible; for example, by attempting to remove any unnecessary `export` keywords, and converting `public` variables to `private` variables where they aren't accessed externally.

All of this is much easier if you already have unit tests for your JavaScript, as you can be more confident that the changes won't break expected behavior—but if you don't have unit tests, this may be the ideal time to start adding them to your project. I will discuss this topic in more detail in Chapter 7, "Unit testing with TypeScript."

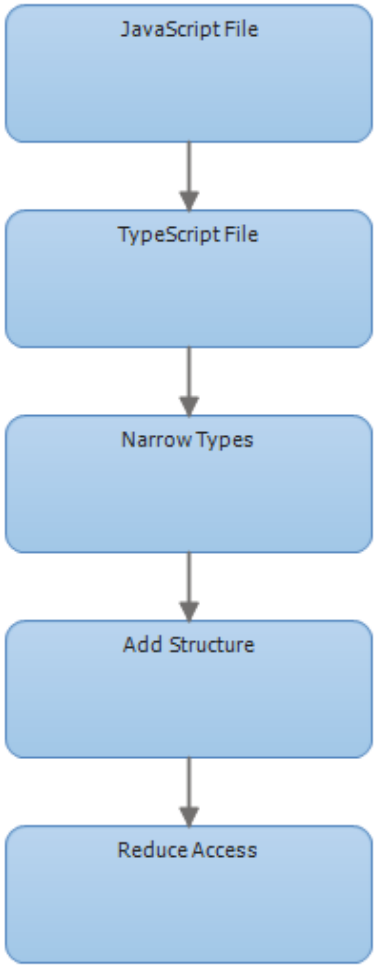

Figure 8: Conversion Process

Transferring Design Patterns

As well as transferring your JavaScript into TypeScript, there are many things you can transfer to TypeScript from your .NET code, for example your good habits, best practices, and design patterns.

Whenever you read a book or article about object-oriented programming, SOLID principles, design patterns, and clean code, you can now think of TypeScript as well as C#, VB.NET, and Java.

When you are transferring your JavaScript to TypeScript, don't leave in the bad practices you come across. Apply your .NET programming skills to transform it into clean, maintainable code.

Chapter 7 Unit Testing with TypeScript

Testing with tsUnit

If you want a unit testing framework that is written in the language you are writing, tsUnit is the choice for TypeScript. The tsUnit testing framework is written as a single module that has a test runner, assertions, and lightweight fakes. The project has been released on CodePlex under the same Apache 2.0 license as TypeScript.

http://tsunit.codeplex.com/

Setting it Up

Setting up tsUnit is quick and easy. You can actually start using it with just one file, but I recommend you get the following three files from the **SOURCE CODE** tab on CodePlex:

- tsUnit.ts
- app.css
- default.htm

The *tsUnit.ts* file is the actual unit testing framework; *app.css* and *default.html* form a nice template for displaying test results in a web browser. You can add these to any TypeScript program to start testing.

Note: The tsUnit framework is fully self-tested, so you can view example tests in the Tests.ts file and in the deliberately failing BadTests.ts file on CodePlex.

If you are using either CommonJS or AMD modules in your TypeScript program, you will need to remove the enclosing module in the *tsUnit.ts* file, as described in Chapter 5, "Loading Modules."

Code under Test

To demonstrate tsUnit tests, I will use the following example module, which is a classic test example that takes two numbers and returns their sum. I have used a simple example so you can spend more time thinking about the tests and less time working out what the example does.

Calculations.ts

```
module Calculations {
    export class SimpleMath {
```

```
        addTwoNumbers(a: number, b: number): number {
            return a + b;
        }
    }
}
```

The `SimpleMath` class has no dependencies, but if it did you could create test doubles to use in place of real code to make your tests more predictable. I will return to the subject of test doubles later in this chapter.

Writing a tsUnit Test

In order to test the `Calculations` module, you need to reference or import both *Calculations.ts* and *tsUnit.ts*. All of your tests are grouped within a test module. Organizing unit tests into modules and classes keeps them out of the global scope and also allows the unit testing framework to perform auto-discovery on a class to find all of the functions to run. This structure is similar to the style used by frameworks, such as MSTest or nUnit when testing .NET code.

CalculationsTests.ts – Empty Structure

```
/// <reference path="Calculations.ts" />
/// <reference path="tsUnit.ts" />

module CalculcationsTests {

}
```

Technically, each test class should implement the `tsUnit.ITestClass` interface, but TypeScript is smart enough to convert any class you write to the empty `ITestClass` interface, so you don't need to specify it explicitly. All you need to do is supply functions that tsUnit will find and run for you.

CalculationsTests.ts – Test Class

```
/// <reference path="Calculations.ts" />
/// <reference path="tsUnit.ts" />

module CalculcationsTests {
    export class SimpleMathTests {
        addTwoNumbers_3and5_8(context: tsUnit.TestContext) {
```

```
            // arrange
            var math = new Calculations.SimpleMath();

            // act
            var result = math.addTwoNumbers(3, 5);

            // assert
            context.areIdentical(8, result);
        }
    }
}
```

I have named the test class `SimpleMathTests`, but I sometimes write a separate class for each function under test if there are a lot of tests for that specific function. You can begin by adding your tests to a general test class, and then split them if you need to. I have only added one test so far, using Roy Osherove's naming convention, *functionName_scenario_expectation*.

The function name `addTwoNumbers_3and5_8` tells anyone reading the test output that the test is calling a function named "addTwoNumbers" with arguments 3 and 5, and is testing that the result is 8. When a test fails, a test function with a name that describes exactly what is being tested is a big help, as long as the function name is correct.

The only special part of this function that comes from tsUnit is the parameter: `context: tsUnit.TestContext`. When the tests run, the tsUnit test runner will pass in a `TestContext` class that contains functions you can use to test your results, and either pass or fail the test. These are mostly shortcuts for code that you might write yourself. They don't do anything magical, they just make your tests shorter and easier to read. I have summarized the test context assertions that are available later in this chapter.

The test function is written using the Arrange Act Assert test structure, which is also known as Triple-A test syntax.

- Arrange—perform the tasks required to set up the test.
- Act—call the function you want to check.
- Assert—check that the result matches your expectation.

It sometimes helps to use the Arrange Act Assert comments to remind you of this structure until it becomes second nature. The comments don't have any special meaning and are not needed by TypeScript or tsUnit.

You can now register your tests with tsUnit by passing it an instance of your test class. All of the functions in the test class are automatically detected.

```
// Create an instance of the test runner.
```

```
var test = new tsUnit.Test();
// Register the test class.
test.addTestClass(new CalculcationsTests.SimpleMathTests());
// Run the tests and show the results on the webpage.
test.showResults(document.getElementById('result'), test.run());
```

For the purpose of this example, you can add this section to the end of the *CalculationsTests.ts* file. In a proper TypeScript program, you would have many test modules and would compose them in a test registration file.

The `addTestClass` function can be called many times to add all of your test classes to a single test-runner instance. I will discuss a pattern for composition later in this chapter that moves the registration to each test module, which makes it easier to keep the registration up to date.

This example uses the default HTML output, but it is possible to work with the raw result data in tsUnit. This can be done by calling the `run` function and using the returned `TestResult` class to either log the errors, or present them in a custom format to use in your build system.

```
var testResult = test.run();
for (var i = 0; i < testResult.errors.length; i++) {
    var err = testResult.errors[i];
    console.log(err.testName + ' - ' + err.funcName + ' ' + err.message);
}
```

Running the Test

To run this test in a web browser, you can use the standard tsUnit HTML file, with references to your module under test and the test module.

default.html – Test Host

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="utf-8" />
    <title>tsUnit</title>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8">
    <link rel="stylesheet" href="app.css" type="text/css" />
</head>
```

```
<body>
    <div id="result">
        <h1>Awaiting result...</h1>
        <span>If this section doesn't update, check your console for
errors!</span>
    </div>
    <script src="tsUnit.js"></script>
    <script src="Calculations.js"></script>
    <script src="CalculationsTests.js"></script>
</body>
</html>
```

When you open this page in a browser, you should see the following output:

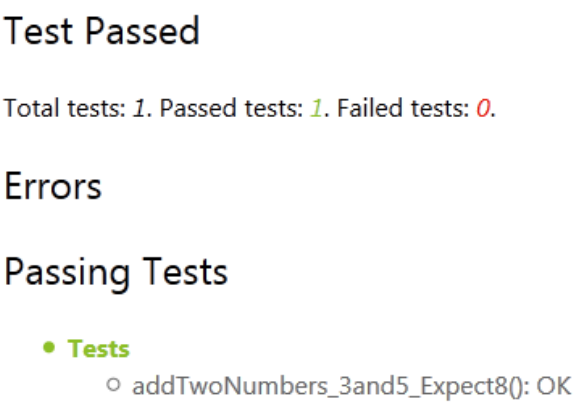

Figure 9: tsUnit Test Report

If you don't see a test result, make sure you have referenced all of the required compiled JavaScript files: in this case *tsUnit.js*, *Calculations.js,* and *CalculationsTests.js*. You can also use the web browser's error console to track down any errors.

When you add additional functions to your test class, they will appear in the test report automatically. When a test fails, it is displayed under the Errors heading, along with a detailed explanation of the test failure.

Test Composition

Code that changes for the same reason should be colocated, and in the original example there would be a need to change two different code files if a new test class were added to an existing test module. To improve cohesion, you should add the registration to your test module. By registering the classes from the module, you only need to edit one code file to add a new test class and register it to the test runner.

CalculationsTests.ts – Composer Class

```
module CalculcationsTests {
    export class Composer {
        static compose(test: tsUnit.Test) {
            test.addTestClass(new CalculcationsTests.SimpleMathTests());
        }
    }
    export class SimpleMathTests {
        addTwoNumbers_3and5_8(context: tsUnit.TestContext) {
            var math = new Calculations.SimpleMath();
            var result = math.addTwoNumbers(3, 5);
            context.areIdentical(8, result);
        }
    }
}
```

The `Composer` class has a single static method that takes in a test runner and registers test classes against it. It only registers test classes that are in the module.

The calling code is then updated to call the composer, rather than registering the test class. The test registration code would call a composer on each test module.

```
// Create an instance of the test runner.
var test = new tsUnit.Test();
// Register the test class.
CalculcationsTests.Composer.compose(test);
// Run the tests and show the results on the webpage.
test.showResults(document.getElementById('result'), test.run());
```

When you add a new test class or change an existing test class, you don't need to make any changes outside of the test module.

CalculationsTests.ts – Added Test Class

```
module CalculcationsTests {
    export class Composer {
        static compose(test: tsUnit.Test) {
            test.addTestClass(new CalculcationsTests.SimpleMathTests());
            test.addTestClass(new CalculcationsTests.ComplexMathTests());
        }
    }
    export class SimpleMathTests {
        addTwoNumbers_3and5_Expect8(c: tsUnit.TestContext) {
            var math = new Calculations.SimpleMath();
            var result = math.addTwoNumbers(3, 5);
            c.areIdentical(8, result);
        }
    }
    export class ComplexMathTests {
        addThreeNumbers_3and5and2_10(context: tsUnit.TestContext) {
            var math = new Calculations.ComplexMath();
            var result = math.addThreeNumbers(3, 5, 2);
            context.areIdentical(10, result);
        }
    }
}
```

Test Context Assertions

The test context contains built-in assertions to shorten the code you have to write in your tests. In the following example, the code using the assertion is much shorter than the manually written code, and is very easy to understand. Using the assertion makes it less likely that you will make a subtle mistake when determining the test outcome, such as using `!=` in place of `!==`, which could give you incorrect behavior. The error message that is generated is also much better in the version that uses the `areIdentical` assertion.

```
// with
context.areIdentical(8, result);

// without
if (result !== 8) {
    throw new Error("The result was expected to be 8, but was " + result);
}
```

To get the most out of the assertions, it is worth committing the following list to memory even though TypeScript will give you autocompletion when you are typing your code. Knowing what functions are available will help you to write cleaner code by selecting the most appropriate checks in your test.

Identical

The functions for `areIdentical` and `areNotIdentical` test two arguments for both type and value. This means that a "string of 1" will not match a number of 1. To pass the `areIdentical` test, the values must be the same type and the same value for primitive types, or the same instance of an object for a complex type. The `areNotIdentical` assertion does the reverse.

True or False

The `isTrue` assertion checks that a Boolean parameter is true and `isFalse` checks that a boolean parameter is false. To check other types that are not Booleans, but can be coerced to true or false in JavaScript, you can use `isTruthy` and `isFalsey`. The values that are considered falsey or false-like are:

- null
- undefined
- false
- NaN
- 0
- `''`

Anything that is not on this list is truthy or true-like. This can catch people off guard because the string `'false'` evaluates as true-like, because it is not an empty string.

Throws

The `throws` assertion takes a function as a parameter. To pass the test, the function must raise an error when it runs. This test is useful for checking that your error handling routines are raising errors when they should. The function you pass to the `throws` assertion would normally just contain the call you expect to fail. For example, if the `addTwoNumbers` function didn't accept negative numbers, you could check that a call to the function with a negative number results in an error.

```
addTwoNumbers_NegativeNumber_Error(context: tsUnit.TestContext) {
    var math = new Calculations.SimpleMath();


    context.throws(function () {
        math.addTwoNumbers(-1, 2);
    });
}
```

If the call to `addTwoNumbers` throws an error, the test passes, otherwise it fails.

Fail

The `fail` assertion simply forces a test to fail every time. Although this sounds useless, you can use this assertion to fail a test based on your own custom logic. The `fail` assertion is sometimes used as a placeholder when writing a test to prevent an accidentally loose test. For example, if you are working on the arrangement of the test, you could place a `fail` assertion at the end of the test to ensure it fails, which reminds you the test is not yet complete. If you didn't do this and forgot to add any assertions, the test would give you false confidence.

Test Doubles

To protect your unit tests from changes in other modules, you will need to isolate the classes you test by swapping out real dependencies with test doubles. This is a common practice in .NET using the built-in Fakes assemblies, or a mock repository such as Rhino, Moq, or nMock.

TypeScript has some smart features that make it easy to supply a test double in place of a real dependency. It is worth looking at a manual test double before using a framework that will supply one for you.

```
module Example {
    export class ClassWithDependency {
        constructor (private dep: MyDependency) {
        }
```

```
        exampleFunction() {
            return this.dep.functionOnDependency(2);
        }
    }
}

module Test {
    export class ExampleTests {
        testIsolation(context: tsUnit.TestContext) {
            var dependency: MyDependency = new MyDependency();
            var target = new Example.ClassWithDependency(dependency);

            var result = target.exampleFunction();

            context.areIdentical('Fake string', result);
        }
    }
}
```

In this example, the `ClassWithDependency` class needs an instance of `MyDependency` to work, and calls `functionOnDependency` as part of `exampleFunction`. In the test I am creating an instance of the real `MyDependency`, but this is problematic because the test is no longer isolated from changes in the dependent classes. A unit test should only rely on the class it is testing; when it relies on other classes, it makes the unit test more brittle as changes to all of the referenced classes could require a change to the test. You can also get caught in long dependency chains; for example, if `MyDependency` depended on another class, you would need to create one of those too and so on.

To solve these problems, you can create a test double to use in place of the real dependency. Your test double can also supply canned answers that make your tests more predictable.

```
module Test {
    class MyTestDouble extends MyDependency {
        functionOnDependency(a: number): string {
            return 'Fake string';
        }
```

```
    }

    export class ExampleTests {
        testIsolation(context: tsUnit.TestContext) {
            var testDouble: MyDependency = new MyTestDouble();
            var target = new Example.ClassWithDependency(testDouble);

            var result = target.exampleFunction();

            context.areIdentical('Fake string', result);
        }
    }
}
```

The test double is simply a class that is only available within the test module that inherits from the class you need to substitute. The implementation of the function returns a fixed value that makes tests more predictable.

One of the downsides to this approach is that if the base class has a constructor, you will need to pass in a value that TypeScript can match to the parameter types on the constructor. You should be able to use `null` or `undefined` in most cases to break the dependency chain, as no methods on the base class will ever be called. This is one good reason not to perform any actions in your constructor, as the constructor will be called. You will also need to ensure that your test double implements any function or variable that will be called when your test runs. If you leave something out, it will be called on the base class.

If you are creating a test double based on an interface, you will need to implement all of the properties and methods, but for any that aren't being used, feel free to return `null` or `undefined` rather than wasting your time creating implementations for all functions.

Lightweight Fakes

Now that you have seen manual test doubles, you are ready to replace them with automatically generated fakes.

```
module Test {
    export class ExampleTests {
        testIsolation(context: tsUnit.TestContext) {
            var fake = new tsUnit.Fake(new MyTestDouble());
            var testDouble = <MyDependency> fake.create();
```

```
            var target = new Example.ClassWithDependency(testDouble);

            var result = target.exampleFunction();

            context.areIdentical('Fake string', result);
        }
    }
}
```

This example shows that you can create a test double using just two lines of code, no matter how complex the real class is that you need to fake. You don't need to supply any functions or variables unless you need specific behavior from the test double.

Running Tests in Visual Studio

Being able to run your TypeScript tests in the browser is all well and good, but ideally you should make them part of your normal test run. By wrapping your TypeScript tests in a .NET unit test, you can run them inside of Visual Studio and also include them in your continuous integration process. In this section, the examples have come from MSTest and MSBuild, but the same techniques will work just as well with any .NET test runner and build engine.

The TypeScript tests will be compiled to JavaScript, and the unit test will run against these JavaScript files using the Microsoft Script Engine. The tests won't be running in the context of a browser.

Unit Test Project

You can add your TypeScript tests to an existing unit test project, or create a new one. The first step is to add a reference to the unit testing project for the COM Microsoft Script Control 1.0, which will run the tests using JScript, Microsoft's ECMAScript implementation.

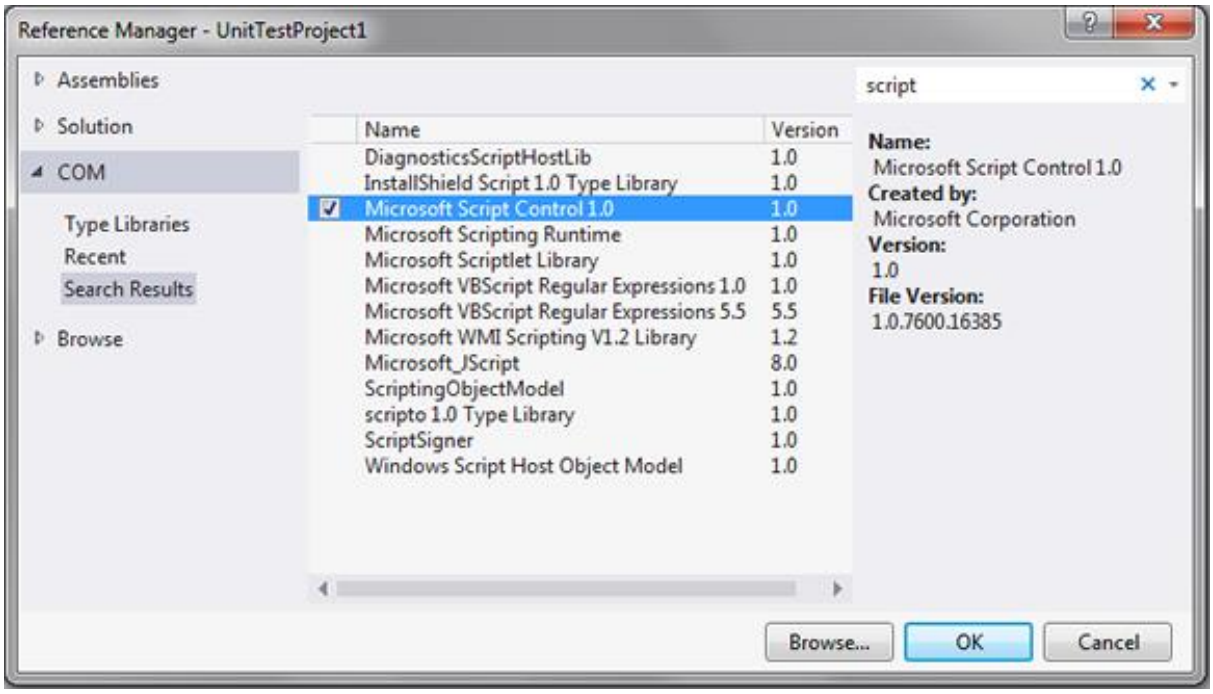

Figure 10: Reference the COM Microsoft Script Control

To make the Script Control easier to work with, you can use this adapted version of Stephen Walther's `JavaScriptTestHelper`. This class handles all of your interaction with the Script Control. For now you can add this to your unit testing project; long term, it belongs someplace that all your unit testing projects can access.

```
public sealed class JavaScriptTestHelper : IDisposable
{
    private ScriptControl _scriptControl;
    private TestContext _testContext;

    public JavaScriptTestHelper(TestContext testContext)
    {
        _testContext = testContext;
        _scriptControl = new ScriptControl()
        {
            Language = "JScript",
            AllowUI = false
        };
        LoadShim();
```

```
    }

    public void LoadFile(string path)
    {
        _scriptControl.AddCode(File.ReadAllText(path));
    }

    public void ExecuteTest(string testMethodName)
    {
        dynamic result = null;
        try
        {
            result = _scriptControl.Run(testMethodName, new object[] { });
        }
        catch
        {
            RaiseTestError();
        }
    }

    public void Dispose()
    {
        _scriptControl = null;
    }

    private void RaiseTestError()
    {
        var error = ((IScriptControl)_scriptControl).Error;
        if (error != null && _testContext != null)
        {
            _testContext.WriteLine(String.Format("{0} Line: {1} Column: {2}",
                error.Source, error.Line, error.Column));
        }
```

```
            throw new AssertFailedException(error.Description);
        }

        private void LoadShim()
        {
            _scriptControl.AddCode(@"
                var isMsScriptEngineContext = true;
                var window = window || {};
                var document = document || {};");
        }
    }
```

Once these two items have been added to the project, you are ready to put them into action. I personally prefer to run all of my TypeScript tests under a single .NET unit test. If any of the tests fail, the error is descriptive enough to tell you exactly which one failed; if you divide them into multiple .NET unit tests, you create extra work mapping new TypeScript tests to the right .NET tests.

Referencing the Script Files

While it is possible to use a relative path to traverse out of your test project and into the project containing your TypeScript files, doing so can be brittle when it comes to running the tests as part of a continuous integration build. A better mechanism is to add the compiled JavaScript files as linked files in your test project and set them to copy always.

1. Add a new folder to your unit test project called **ReferenceScripts**.
2. Right-click on the folder and select **Add**, and then click **Existing Item**.
3. Change the file-type filter to **All Files**.
4. Browse to the folder containing your TypeScript and JavaScript files.
5. Select the compiled JavaScript files for tsUnit, the modules you are testing, and the test modules that exercise them.
6. Open the **Add** menu, and select **Add As Link**.

The JavaScript files will now be listed in the **ReferenceScripts** folder. Select all of them and change their properties to set **Copy To Output Directory** to **Copy always**.

This will cause the script files to be placed in the bin directory of the test project when it is built. When you are running under a build system, this is required because it may build your projects in a special directory structure, and relative paths would be broken.

Adjusting the TypeScript Test

When your tests run under the Microsoft Script Control, there is no browser window or document, so rather than using the TypeScript HTML output, you will want to capture the result and use it to pass or fail the test.

You don't need to rewrite your tests to make them work, because the `JavaScriptTestHelper` contains a shim that notifies the script that it is running in a browser-less context. Earlier in the book, the last few lines of the test file called tsUnit like this:

```
// Create an instance of the test runner.
var test = new tsUnit.Test();
// Register the test class.
CalculcationsTests.Composer.compose(test);
// Run the tests and show the results on the webpage.
test.showResults(document.getElementById('result'), test.run());
```

To make this work with both the tsUnit runner and the Microsoft Script Control runner, you can change this code to switch based on the `isMsScriptEngineContext` flag passed by the `JavaScriptTestHelper`.

```
class TestRunner {
    private test: tsUnit.Test;
    constructor() {
        this.test = new tsUnit.Test();
        CalculcationsTests.Composer.compose(this.test);
    }
    runInBrowser() {
        this.test.showResults(document.getElementById('results'),
this.test.run());
    }
    runInScriptEngine() {
        var result = this.test.run();
        if (result.errors.length > 0) {
            var message = '';
            for (var i = 0; i < result.errors.length; i++) {
                var err = result.errors[i];
                message += err.testName + ' ' +
```

```
                    err.funcName + ' ' +
                    err.message + '\r\n';
            }
            throw new Error(message);
        }
    }
}

declare var isMsScriptEngineContext: boolean;

var testRunner = new TestRunner();
if (!isMsScriptEngineContext) {
    testRunner.runInBrowser();
}

function getResult() {
    testRunner.runInScriptEngine();
}
```

When you run this in a browser, it will display the results as before, but when the .NET unit test runs, the output will be controlled by the code in the `runInScriptEngine` function. If a test fails, it is this function that will forward the complete error message to the .NET test.

Creating the Unit Test

The final piece of the puzzle is the .NET test method.

```
[TestClass]
public class TypeScriptTests
{
    private TestContext _context;

    public TestContext TestContext
    {
        get { return this._context; }
```

```
        set { this._context = value; }
    }

    [TestMethod]
    public void RunTypeScriptTests()
    {
        var runner = new JavaScriptTestHelper(_context);

        // Load JavaScript files.
        runner.LoadFile("ReferenceScripts\\tsUnit.js");
        runner.LoadFile("ReferenceScripts\\Calculations.js");
        runner.LoadFile("ReferenceScripts\\CalculationsTests.js");

        // Execute JavaScript test.
        runner.ExecuteTest("getResult");
    }
}
```

When you run your tests, the `TypeScriptTests` test class will be indistinguishable from your other tests. It will report back any test failures with full details if there is a problem, and there will be a big green tick if the tests pass.

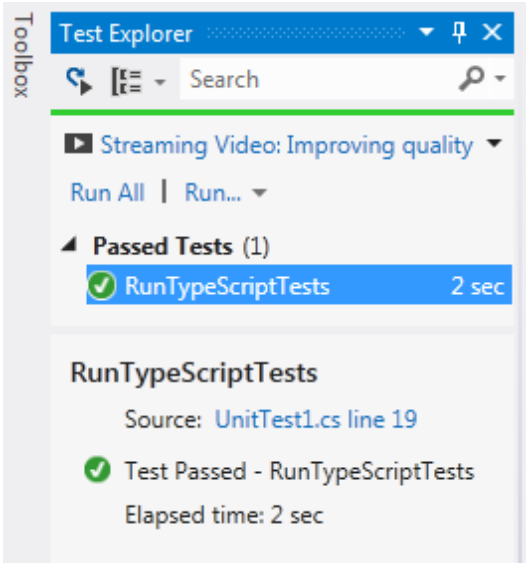

Figure 11: Test Explorer

When your continuous integration build runs, it will be able to run these tests just like any other test, and you should have similar results using any of the popular unit testing frameworks for .NET.

Testing with a JavaScript Framework

You can use any of the existing JavaScript unit testing frameworks to write tests in TypeScript. You may need to find or write ambient declarations to get the type checking and compilation warnings that you get for free with tsUnit. You may also find that there are some naming clashes caused by some of the unit testing frameworks adding too much to the global scope; for example, qUnit has a function named module in the global scope, which clashes with the reserved word in TypeScript. I have listed links to some JavaScript unit testing frameworks in *Appendix C: External Resources*.

Summary

Key Points

TypeScript is fundamentally a design-time productivity tool and a compile-time check that makes JavaScript development more robust. It helps your development tools to be smarter and to provide tooling support that wouldn't be possible without the structure and types annotations. In Visual Studio, as well as type checking and autocompletion, it is possible to use the refactoring tools with TypeScript.

Many of the language features provided by TypeScript are removed when generating the JavaScript output, so if you write a program that is consumed in its compiled JavaScript form, the consumers of your library won't benefit from these features.

TypeScript is not meant to be C# or VB.NET, so the syntax is different even for features you may have already been very familiar with. Although it is tempting to see these differences as flaws, it is worth noting that different doesn't mean worse. For example, the keywords for inheritance and interface implementation in TypeScript are far more explicit than the colon syntax in C#.

Futures

I have already mentioned that there is experimental support for enumerations in TypeScript, although it is highly likely to be changed in the next version. There is also planned support for generics, which were the most popular addition to .NET 2. The actual implementation for generics is currently unknown, so I haven't been able to preview this feature in the book like I did with enumerations.

There is also an active community discussing how people are using TypeScript, which highlights the need for subtle changes and new language features. The project is designed to align to the ECMAScript 6 standards and compile to raw JavaScript, which may constrain some of the more off-the-wall ideas that are suggested.

I hope this book has been a useful dive into TypeScript as a serious programming language. I'm certain that TypeScript has an important part to play in making code more maintainable, which will reduce the cost of change. Ultimately, it will make it cheaper and faster to add features to programs that need scripting in the browser, or that run on a JavaScript virtual machine on a server processing large numbers of requests on a tiny number of threads.

Appendix A: Alternative Development Tools

You can use any text editor to write TypeScript files, although you won't benefit from the design-time checking.

Free Editors

Notepad++: http://notepad-plus-plus.org/

Emacs: http://www.gnu.org/software/emacs/

VIM: http://www.vim.org/

Commercial Editors

Sublime Text: http://www.sublimetext.com/

Cloud9 IDE: https://c9.io/

TypeScript Plug-ins

Plug-ins for some of these editors are already available for TypeScript. I expect the number of editors and development environments that support TypeScript to increase rapidly.

http://aka.ms/qwe1qu

Appendix B: TypeScript Command Line

To compile TypeScript into JavaScript manually using the command line, use the `tsc` command. All of the following examples have been run in the context of the folder containing the TypeScript files.

Basic TypeScript Command Line

```
>tsc Main.ts
```

The basic `tsc` command takes the specified file and generates a compiled JavaScript file with the same name, but with a .js file extension.

TypeScript CLI with Modules

```
>tsc Main.ts --module commonjs
```

You can specify the module using the `--module` flag, which accepts either `commonjs` or `amd` values.

TypeScript CLI ECMAScript Version

```
>tsc Main.ts --target ES5
```

You can specify the ECMAScript version using the `--target` flag. The default is `ES3`, but you can override with `ES5`.

TypeScript Generate Ambient Declaration

```
>tsc Main.ts --declarations
```

You can generate an ambient declaration file with a .d.ts file extension by running `tsc` against a TypeScript file with the `--declarations` flag.

TypeScript Setting Output

```
>tsc Main.ts --out Name.js
```

You can change the name of the generated JavaScript file using the `--out` flag and passing an alternate file name. I don't recommend you use this option, as it loses the association between the source file and the compiled output.

You can mix and match any of these commands to customize your compilation.

Appendix C: External Resources

TypeScript

Official TypeScript website

http://www.typescriptlang.org/

http://typescript.codeplex.com/

Loading Modules

CommonJS: http://www.commonjs.org/

RequireJS: http://requirejs.org/

Unit Testing

tsUnit: http://tsunit.codeplex.com/

Jasmine: http://pivotal.github.com/jasmine/

qUnit: http://qunitjs.com/

YUI Test: http://developer.yahoo.com/yui/yuitest/