THE EXPERT'S VOICE® IN .NET

TypeScript Revealed

DISCOVER ANDERS HEJLSBERG'S NEW WAY TO MAKE JAVASCRIPT SCALABLE AND EASIER

Dan Maharry

Apress®

For your convenience Apress has placed some of the front matter material after the index. Please use the Bookmarks and Contents at a Glance links to access them.

Apress®

Contents at a Glance

Introduction

Anders Hejlsberg has a lot to answer for. Perhaps not as well known as Bjarne Stroustrup (C++) or Dennis Ritchie (C), he's responsible for the creation of Turbo Pascal, Delphi, J++, and C#. And, as the creator of the language that the majority of .NET developers have been using for the past ten years, it is fair to say that he knows a fair bit about language design.

October 1, 2012 heralded the start of the journey for another Hejlsberg creation. TypeScript is a superset of JavaScript that brings to it an additional object-oriented-like syntax familiar to .NET programmers that compiles down into JavaScript that any browser can run today. As I write this three weeks since its announcement, it's interesting to see how, with hindsight, the loudest reactions have been, "Why do we need this?" or "Not in my backyard," and so on—the naysayers getting their dimes' worth before this Friday, when Windows 8 is officially launched and they can copy and paste the same reactions back into their slightly revised pieces about the new touch interface that isn't called Metro. Now that the dust has settled, my personal favorite response to all the initial negativity was simply, "But this is Anders."

TypeScript is an interesting project, considering the forces that will work at it and the target at which it is aimed.

- Its mission statement is "JavaScript for application-scale development."
- It's an open source project, so there's direct input from its users.
- It's a standard in its own right.
- It's a proof of concept for, and therefore will/should react to, parts of the ECMAScript 6 standard.

It's inevitably a well-timed boost for the development of Windows 8 applications using HTML and JavaScript rather than C# and XAML, or a hybrid approach between the two for lack of JavaScript structure.

How TypeScript is allowed to evolve will be interesting. In the meantime, we have the initial v0.8.x releases that this book targets. It covers the following:

- The language features in TypeScript additional to those in JavaScript
- The tooling and community support for TypeScript development
- How TypeScript can be included in ASP.NET, and Windows 8 application development projects

Who Is This Book For?

This book is for .NET and JavaScript developers who want to get a head start incorporating TypeScript into their applications. I explain the tools and language features you need to get up to speed quickly and also the current, soon to be ex-limitations of TypeScript as it makes its way toward a version 1.0 release.

What Do You Need to Know Before You Read This Book?

You need a good knowledge of an object-oriented language such as C#, ideally from creating ASP.NET web applications or Windows 8 applications. TypeScript tooling is, in the main, based on Visual Studio, so a familiarity with that would also be good.

You also need a basic familiarity with JavaScript, but advanced JavaScript skills are not necessary.

What Software Do You Need for This Book?

Technically, you don't need anything more than a text editor such as Notepad to write TypeScript applications. However, to take full advantage of the tooling that exists, you'll need a copy of a non-Express version of Visual Studio 2012. If you don't have access to that, there is support for Visual Studio 2012 Express for Web and also for Sublime Text 2, Vim, and eMacs, which are all free downloads.

You'll learn where to download and how to install TypeScript in Chapter 1.

What Is the Structure of the Book?

This is a primer to get you started on TypeScript. As TypeScript is at such an early stage in its adoption and development, this book does not aim to be a comprehensive tutorial and focuses instead on the three key topics that you'll need to consider in order to determine whether you'll want to start using TypeScript now, and if so, how to do it:

- Chapter 1: Getting Up to Speed with TypeScript

  Just exactly what is TypeScript, and how does it fit in to your projects? In this chapter, you'll look at the aims of TypeScript, how it compares to its perceived competition, and what exactly it does and does not do. You'll also see how to install it, what you get, and how to run through creating a simple TypeScript-based page.

- Chapter 2: The New Syntax

  TypeScript is all about the additional object-oriented-like syntax and features that it brings to JavaScript. In this chapter, you'll look at the main new constructs in the language: static typing, classes, interfaces, arrow functions, and modules. You'll see the similarities between them and their equivalents in C# as well as the common gotchas encountered so far.

- Chapter 3: Working with TypeScript

  With the new syntax covered, the final chapter takes a look at how to incorporate TypeScript into your project. You'll look at the various options on the TypeScript compiler, learn how to create declaration files for third-party JavaScript libraries, and see how to include TypeScript in your existing Visual Studio projects.

- Appendix: Resources

  Finally, the appendix at the back lists all the code projects and the online communities that currently exist as well as the URLs mentioned in the book for a quick reference.

CHAPTER 1

Getting Up to Speed with TypeScript

In this chapter, you'll start your look at TypeScript with an overview and an example. In particular, this chapter covers the following:

- The motivation for the TypeScript project
- What TypeScript does and what it doesn't do compared to JavaScript and other JavaScript-related languages
- How to install it and what you get as a result
- A simple example using TypeScript and Visual Studio

Hopefully, by the end of this chapter, you'll have a good idea of what TypeScript does and how. You'll follow this up in Chapter 2 by looking in depth at the many new language features of TypeScript, and in Chapter 3 by learning how to incorporate TypeScript into some projects as a whole.

The Development of TypeScript

> *JavaScript probably should not have been the first choice for the language of all Web functionality—at least, not without some serious reworking. It became standardized long before it was ever rationalized. And had rationality been the goal, it should have looked much more like Java than script.*
>
> Scott M. Fulton, ReadWrite
>
> http://readwrite.com/2012/10/03/microsofts-typescript-fills-a-long-standing-void-in-javascript

Stop. Before you read any further, open a browser, load the home page for Google or Bing, and then view the source code for the page. It's just a mass of minimized JavaScript. Would you like to figure out how the page works by looking at the source code? No. Me neither.

JavaScript is an unusual beast. When it was first introduced in late 1995 to give a web page some dynamism, neither the browsers nor the browser vendors were ready. It arrived during the first skirmishes of the browser wars. Each browser had its own Document Object Model and its own Browser Object Model, and they were still changing. Trying to keep even some basic functionality on a web page compatible across browsers was a nightmare and required far more code than it should have. Microsoft first tried to get us to script pages by using VBScript and then caused confusion with JScript, its own version of the language. Oh, and all the JavaScript execution engines were pretty woeful. Even the name of the standard for JavaScript had to be changed to ECMAScript because Sun Microsystems wouldn't let the standards body use the name.

A little messy, wasn't it?

Fortunately, common sense has prevailed over time. Browsers now generally follow web standards, their Document Object Models are nearly all implemented in the same way, and modern JavaScript execution engines such as V8 in Google Chrome and Chakra in Microsoft Internet Explorer are exponentially quicker than their predecessors. JavaScript itself has come a long way as well. Five editions of ECMAScript have been published, and a sixth, code-named Harmony, is under development. The popularization of object-oriented JavaScript, JSON, AJAX, and the HTML5 JavaScript APIs (in that order) have seen its use in serious, nontrivial coding on the client side grow sharply. On the server side, meanwhile, Node.js has shown to the world that JavaScript is a perfectly valid server-side technology, and even Microsoft has been pushing the use of JavaScript as an equivalent to C# and Visual Basic; Windows 8 applications can be built with JavaScript and HTML5 as well as C#/Visual Basic and XAML.

Which brings us to the big question: is JavaScript "big enough" for writing complete applications? JavaScript was never designed to be a language for larger applications. Google has proven it possible, but the fact that it has also created a whole new language—Dart—which can compile down into JavaScript because that's what we all use but which requires its own virtual machine (currently available only in a developmental build of Chrome) for optimal performance, might indicate just how hard it is and how awkward maintenance can be. JavaScript does support classical inheritance and other code reuse patterns, but it's a dynamic language without the features we see in IronPython and IronRuby. It lacks the familiar home comforts of a C# or Visual Basic class-based object-oriented programming model, not to mention strong typing, interfaces, and namespaces, which we might consider the bread and butter of our development efforts. The forthcoming Harmony specification does include these, but ratification is a way off, and it'll be longer still before browsers include JavaScript engines that support it. There are still engines supporting only v3 of ECMAScript rather than the latest (published in December 2009) v5 in full, for heaven's sake.

Microsoft has decided that the big answer to the big question is no, and thus has created TypeScript, a superset of JavaScript that includes static typing, classes, interfaces, modules, and a couple of other features that .NET developers will find familiar and that are borrowed from the Harmony specification. We now have at our disposal a way to build large applications in JavaScript (or to switch into application-scale JavaScript development, as Microsoft call it) far more easily than was previously possible.

More important, we have a system that makes it easier to scale, debug, and revise applications once written. And, at the heart of it, are four simple truths:

- TypeScript is just JavaScript. You only need to know JavaScript to use TypeScript, and all your favorite third-party JavaScript libraries (jQuery, MooTools, Prototype, and so forth) will work fine with it.
- JavaScript is just TypeScript. Any valid `.js` file can be renamed `.ts` and be compiled with other TypeScript files.
- TypeScript borrows from the Harmony specification. When Harmony is ratified, the TypeScript compiler can just let the new language elements alone and allow the new Harmony-compliant engines to parse your TypeScript as pure JavaScript instead.
- The TypeScript compiler is itself a JavaScript file (compiled down from TypeScript). Thus it can be hosted in any browser, on any host, on any operating system if required. There's no dedicated VM and no plan to develop one.

Oh, and TypeScript is an open source project too:

- The compiler is an open source project and released under the Apache 2.0 license.
- The TypeScript language is available under the Open Web Foundation's Final Specification Agreement (OWFa 1.0).

All of this means that you can have direct input into bug fixing and new features by submitting bug reports and feature requests on `http://typescript.codeplex.com`.

What TypeScript Is, Does, and Does Not Do

So yay for Microsoft having the gumption to create and propel TypeScript into a crowded web development world where several efforts to make app-scale development more manageable (Dart and CoffeeScript, to name two) already exist. But now we're past the justification for its existence, what exactly does it do, and how does it work? More important, perhaps: what doesn't it do, exactly?

At its heart, TypeScript has three components:

The TypeScript compiler: Written itself in TypeScript, the compiler recompiles TypeScript down to idiomatic JavaScript, compliant to ECMAScript v3 by default or v5 with a compiler switch. The compiler is covered in Chapter 3.

The TypeScript Language Service (TLS): The TLS is a Visual Studio extension. It is this extension that provides you with IntelliSense for TypeScript, enforcing type safety and several features of TypeScript within Visual Studio.

Declaration files: The TLS uses declaration files (`.d.ts` files) to provide IntelliSense for the types, function calls, and variables expected by third-party libraries such as jQuery , Prototype, MooTools, and so forth.

You'll find templates for Visual Studio, sample projects, color-coding rules, and an MSBuild rule as well, but they're more about enriching the TypeScript development experience rather than enabling it. Understandably, as Microsoft wrote it, you'll need Visual Studio for the best TypeScript development experience. The TLS is the key. It's no less valid to write TypeScript outside of Visual Studio, but it is more likely to be prone to errors as the TLS is not there to help you along, writing your code before it goes to the compiler.

WHAT TYPESCRIPT IS NOT

The important thing to remember is that TypeScript is just a tool to be used at coding-time to improve the quality of your JavaScript. After your TypeScript is compiled to JavaScript, you can use all your other tools—minifiers, packagers, runtime loaders, unit test frameworks, and so forth—as you would if you had written the JavaScript from scratch. Microsoft provides some basic support for editing TypeScript from Sublime Text 2, Vim and eMacs, but it does not include the TLS, so there is no IntelliSense or feature enforcement. On the third-party front, JetBrains has announced TypeScript support in WebStorm v6, making this the first cross-platform IDE to do so. There are also several preliminary discussions about supporting it in MonoDevelop.

■ **Note** Please don't forget that TypeScript is still only a prerelease (v0.8.1.1 at the time of this writing). The level of TypeScript support for non–Visual Studio IDEs will change.

The TypeScript compiler is also available as a Node.js package, but it is not exposed as an explicit public API for module access. There is a workaround, however (https://github.com/eknkc/typescript-require) while Microsoft works on the official fix (http://typescript.codeplex.com/workitem/97).

TypeScript vs. JavaScript Gotchas

TypeScript's aims are squarely pointed at the long-term issues of scalability and maintainability, but does it assist with the day-to-day task of writing functional code? JavaScript has more idiomatic programming pain points than most languages, and many online pages are dedicated to listing these gotchas, so you don't have to figure them out yourself. Because it is just JavaScript itself, compiled TypeScript is subject to these gotchas as well. However, thanks to the real-time type checking and inference, the TLS

does catch a few of these before they get compiled. This section details three of the most common that TypeScript can help to avoid.

Type Coercion Issues

Suppose you have two variables, one a string and the other a number:

```
var oneNumber = 1;
var oneString = "1";
```

In JavaScript, you can compare these two variables by using the == operator and expect JavaScript to return `true`. The == operator coerces one variable into the same type as the other and then compares the two:

```
var theSameJS = (oneNumber == oneString);  // true in JavaScript. Invalid
in TypeScript
```

In TypeScript, the same operation will not compile until one of the two variables is explicitly cast to the other's type. Even if the types of `oneNumber` and `oneString` are not specifically set, the TLS will infer them from the values assigned, meaning this gotcha is one you shouldn't need to worry about.

```
var theSameTS = (oneNumber.toString() == oneString);  // true in TypeScript
and JavaScript.
```

A similar gotcha occurs when using the + operator. If one of the operands is a string, JavaScript will always concatenate the two values together and return a string, rather than adding the two as numbers and returning a number, which may have been the desired result.

```
var thisIsAString = 1 + "1";  // returns "11"
var thisIsANumber = 1 + parseInt("1", 10);  // returns 2
```

The TLS won't stop you from making the same mistake, but it will infer the type of the returned value and flag potential errors later if you reuse it. If you hover the cursor over the name of the variable, the TLSwill display the inferred type of the variable as well.

Block Scope

TypeScript inherits JavaScript's lack of block scoping for variables. Consider the following (contrived) code:

```
for (var i = 0; i < 10; i++) {
    console.log(i);  // displays numbers 0 to 9
}
```

```
//  variable i now redeclared but value not reset to undefined.
// instead it uses last set value of 9 from inside the for loop block
var i;
console.log(i);  // displays 10
```

When `i` is declared, it will remain in scope after the loop exits, so the console will display 10 as well as 0–9. The TLS won't stop this from happening, but it will infer that `i` is of type `number` or type `any` when it is redeclared because of its previous use as a number in the `for` loop. Therefore, should `i` be assigned or inferred a type other than `number` or `any` outside of the `for` loop, the code as a whole will not compile because the code inside the `for` loop—specifically, the part where `i` is treated as a number—is then invalid.

Optional Semicolons and Bracket Placement

For C# programmers, the idea that you need not use semicolons to indicate the end of a statement may seem anathema. Likewise, if you press Ctrl+K, D enough times in Visual Studio, it may start to seem odd if the curly braces in your code aren't on their own lines. So consider this particular code and ask yourself what the function returns:

```
function optionalSemicolons() {
    return
    {
        a: "Hello"
    };
}
```

In fact, this function returns `undefined` because JavaScript interprets the code by adding a semicolon immediately after the `return` statement. If the opening curly brace is placed next to the `return` keyword, the function returns an object as expected.

```
function optionalSemicolons2() {
    return {
        a: "Hello"
    };
}
```

All of this means that both C# and Visual Basic programmers need to take care when adding semicolons and new lines into code. Fortunately, the TLS can save us some head-scratching in two ways here.

The TLS, together with the Web Essentials extension for Visual Studio, will display your TypeScript code recompiled into JavaScript in the right-hand pane of your Visual Studio window whenever you save your TypeScript file. The compiled version of `optionalSemicolons()` shows the inserted semicolons (there are actually two of them) not where you'd want them to be.

```
function optionalSemicolons() {
    return;
 {
        a:
"Hello"
    }
    ; ;
}
```

The compiled version of `optionalSemicolons2()`, however, is exactly the same as the TypeScript original.

- You can also explicitly define the return type of the function. In this case, the function returns an object, so we can rewrite the first line of our errant function as

```
function optionalSemicolons() : Object { ...
```

and the TLS will instantly underline the whole function with red squigglies as it does not return an object. Adding the same explicit return type to `optionalSemicolons2()` will result in no change.

■ **Note** Full details of TypeScript's strong typing and function signature syntax can be found in Chapter 2.

TypeScript vs. Other Compile-to-JavaScript Languages

So TypeScript does help with the quality of the JavaScript you can produce as well as its overall structure, but there's plenty of competition in the particular space. Several dozen similar language projects are listed at https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS, for example. How does TypeScript stack up against them all, and is it right for you? Let's start by asking a few questions:

- What's the main aim of the language?
- Is it still JavaScript, or something else?
- Does it add static typing?
- Does it add a class-based, object-oriented programming model?
- Can I use third-party JavaScript libraries?
- Is it supported in Visual Studio or Windows?
- Is it supported on Mac OS or Linux?
- What VM does it use to run?

First we'll answer for TypeScript and then for the two most visible alternatives to TypeScript at the moment: Dart and CoffeeScript.

- The main aim of TypeScript is to enable large-scale application development with JavaScript.
- TypeScript is a superset of JavaScript, so yes, it is still JavaScript.
- Yes, TypeScript incorporates static typing.
- Yes, TypeScript adds a class-based, object-oriented programming model.
- Yes, but you'll need a declaration file for the library to enable IntelliSense and strong typing.
- Yes, with the TypeScript Language Service extension.
- No concrete IDE support yet for Mac OS or Linux, although the compiler itself is easy to install by itself or in Node.js.
- It uses the JavaScript VM.

Dart (www.dartlang.org) is Google's answer to JavaScript. This completely new language addresses JavaScript's issues, cleans up its semantics, adds static types (as does TypeScript) and class-based object-oriented features, and improves startup times and performance overall. It requires its own Dart VM (currently available only in a developmental build of Chrome) for optimal performance but does come with a Dart-to-JavaScript compiler. It shares some common goals with TypeScript—the main one is making large-scale web application development possible—but in general, the two are separate, with only the dart2js compiler linking the two languages.

- The main aim of Dart is to learn from the mistakes of JavaScript and provide an alternative.
- No, it's a new language but it compiles to JavaScript.
- Yes, Dart incorporates static typing.
- Yes, Dart adds a class-based, object-oriented programing model.
- Not yet, although there is a js-interop library.
- The Dart Editor IDE is available for Windows, Mac OS, and Linux.
- Dart has its own VM, but when you compile it to JavaScript, you can run it in any JavaScript VM.

CoffeeScript (http://coffeescript.org), meanwhile, tries not to replace JavaScript but to produce a syntax based on other dynamic languages. This makes CoffeeScript easier and quicker to write, as it then compiles into neat, well-formed JavaScript. Essentially, you could say it's JavaScript without the large numbers of braces, brackets, and parentheses, but that would be missing the point. It sits in your tool chain at the same point as TypeScript but helps you generate better JavaScript while using a non-JavaScript syntax.

- The main aim of CoffeeScript is to provide a language that has neater syntax than JavaScript (based on Ruby and Python) but that compiles back into JavaScript.
- No, it's a different language, but like TypeScript, it's a design-time-only construct to help you write better JavaScript.
- No, CoffeeScript does not add static typing.
- Yes, CoffeeScript adds a class-based, object-oriented programming model.
- You can use external JavaScript libraries from inside CoffeeScript, but you'll get syntax errors if you compile JavaScript as is, without converting it.
- CoffeeScript is supported via the Web Extensions add-in for Visual Studio. There are also several other Windows IDEs that support CoffeeScript development.
- CoffeeScript is available as a Node.js package and via APT for Linux and Mac. Eclipse, Vim, TextMate, and many other IDEs support CoffeeScript.
- CoffeeScript compiles down to JavaScript, so all you need is a JavaScript VM.

TypeScript vs. ECMAScript

The final question in this section is how TypeScript may evolve as we continue to use it. It is an open source project, so we know that Microsoft will be listening to feature requests as well as taking on bug patches, but we also know that it is trying hard to keep its "additional features" aligned with the ECMAScript 6 standard. For example, the class, module, and arrow function syntax conform to this proposed future standard.

Going one further, it has been stated that,

> *In the long term, we expect TypeScript to include everything that becomes part of future versions of ECMAScript. In some cases, these will only be available when targeting newer browsers, when the feature relies upon some new runtime capability. But in many cases, where the features are just syntactic sugar, we expect to be able to compile to simple and clean JavaScript for ES3/ES5, which implements a very good approximation.*
>
> http://TypeScript.codeplex.com/workitem/15

Looking at the spec as it stands (http://wiki.ecmascript.org/doku.php?id=&idx=harmony), there's quite a lot to keep the project busy for several major version releases. What's not clear, however, is how Microsoft will react if ECMAScript 6 is altered such that realigning TypeScript with the new syntax would create breaking changes across all existing TypeScript projects. Time will tell.

Installation

Now that we've established what TypeScript is, what it does, and what it doesn't do, let's get cracking and install it. TypeScript has a couple of homes on the Internet:

- `typescriptlang.org` is its official home. Here you'll find the latest bits to download, tutorials, and links to the main help forums.
- `typescript.codeplex.org` is where you'll find the latest source code for the TypeScript compiler, along with a list of open bugs and more discussion forums.

You'll be concentrating on setting up a TypeScript development environment here rather than working with nightly builds of the compiler. There are two environments you'll see how to build:

- Visual Studio 2012
- Sublime Text 2

■ **Note** You'll also find links on `typescriptlang.org` to syntax highlighter files for Vim and eMacs.

Visual Studio 2012

Microsoft has focused on giving TypeScript developers the best experience through the non-Express editions of their Visual Studio IDE. In this section, you'll see how to integrate TypeScript as best as currently allows for work in Visual Studio Professional, Premium, or Ultimate. You'll find notes about using TypeScript with Visual Studio 2012 Express editions and WebMatrix afterward.

Installing the Compiler and TLS Extension

Assuming you already have Visual Studio 2012 installed, the first task is to install the TypeScript plug-in:

1. Open a browser at `www.typescriptlang.org/#Download`.
2. When it loads, click the link to Download the Plugin for Visual Studio 2012. Currently this points to `www.microsoft.com/en-gb/download/details.aspx?id=34790`.
3. Download the installer file `TypeScriptSetup.<VERSION>.msi` and double-click to install it.

When the installer has finished, open Windows Explorer. Then do one of the following:

- If you're running 64-bit Windows, browse to `C:\Program Files (x86)\Microsoft SDKs\TypeScript\<VERSION>` to find the installed files.
- If you're running 32-bit Windows, browse to `C:\Program Files\Microsoft SDKs\TypeScript\<VERSION>`.

Figure 1-1 shows the contents of the directory.

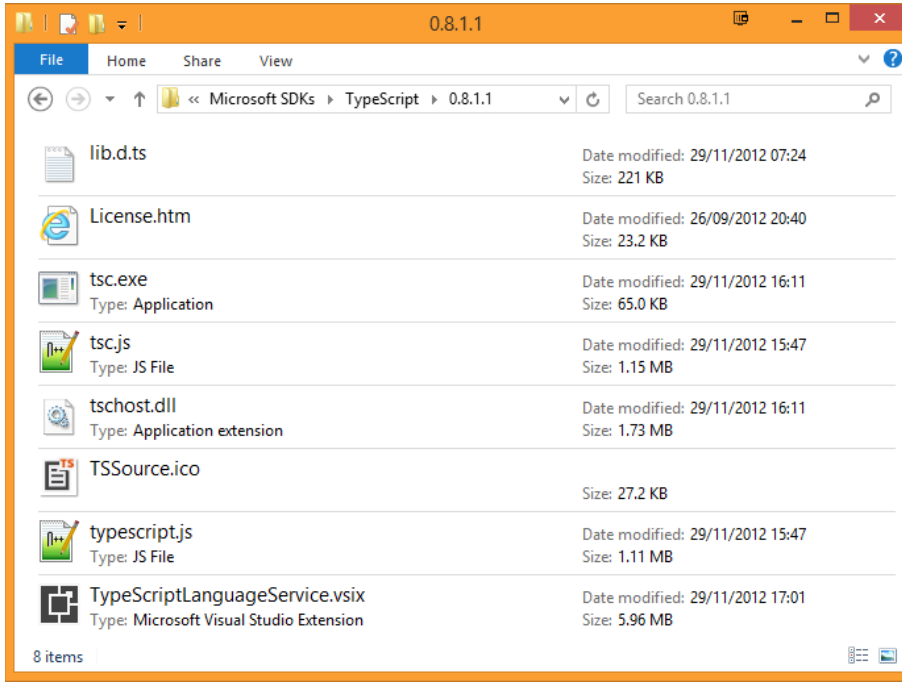

Figure 1-1. *The TypeScript installation directory and its contents*

The key files are as follows:

- `tsc.exe` is the TypeScript compiler as an executable file.
- `tsc.js` is the command-line compiler called by `tsc.exe`. It contains an options parser and a batch compiler needed for compiling multiple files.
- `typescript.js` is the core compiler without the options parser and batch compiler.
- `tschost.dll` is the scripting host for the compiler.

- `TypeScriptLanguageService.vsix` is the installer file for the Visual Studio TLS extension.
- `lib.d.ts` is a TypeScript declarations file containing the type descriptions for the entire Document Object Model and JavaScript standard runtime. The TLS extension uses this file to provide IntelliSense and type checking when coding.

■ **Note** See Chapter 3 for the various command-line options available for the TypeScript compiler and for more on declarations files.

At this point, you may want to do the following:

- Check that the installation directory has been added to your `PATH` environment variable so you can easily run `tsc.exe` from the shell of your choice.
- Download some more definitions files for your future use. If you are likely to use some third-party JavaScript libraries in your application (such as jQuery, WinJS, or Node.js), you'll need these for the TLS extension to provide IntelliSense and type checking in your calls to these libraries.

  You can find `d.ts` files for WinJS, WinRT, and jQuery 1.7 on the TypeScript CodePlex web site. You'll find them in the source code listings in the typings directory, and also at `https://github.com/borisyankov/DefinitelyTyped`, along with many others.

The TLS extension adds several new templates and a new build step into Visual Studio:

- You can find the HTML Application with TypeScript project template listed under Visual C# (not Visual C# ➤ Web) in the New Project dialog box, as shown in Figure 1-2.

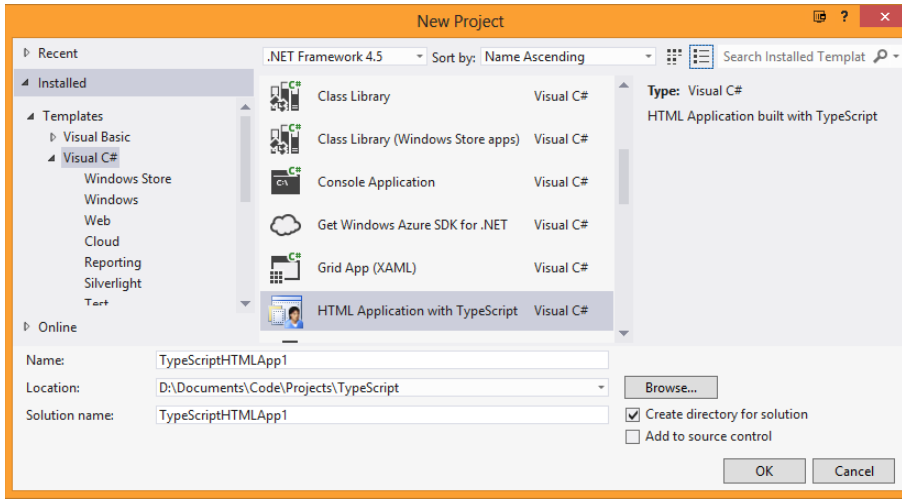

Figure 1-2. *Locating the TypeScript project template*

- To add a TypeScript file to an existing project, use the TypeScript File file template listed under Visual C# (not Visual C# ➤ Web) in the New File dialog box, as shown in Figure 1-3.

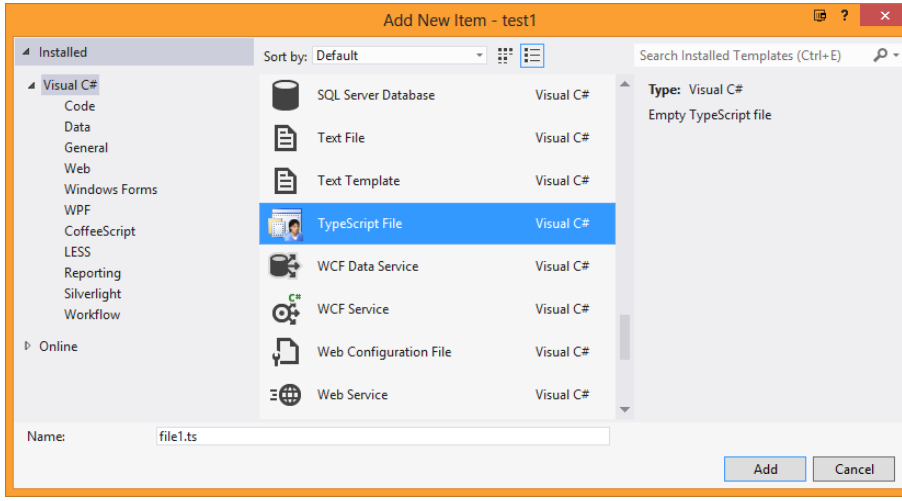

Figure 1-3. *Locating the TypeScript File file template*

When you add a new TypeScript file to your project, Visual Studio will open it automatically and present some sample code, which you'll need to delete.

■ **Tip** If you can't find the templates in Visual Studio, try rerunning the `TypeScriptLanguageService.vsix` file manually.

The TSL extension also adds a few editing options if you'd like to tweak the way Visual Studio displays your code. From the Tools menu, choose Options. When the Options dialog box appears, expand Text Editor and then TypeScript, as shown in Figure 1-4.

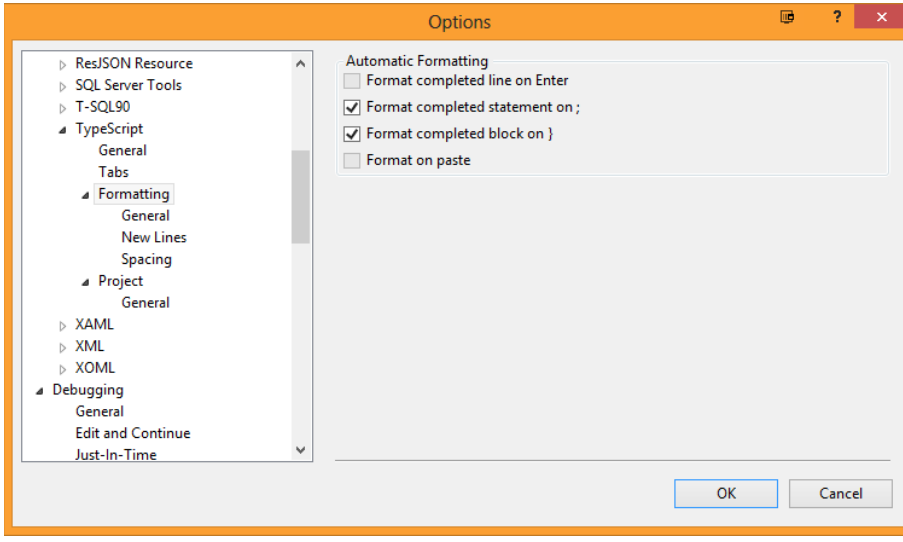

Figure 1-4. *The TypeScript Options dialog box in Visual Studio 2012*

Using Web Essentials

An optional but very handy extension for Visual Studio 2012 only is Mads Kristensen's Web Essentials 2012. His version for VS 2010 does not include TypeScript support.

■ **Note** Mads Kristensen is the creator of BlogEngine.NET, among other things, and a program manager for the Web Platform and Tools team at Microsoft. You'll find his blog at `http://madskristensen.net`.

Web Essentials adds three very useful functions to your TypeScript development process:

- When you open a TypeScript file, Web Essentials presents it in a split-screen editing panel, as shown in Figure 1-5. The left side contains the TypeScript you are writing, and the right-hand panel automatically updates with the JavaScript that will be generated. By default, this is ECMAScript 5–compliant JavaScript. If you want to compile it as ECMAScript 3–compliant JavaScript, you'll need to tweak the `TypeScriptCompile` build action for MSBuild, which is discussed in Chapter 3. Web Essentials will automatically compile the TypeScript file and update the preview as soon as it is saved in Visual Studio.

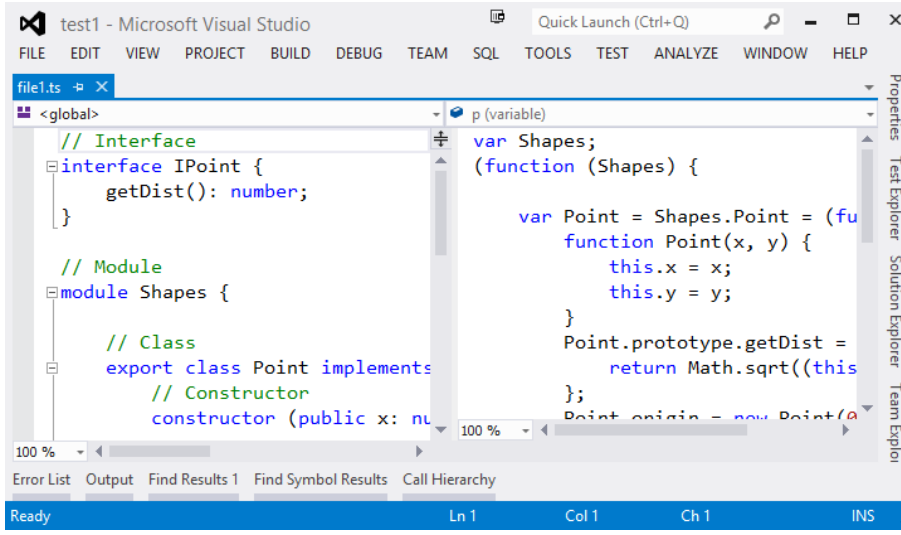

Figure 1-5. *The split-pane view of a TypeScript file in Visual Studio 2012 with the Web Essentials extension installed*

- Optionally, you can also set Web Essentials to produce a minified version of the generated JavaScript file.
- Web Essentials will automatically produce source map (`.js.map`) files for debugging purposes as soon as the `.ts` file is saved in Visual Studio.

In fact, it's a regular Swiss army knife for web developers, with features for style sheet, JavaScript, CoffeeScript, LESS, and JSON development.

Assuming you already have Visual Studio 2012 and the TLS extension installed, you can install Web Essentials 2012 in one of two ways:

- Using the Extensions and Updates dialog box in Visual Studio itself:
  a. From the Tools menu, choose Extensions and Updates.
  b. When the dialog box appears, choose Online ➤ Visual Studio Gallery and press Ctrl+E to move to the search box.
  c. Type Web Essentials and wait. After a little while, Web Essentials 2012 will appear at the top of the list in the center of the dialog box.
  d. Select its entry and then click Download to have it install.
- Download and install it manually:
  a. Open a browser and browse to `http://visualstudiogallery.msdn.microsoft.com/`.
  b. Click in the search box on the right-hand side, type Web Essentials 2012, and press Return.
  c. Click the entry for Web Essentials 2012 in the search results list and then click Download on the following page.
  d. Double-click the `WebEssentials2012.vsix` file you've just downloaded to install it.

After installing Web Essentials 2012, you can set a number of TypeScript compiler flags to be used by Web Essentials along with a couple of other options from within the Visual Studio Options dialog box. From the Tools menu, choose Options. When the Options dialog box appears, expand Web Essentials and then TypeScript, as shown in Figure 1-6.

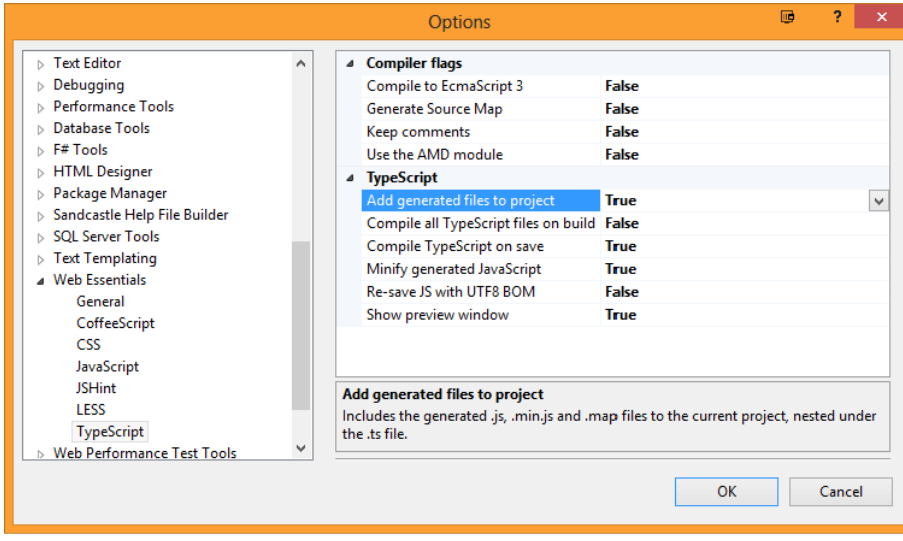

Figure 1-6. *TypeScript options for Web Essentials 2012*

For more information on the various compiler options, please see "The Compiler" in Chapter 3.

■ **Note** These compiler options control Web Essentials' compilation of your TypeScript files *across all your projects.* Mads Kristensen is currently working with the TypeScript team to come up with a less rigid set of options.

Other Visual Studio Editions

Although it may not be possible to enjoy the full TypeScript experience if you aren't using Visual Studio 2012 Professional or higher, remember that you can download a current version of the compiler and sample files from `http://typescript.codeplex.com` and go from there. There are also a few other things you can try.

Visual Studio 2012 Express Editions

The TLS extension for Visual Studio 2012 can be installed successfully into copies of Visual Studio 2012 Express for Web. However, if you have VS 2012 Express for Windows 8 or Windows Desktop installed, the compiler will install successfully, but the TLS extension will not.

The Web Essentials 2012 extension will not install on any Express edition of Visual Studio.

Visual Studio 2010 Professional and Higher

It is apparently possible to install the TLS extension in VS 2010, by installing the compiler as usual and then running the `TypeScriptLanguageService.vsix` installer manually from the installation directory. Of course, this is not supported by Microsoft.

WebMatrix

Microsoft does not officially provide an extension for the WebMatrix IDE. However, an unofficial extension is already available at `http://extensions.webmatrix.com/packages/TypeScript4WebMatrix/`. This is an open source extension, the source code for which can be found at `http://macawnl.github.com/TypeScript4WebMatrix/`.

On Node.js

The TypeScript compiler can be installed as an npm package on Node.js. Just type the following:

```
npm install -g TypeScript
```

You'll be able to compile code by running `tsc helloworld.ts`, for example. Note, however, that the compiler itself is not exposed as an instruction, although there is an issue raised in CodePlex to do this (WorkItem 97).

Sublime Text

If you're not a fan of Visual Studio, you can grab a copy of the TypeScript compiler (either from the CodePlex site or by installing the download) and then run it manually. Sublime Text 2 users can also find a set of TypeScript syntax highlighter rules written by Microsoft at http://blogs.msdn.com/b/interoperability/archive/2012/10/01/sublime-text-vi-emacs-typescript-enabled.aspx.

To install these rules and include TypeScript compilation in your build system, do the following:

1. Download the Sublime Text zip file from the MSDN Blogs page noted earlier and extract the contents. You'll find a file inside called `typescript.tmlanguage`.
2. Open Sublime Text and from the Preferences menu choose Browse Packages.
3. When Solution Explorer / Finder opens, create a folder in the current directory called TypeScript and copy the `.tmlanguage` file into it.
4. From the Tools menu, choose Build System ➤ New Build System.
5. Add the following to the new sublime-build file that has been created for you:

```
{
    "cmd": ["tsc","$file"],
    "file_regex": "(.*\\.ts?)\\s\\(([0-9]+)\\,([0-9]+)\\)\\:
\\s(...*?)$",
    "selector": "source.ts",
    "osx": {
       "path": "/usr/local/bin:/opt/local/bin"
    },
    "windows": {
        "cmd": ["tsc.cmd", "$file"]
    }
}
```

6. Restart Sublime Text 2.

From JavaScript to TypeScript

Now that you've installed TypeScript, let's look at a simple example. You'll take an HTML page with some JavaScript embedded in it and rewrite it as an HTML page backed by some unobtrusive TypeScript. You'll also see how to add jQuery and some basic structure to the script that might be easier to read coming from .NET.

■ **Note** The purpose of this exercise is not to learn TypeScript as such, but to illustrate some of the new syntax and how the TLS extension in Visual Studio can help.

Let's start by creating a new TypeScript project:

1. Open Visual Studio. From the File menu, choose New Project.
2. From the Visual C# templates list, select HTML Application with TypeScript and give it the name FirstSteps.
3. Click OK.

A Simple HTML+JS Page

The project template includes an HTML page backed with a TypeScript file that demonstrates a few more features than we'll use here, so it will be worth your while coming back to it in a while. In the meantime, you can delete the files `app.cs` and `app.ts`.

1. Press Ctrl+Shift+A, and add a new HTML page to the project called `SimplePage.html`.
2. Add the following code:

```
<!DOCTYPE html>
<html>
<head>
  <title>A Simple Demo</title>
  <script>
    function displayDate() {
      document.getElementById("txtDemo").innerHTML = Date();
    }
  </script>
</head>
<body>
  <h1>My First JavaScript</h1>
  <p id="txtDemo">This is a paragraph.</p>
  <button type="button" id="btnGo" onclick="displayDate()">
```

```
    Display Date
  </button>
</body>
</html>
```

3. Press Ctrl+F5 to run this page. You'll see that the current date and time replaces the paragraph text when you press the button.

TypeScript and IntelliSense

Now it's time to split out the script from the HTML and rewrite the script as a TypeScript file. The compiler targets only `.ts` files, so you can't just embed TypeScript in an HTML page. Here are the steps:

1. Create a copy of `SimplePage.html` and rename it `SimplePageTS.html`.
2. Delete the `<script>` block from the `<head>` element in `SimplePageTS.html` and also the `onclick` attribute from the `<button>` element.
3. Right-click `SimplePageTS.html` in Solution Explorer and then click Set as Start Page.
4. Press Ctrl+Shift+A, and add a new TypeScript file called `SimplePage.ts` to the project. Delete its contents and add the following, copying the `displayDate` function verbatim and adding an event handler for the button.

```
function displayDate() {
    document.getElementById("txtDemo").innerHTML = Date();
}

document.getElementById("btnGo").addEventListener("click",
displayDate);
```

Note that as you type, you get full IntelliSense support for the JavaScript language, the Document Object Model, and any functions you've written yourself. Also, if you hover your cursor over a type, function, or variable, you'll get a full set of information about it, as shown in Figure 1-7.

Figure 1-7. *The TypeScript Language Service displays type and method information as you hover*

5. Add a new `<script>` block *as the last child of the* `<body>` *element*. Set its `src` attribute to `SimplePage.js`. This is the JavaScript file our TypeScript will compile into. You could reference the `SimplePage.ts` file directly here and have it compiled on the fly on the web server, but the performance loss would make this not a particularly viable option.

Errors and Declaration Files

If you run `SimplePageTS.html`, you'll see that the page still functions as before, with the TypeScript file being compiled into JavaScript when the site is built. Let's introduce an error and see what happens.

In `SimplePage.ts`, change `displayDate()` to read as follows:

```
function displayDate() {
    var currentDate: Date = new Date();
    document.getElementById("txtDemo").innerHTML = currentDate;
}
```

You'll see that an error is raised, indicating that `currentDate` cannot be converted to a string (see Figure 1-8). Also, the right-hand pane showing the resultant compiled JavaScript has been grayed out.

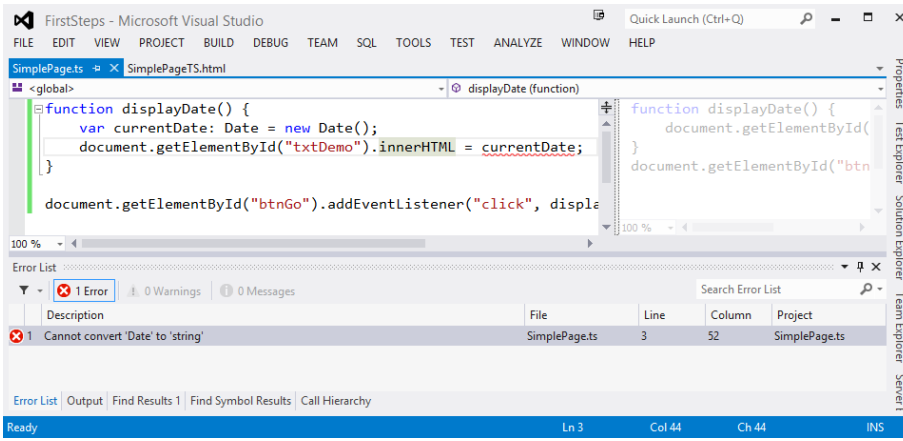

Figure 1-8. *TypeScript errors are underscored in the edit window and listed in the Error List window*

Why the error? All we did was replace the `Date()` constructor with a variable containing a `Date` object. Well, like the gotchas noted earlier, JavaScript silently coerces the `Date` object into a string in the original example. When we strongly type the `currentDate` variable as a `Date`, TypeScript does not allow this conversion to happen implicitly.

In `SimplePage.ts`, right-click innerHTML and select Go To Definition from the context menu (or press F12). Visual Studio will open the JavaScript runtime and DOM declaration file `lib.d.ts` and highlight the line showing that the `innerHTML` property of an `HTMLElement` object requires a string.

Return to `SimplePage.ts` and convert `currentDate` to a string with the following code:

```
document.getElementById("txtDemo").innerHTML = currentDate.toUTCString();
```

jQuery, Ambient Declarations, and Reference Hints

At this point, sprinkling a little jQuery into our simple function would make it a little more readable:

1. In `SimplePageTS.html`, add a reference to jQuery *in the head element.*

   ```
   <script src="http://code.jquery.com/jquery-latest.js"></script>
   ```

2. In SimplePage.ts, replace the existing code with the following jQuery equivalent:

```
function displayDateJQ(): void {
    var currentDate: Date = new Date();
    $("#txtDemo").text(currentDate.toUTCString());
}

$("#btnGo").click(displayDateJQ);
```

If you try to build the project now, you'll see that TypeScript raises an issue about the use of the jQuery $ variable. It's not defined in the .ts file and it hasn't been declared, so there's a problem. You'll also note that there is no IntelliSense for jQuery. There are two solutions to this.

The first is to add an ambient declaration for the $ variable by using the declare keyword at the top of SimplePage.ts.

```
declare var $: any;
```

This tells the compiler that $ is a variable of type any and that it is defined in some other (third-party library) JavaScript file, which the site has access to but the compiler does not. The compiler takes it on trust that you know what you're talking about. The file will compile, and the page will run. But you still don't get IntelliSense for jQuery.

The second option is to locate a declaration file for jQuery. Like lib.d.ts for the DOM, this will contain all the type information, and class and interface definitions for the jQuery library. The TLS will be able to refer to this, understand from it that $ is a variable of type JQueryStatic, and provide IntelliSense from there.

Add a new folder called lib to your project and add a copy of the latest jQuery declaration file to it. You'll find it at https://github.com/borisyankov/DefinitelyTyped/tree/master/jquery named jQuery-<version>.d.ts, where <version> is the version you want. At the time of this writing, the version is 1.8.

At the top of SimplePage.ts, add the following reference hint to the compiler. Remove the ambient declaration if you added it earlier.

```
/// <reference path="lib\jquery-1.8.d.ts" />
```

You'll see that the file now compiles, and the page will run. If you retype some of the function code, you'll see that you now have IntelliSense for jQuery functions too.

■ **Note** If you can find a declaration file for the third-party library you are using, make use of it. There is a suggestion on the CodePlex site for a declaration file community site (http://typescript.codeplex.com/workitem/191), but while that is being considered, have a look at https://github.com/borisyankov/DefinitelyTyped for now.

A Little Code Structure with Classes and Modules

In our final revision for this demonstration, you're going to see how TypeScript's class and module features can help you refactor your code into a more maintainable structure. As you'll see in Chapter 2, class and module are roughly analogous to C#'s class and namespace, allowing you to create objects as you would in .NET and give them some scope. In this simple example, adding classes and modules is overkill, of course, but it's useful to give you a quick flavor of how they work in TypeScript.

1. In `SimplePage.ts`, wrap the `DisplayDate` function inside a class called `Chapter1`. Note that you'll need to delete the `function` keyword preceding `DisplayDate`, as it is now an object method.

2. Update the event handler to bind `Chapter1.prototype.DisplayDate` to the button's click event, as shown in the following code.

```
class Chapter1 {
  DisplayDate(): void {
   var currentDate: Date = new Date();
            $("#txtDemo").text(currentDate.toUTCString());
        }
    }

$("#btnGo").click(Chapter1.prototype.DisplayDate);
```

TypeScript includes the ability to make object methods static so we don't have to go through an object's prototype to find its method reference for the event handler.

3. Make `DisplayDate` a static method by adding the `static` keyword before its name.

4. Update the event handler to bind `Chapter1.DisplayDate` to the button's click event.

```
class Chapter1 {
  static DisplayDate(): void {
   var currentDate: Date = new Date();
            $("#txtDemo").text(currentDate.toUTCString());
        }
    }

$("#btnGo").click(Chapter1.DisplayDate);
```

Finally, we'll wrap the class in a module. There are two rules to remember for accessing classes from outside the module it is declared in:

- The class must be made visible by using the `export` keyword.
- There's no equivalent keyword in TypeScript for the using keyword in C#, or imports in Visual Basic. References to the class or static method must be fully qualified with the module and class name.

5. Wrap the `Chapter1` class inside a module called `TypeScript.Revealed`.
6. Add the `export` keyword to the class statement so the event handler can access it.
7. Update the event handler to reflect the new fully qualified name for `DisplayDate`.

```
module TypeScript.Revealed {
    export class Chapter1 {
        static DisplayDate(): void {
            var currentDate: Date = new Date();
            $("#txtDemo").text(currentDate.toUTCString());
        }
    }
}

$("#btnGo").click(TypeScript.Revealed.Chapter1.DisplayDate);
```

Once again, you'll find that the page runs as before. Hopefully, you've seen how some of the new features of TypeScript—strong typing, ambient declarations, reference hints, declaration files, classes, and modules—can help you feel more at home creating TypeScript/JavaScript. Now might be a good time to go back to the `default.html` and `app.ts` files originally generated with the TypeScript project and review the further syntax features it demonstrates: class member variables and constructors, and lambda (arrow) functions. All of these and more are covered in Chapter 2.

You can find the code for the FirstSteps project in the Source Code/Download area of the Apress web site (www.apress.com).

Summary

JavaScript is in good health, but hasn't the structure, tooling, or inherent maintainability to be used for application-scale development. Or at least that's what Microsoft thinks. In this chapter, you've taken a wide-angle view of TypeScript—Microsoft's answer to these problems with JavaScript—and how it works. You've seen how to install it and then written your first code with it, using basic syntax features and the TLS service to rewrite some basic JavaScript embedded in a web page into a nicely structured piece of unobtrusive TypeScript.

In the next chapter, you'll look at the new language features of TypeScript.

CHAPTER 2

The New Syntax

In this chapter, you're going to look at the TypeScript additions to the JavaScript language. You'll learn about its strong typing features first and then move on to its object-oriented specific features. In particular, this chapter covers the following:

- Static typing
- Classes
- Interfaces
- Modules

TypeScript is a design-time-only language—that is, it exists only prior to compilation (into JavaScript) in your IDE—so I'll try to reveal as much of the tooling that's relevant to each feature as you go along as well.

Most of you reading this will be new to TypeScript, but not necessarily to JavaScript, so I'll assume you know at least some basic JavaScript as well as C# or Visual Basic. I'll drop into a discussion of JavaScript features only when relevant.

Tip If you're completely new to JavaScript, have a look at `www.w3schools.com/js/default.asp` for a quick guide and then come back. I'll wait.

The Story So Far

In Chapter 1, you saw how to create an HTML Application with TypeScript project and wrote your first TypeScript code:

```
/// <reference path="lib\jquery.1.8.d.ts" />

module TypeScript.Revealed {
    export class Chapter1 {
        static DisplayDate(): void {
```

```
            var currentDate: Date = new Date();
            $("#txtDemo").text(currentDate.toUTCString());
        }
    }
}

$("#btnGo").click(TypeScript.Revealed.Chapter1.DisplayDate);
```

You saw that as you typed, the TypeScript Language Service (TLS) that is part of the Visual Studio extension prompted you with type information and alerted you to any errors it detected in your code. You also learned how declaration files are used to provide TLS with IntelliSense and type-safe information for third-party libraries such as jQuery. Finally, you saw that when the project was compiled, the TypeScript file was compiled into native ECMAScript 3–compliant JavaScript for use in the HTML page you wrote to accompany it, as shown in Figure 2-1.

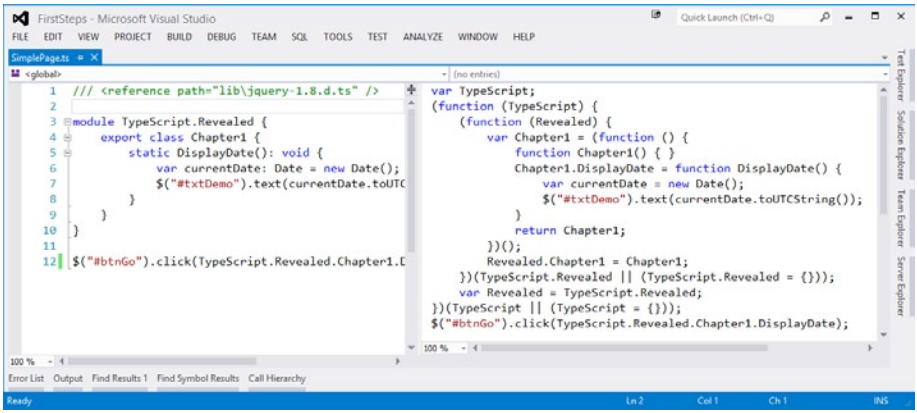

Figure 2-1. *The TLS displays the JavaScript your TypeScript will be compiled into*

You should have no problem figuring out how the code translates to the output, but let's point out a few features that you'll investigate further in this chapter:

- Line 1 references the jQuery declaration file so the TLS extension can provide IntelliSense for and type-safe calls into the jQuery library.
- Line 3 declares a module, `TypeScript.Revealed`. Modules function much the same way as namespaces do in C# or Visual Basic. However, there's no equivalent `using` or `imports` statement in TypeScript, so a function must be called using its fully qualified name, in this case `TypeScript.Revealed.Chapter1.DisplayDate` (line 12).

- Line 4 declares a simple class, `Chapter1`, much as we would create a class in C#.
- Line 6 declares a variable, `currentDate`, and gives it a type `Date`. The TLS extension now regards this as strongly typed and will flag an error if you try to use it like an `Array` or add it to a `Number`, for example.

These features are arguably the most important language features in TypeScript, but there are many more besides, as you'll see in this chapter.

The Type System

TypeScript may bring a level of maintainable structure to JavaScript development through its class and module features, but it is the optional static typing and type inference system that TypeScript—through the TypeScript Language Service extension for Visual Studio (TLS for short)—brings to design-time development that will reduce the number of errors found at compile time as well as the tooling. Thanks to a strongly typed system, the TLS extension can also support many features—such as IntelliSense, refactorings, renamings, go-to definitions, and cross-file compiling—that .NET developers take for granted these days. Of course, you have the option not to apply types to any of your TypeScript code and have it behave dynamically as raw JavaScript does, but if you do start to define types, the TLS can infer further types and provide refactoring information. In fact, even if you don't explicitly define a type for a variable, the TLS will attempt to infer one from the values and literals you give it according to the (many) rules laid out in the TypeScript specification. For example:

```
var a = 1584;
// type inferred as Number

var b = { height : 180, name : "Gandalf" };
// type inferred as { height : Number; name : String; }
```

Let's not get ahead of ourselves, however. TypeScript inherits its five primitive types—`Number`, `String`, `Boolean`, `undefined`, and `null`—from JavaScript. As you saw in Chapter 1, because TypeScript is strongly typed, some of the classic JavaScript gotchas caused by JavaScript converting values on the fly are easily caught by the TLS extension and by the compiler. For instance:

```
var oneNumber : number = 1;       // creates a Number
var oneString : string = "1";     // creates a String

console.log(oneNumber == oneString);
// JavaScript would output 'true'. TypeScript won't compile this
```

As you can see from the example, a variable's type is declared by using the `var` keyword and a postfix syntax rather than the prefix system we know from C# or Visual

Basic. The type is separated from the name of the variable by a colon. When you declare a variable, you have four options:

- Declare its type and value (as a literal) in one statement.
- Declare its type but no value. The value will be set to `undefined`.
- Declare its value but no type. The variable will be of type `Any` (that is, an old-school dynamic JavaScript variable), but its type may be inferred by the TLS based on its value.
- Declare neither value nor type. The variable will be of type `Any`, and its value will be `undefined`.

For example:

```
var numberOfDwarves : number = 13;
// Option 1. Type and value set

var numberOfGoblins : number;
// Option 2. Type set but value undefined

var numberOfHobbits = 1;
// Option 3. Value set and type inferred as number

var lengthOfFilm;
// Option 4. Neither value nor type set yet.
```

Variables can also be declared with an interface type or a class type. You'll look at both interfaces and classes later in this chapter. For now, you'll examine the other possibilities—that a variable is of a primitive, function, or array type.

■ **Note** Variables have varying levels of scope in TypeScript, depending on where they are declared. Those declared outside functions, classes, and modules have global scope and can be accessed by all code in the application or in the web page on which it is declared.

Primitive Types

From a .NET perspective, JavaScript primitive types are slightly limited. The `String` and `Boolean` types are as you would expect, and the `Number` type is the equivalent of `System.Double`; there is no integer type in TypeScript or JavaScript.

```
var heroName : string = "Bilbo Baggins";
var orcsNeedAShower : bool = true;
```

The `null` and `undefined` types, however, are often a source of confusion. In a nutshell, `undefined` is equivalent to a variable having no value or object assigned to it,

while null means an object has been assigned to the variable but that object is null (a la C#). So undefined is a value, while null is an object. To add a little more confusion, variables cannot be typed as Undefined or Null, but they can be given the values undefined and null explicitly (or by not setting a variable's value when declaring it, in undefined's case).

```
var numberOfGoblins : number;
// Same as var numberOfGoblins : number = undefined;

var ageOfGandalf = undefined;
// Same as var ageOfGandalf : any = undefined;

var ageOfSaruman : number = null;
// Primitives can be given the null value (confusingly)

var ageOfRadagast = null;
// Same as var ageOfRadagast : any = null;

var error1 : Null; // Throws an error
var error2 : Undefined; // Throws an error
```

■ **Note** There's a collection of JavaScript gotchas around null and undefined at www.codeproject.com/articles/182416/A-Collection-of-JavaScript-Gotchas#nullundefined if you aren't already confused enough.

The Any Type

I've mentioned it in passing so far, but there's one more variable type to cover in detail. If you do not assign a variable a type, the TLS assigns it the Any type. If you do this, or set it explicitly, it will be allowed to act as any standard, dynamically typed JavaScript variable that can be set any value, from complex objects to a simple number.

```
var changeling;      // assigned type Any implicitly
var morph : any;     // assigned type Any explicitly
```

When a variable is given the Any type, the TLS will try to infer the type of the variable you are using in order to make sure you're not doing anything potentially incorrect with it. However, if you do want to use it as a dumping ground for random values, that's entirely up to you. JavaScript allows this, and TypeScript does too.

Of course, as TypeScript actually gives you a strongly typed diving board over the shark-infested waters of dynamic JavaScript variables, you should always prefer to assign a specific type to a variable rather than the Any type. You'll get no IntelliSense help or red squigglies to help you deal with Any type variables correctly. Future iterations of the TypeScript compiler will also verify that there are no spurious variables of type Any in your program.

Arrays

If you're a C# developer, arrays use the same syntax and literals as you're already used to. An array type is defined by using the name of the type in the array followed by a pair of square brackets. For example:

```
var emptyArray: any[] = new Array();
```

Array literals are written as comma-separated lists surrounded by a pair of square brackets:

```
var actors: string[] =  ["Martin Freeman", "Ian Holm", "Elijah Wood"];
```

■ **Tip** Always prefer to declare an untyped array as `any[]` rather than just assign it to `new Array()`.

You can assign only one type to all the elements in an array. You can't state a set of allowed types. You can, however, declare the array to be of type `any[]`, in which case the array can hold any value of any type—numbers, objects, functions, other arrays, and so forth.

Arrays are of variable length, and not fixed length as in C#. You can retrieve the number of items in the array by using its `length` property and add or remove items with its `push` or `pop` functions as you would in .NET.

You can iterate through the items in an array by using either `for` or `for..in` loops as demonstrated here:

```
// standard for loop
for (var i = 0; i < actors.length; i++)
{
    console.log(actors[i]);
}

// for..in loop
for (var actor in actors)
{
    console.log(actor);
}
```

Note that you can reference an item in an array by using both a number and a string. However, using a string indexer returns a value of type `any` rather than one of type `string`. For example:

```
var asExpected = actors[0];        // returns string "Martin Freeman"
var gotcha = actors["Ian Holm"];   //  returns value of type any
```

The same is true of all arrays of type `T[]`. Using a numerical index returns a value of type `T`. Using a string index returns a value of type `Any`.

Arrays of Arrays

Multidimensional arrays, or arrays of arrays, are also allowed in TypeScript, and declared to be of type `T[][]`. For example:

```
var trilogies: string[][] = [
    ["An Unexpected Journey", "The Desolation of Smaug", "There and Back Again"],
    ["The Fellowship Of the Ring", "The Two Towers", "The Return Of The King"]
];
```

As you can see, multidimensional array literals can quickly appear a bit confusing (especially when dealing with arrays of functions, or arrays of arrays of functions), but they remain simply a comma-separated list of comma-separated lists. If you prefer, you can build them up programmatically:

```
var trilogies: string[][] = new Array();
trilogies.push(["An Unexpected Journey", "The Desolation of Smaug",
"There and Back Again"]);
trilogies.push(["The Fellowship Of the Ring", "The Two Towers",
"The Return Of The King"]);
```

■ **Note** The TypeScript spec says, "Arrays are obviously a prime candidate for generic treatment. We expect array type literals to eventually become syntactic sugar for generic instantiation of a global interface type `Array<T>`."

Casts

One of the more common issues of working with strongly typed variables is the need to cast between types from time to time. Naturally, it's preferable to use a method such as `ToString` or `ParseInt` to do the conversion for you, but there are occasions in both .NET and TypeScript programming when an explicit cast (boxing) is required. In the TypeScript specification (section 4.13), it's referred to as *type assertion*, but it's still casting. The syntax is to put the target type between < and > symbols and then place it in front of the variable or expression that returns a value. For example:

```
var a : int = <int>SomeNumberAsAString;
```

Those familiar with client-side web programming using JavaScript and the Document Object Model (DOM) may occasionally come across issues as a result of the strongly typed nature of TypeScript that would not arise in plain JavaScript. For example, in the Stack Overflow post-`http://stackoverflow.com/questions/12686927/typescript-casting-htmlelement`, a user copied some working DOM-manipulation code straight from JavaScript into

TypeScript and was told a method he was calling on an HTML element did not exist. It transpired that JavaScript was transparently converting the element his code selected (an `HTMLElement`) into a different type that did define the method (an `HTMLScriptElement`), but as TypeScript would not do that without an explicit cast, an error occurred. However, we can cast the type of the returned object to an `HTMLScriptElement`.

```
var script = <HTMLScriptElement>document.....
```

Don't forget that type assertions are only a design-time feature of TypeScript designed to make sure that you can indeed cast from one type to another and that what you call on the newly cast object is possible. TypeScript uses structural (duck) typing, so if a Duck object has the same methods and properties as a Pig object—Oink(), Snort(), and so forth—apart from it being a very strange duck, it can be cast into a Pig object.

> *Type assertions check for assignment compatibility in both directions. Thus, type assertions allow type conversions that might be correct, but aren't known to be correct.*
>
> Section 4.13 of the TypeScript specification

In TypeScript, then, you won't be able to cast an integer to an `HTMLScriptElement`, for instance, but you would in JavaScript, and you would find out something was wrong only at runtime.

Ambient Declarations

As you saw in Chapter 1, you can use ambient declarations to create a *placeholder* variable to represent an object or other variable not declared in full in your current TypeScript file. For example, you might want to use a third-party JavaScript library that does not have its own declaration file. You can use an ambient declaration to introduce an object created in that library into your script without having the compiler throw an error because it does not know what the variable is referring to. For example, if you wanted to add a reference to the global jQuery object $ without adding a reference to the jQuery declaration file, you would write this:

```
declare var $;
```

The key is to prefix the variable declaration with the `declare` keyword, as you saw in the example at the end of Chapter 1.

Functions

Perhaps the biggest difference between JavaScript and today's .NET languages is the way in which functions are used. Both use them to group code statements together for reuse, but in JavaScript (currently), functions are also the only way of structuring content.

Classes and modules in TypeScript are compiled down to constructor functions that put properties on an object instance. Closures (access to local function variables from outside the function) are used to encapsulate data. Functions are also first-class objects in JavaScript/TypeScript, so they can be assigned to a variable, passed as a parameter (for example, as a callback), or returned from a (factory) function.

TypeScript distinguishes between three types of functions:

- *Top-level (global) functions* are defined outside a class or module declaration. They have global scope and can be called at any point from code. They have a name.
- *Class methods* are named functions declared as part of an object class. They can be called only after the class has been instantiated unless they are marked as static.
- *Anonymous functions* do not have a name. They are typically used when declaring a function that will be used in only one place.

As it does with variable declarations, TypeScript allows you to strongly type a function's signature with the net result that the TLS alerts you to any invalid passed parameter values. It also provides you with IntelliSense for those parameters when you are calling the function (which turns out to be invaluable when trying to pass callback functions). For example:

```
function CelsiusToFahrenheit(celsius: number): number {
    return (celsius * (9 / 5) + 32);
}

function EncodeAndLog(
    unencodedString: string,
    encodefn: (rawString: string) => string): void {
      console.log(encodefn(unencodedString));
}

function CalculateArea(rect: {width: number; height: number;}): number {
    return rect.width * rect.height;
}
```

We can provide a type for each argument in a function's signature and its return type by using the same postfix syntax for typing variables. The parameters for the function are declared as a comma-separated list within a pair of parentheses, followed by a colon and the return type. If the function does not return a type, you should indicate this by using the `Void` type. As with variables, the TLS will do its best to infer the correct signature for those functions not given types explicitly. Just hover the cursor over a parameter or the return value, and a tooltip will appear, displaying what the TLS believes is the correct type, although it will default to the `Any` type if its use is ambiguous in some way.

```
function fnName(
    param1name : param1type,
    param2name : param2type,
    ...,
    paramNname : paramNtype) : returnType { ... }
```

Indeed, the net result is exactly the same as you would see declaring a type for a function expression (a variable holding a function) with one optional change. The return type of the function can be prefixed by either a colon or an arrow (=>). For example:

```
// Set to undefined
var logFn : (rawString: string) => void;

// Set to named function
var converterFn : (celsius: number) => number = CelsiusToFahrenheit;

// Set to anonymous function
var areaFn : (ellipse: {r1: number; r2: number;}) => number =
    function(ellipse) {
        return Math.PI * ellipse.r1 * ellipse.r2;
    };

// An array of typed functions - both syntaxes demonstrated
var areaCalculators : { (s: Shape) => number; }[];
var areaCalculators : { (s: Shape) : number; }[];
```

Functions can be assigned to variables only if their signature completely matches the "brand" of the variable. In other words, all parameter names and types must match—not just types. The comparison is not based solely on the position of the parameters and their types.

■ **Note** TypeScript does not currently support a documentation system such as XMLDoc for .NET. Discussion is continuing at http://typescript.codeplex.com/discussions/397660 as to which documentation style should be used.

Variations on a Signature

TypeScript functions may take three types of parameters when defined:

- Positional parameters
- Optional parameters
- Rest parameter

Positional Parameters

Positional parameters, declared as `paramName[:paramType]`, which you've used exclusively up to now, require you to specify a value for them when you call the procedure:

```
function CelsiusToFahrenheit(celsius: number): number {
    return (celsius * (9 / 5) + 32);
}
```

Optional Parameters

Optional parameters are declared with a postfix question mark: `paramName?[: paramType]`. These should be set as the last arguments in a call to a function:

```
var kelvin: number;
function CelsiusConverter (celsius: number,
    calculateKelvinToo?: bool = false): number {
    if ( calculateKelvinToo) { kelvin = celsius -273.1;  }
    return (celsius * (9 / 5) +32);
}
```

JavaScript doesn't have an explicit `hasValue` method as C# does to check whether an optional parameter has been given a value. The best way to overcome this is to specify a default value for the parameter. If the function is called without a value assigned to the optional parameter, it is given the default value.

```
function CelsiusConverter (celsius: number,
    calculateKelvinToo?: bool = false): number {
   ...
}
```

If you'd prefer not to assign a default value, you can check whether a parameter has been given a value by comparing it to `null` and `undefined`:

```
var kelvin: number;
function CelsiusConverter (celsius: number,
    calculateKelvinToo?: bool): number {
    if ( calculateKelvinToo !== null && calculateKelvinToo !== undefined ) {
       if ( calculateKelvinToo) { kelvin = celsius -273.1;  }
    }
    return (celsius * (9 / 5) +32);
}
```

Note that in this example, we take advantage of the fact that functions can access variables outside their scope, to make up for the fact that TypeScript/JavaScript has no equivalent to an `out` parameter in .NET.

Rest Parameter

The rest parameter, declared as `...paramName[:paramType]` (three periods), represents the last argument in a call to a procedure and can hold an arbitrary number of arguments in addition to those specified before it. These arguments are held in an array of type `T[]` that can be iterated over by using a `for` loop:

```
function CountDwarvesTallerThan(minHeight: number, ...dwarves: Dwarf[]) :
number {
    var count: number = 0;
    for (var i = 0; i < dwarves.length; i++) {
        if (dwarves[i].height > minHeight) {
            count++;
        }
    }
    return count;
}
```

Remember not to use a `for..in` loop over your rest parameter, as this will turn your arguments into strings rather than the desired type (unless, of course, they are strings anyway, in which case, job's a good 'un).

Function Overloads

TypeScript also supports the overloading of functions, albeit without quite the same efficacy as .NET. JavaScript has no concept of method overloading itself, so while the TLS enforces strong typing against the various overloads at design time, you still have to write a generalized form of the function that will work in "raw" JavaScript that discerns which one of the overloads you're using and operates accordingly.

All of this explanation really requires an example to clarify the picture.

In .NET, a method can be overloaded by creating multiple functions with the same name, but each working with different parameter types. For example, in C# I might overload a function called `CalculateArea` like so:

```
public double CalculateArea(Square shape)
{
   return shape.x * shape.y;
}

public double CalculateArea(Ellipse shape)
{
   return shape.r1 * shape.r2 * Math.PI;
}
```

```
public double CalculateArea(Triangle shape)
{
   return 0.5 * shape.x * shape.y;
}
```

The function name is the same, but the signature of each overload (that is, their name and parameters but, crucially, not the return type) is different.

In TypeScript, a method is *overloaded* by stacking each signature variant of the function on top of a single implementation of the function. For example:

```
function CalculateArea(shape : Square) : number;
function CalculateArea(shape : Ellipse) : number;
function CalculateArea(shape : Triangle): number;
function CalculateArea(shape : Shape) : number {     //  <-- The crucial line!!
   if (shape instanceof Square) {
     return (<Square>shape).x * (<Square>shape).y;
   }
   if (shape instanceof Ellipse) {
     return (<Ellipse>shape).r1 * (<Ellipse>shape).r2 * Math.PI;
   }
   if (shape instanceof Triangle) {
     return 0.5 * (<Triangle>shape).x * (<Triangle>shape).y;
   }
   throw new TypeError("Unsupported type!");
}
```

The key differences here are as follows:

- You must explicitly type your functions—parameters and return types—if you want to overload them.
- There is an additional generalized function signature at the bottom of the stack.
  - You need to generalize all parameters and the return type. Use optional parameters if the number of parameters varies plus default parameter values as appropriate.
  - You need to generalize the return type as well: it is key.
  - The signature of the implementation has to be compatible with all of the overloads.
- You must type-check the parameters within the function body. Again, the final JavaScript method is not type safe, so TypeScript forces you to write overloaded functions in this way. Use `typeof` to test for primitive types and `instanceof` for everything else.

Basically, you are creating just one function and giving it a number of signatures so that TypeScript doesn't give compile errors when you pass it differently typed parameters. This means that while you're still writing the code, the TLS will strongly type the function calls correctly, but when compiled to JavaScript, the concrete function alone will be visible. For example, the overloaded `CalculateArea` function in the preceding code compiles down into the following JavaScript:

```
function CalculateArea(shape) {
    if(shape instanceof Square) {
        return (shape).x * (shape).y;
    }
    if(shape instanceof Ellipse) {
        return (shape).r1 * (shape).r2 * Math.PI;
    }
    if(shape instanceof Triangle) {
        return 0.5 * (shape).x * (shape).y;
    }
    throw new TypeError("Unsupported type!");
}
```

And, unless you call `CalculateArea` from outside your TypeScript, you know that all the calls to this concrete function will work with the objects sent to them because all calls have been vetted by the TLS. (They may throw a `TypeError`, but that still counts as working in this case.)

Classes

As .NET developers, we take for granted that we can group properties, methods, and events into classes representing objects that interact within a system. It is fundamental to the concept of object-oriented programming that we can create a class, instantiate it, call methods, and handle events on it. Unless we're using unsafe code, we even take for granted that the garbage collector will free the memory the object used after it is no longer in use. JavaScript has been a capable, object-oriented language for some time now (as Douglas Crockford has taken pains to point out at `http://javascript.crockford.com/inheritance.html` and `http://javascript.crockford.com/prototypal.html`). However, for Java, C#, and Visual Basic developers, the way in which JavaScript implements said object-oriented capabilities is awkward. Fortunately, TypeScript implements a class syntax system that is very similar to C# and also very close to the class proposal in ECMAScript 6.

To define an empty class, the syntax is simply as follows:

```
class SimpleWebSocket {
}
```

When compiled into JavaScript, it looks like the following code. The TypeScript version is clearer and carries the bonus that you can cross-compile the code into JavaScript compliant with both ECMAScript 3 and ECMAScript 5 (and ECMAScript 6 too, in time, you would imagine) for free.

```
var SimpleWebSocket = (function () {
    function SimpleWebSocket() { }
    return SimpleWebSocket;
})();
```

Enough looking at what JavaScript is generated. The point is to get away from that. Have a look at *JavaScript: The Definitive Guide* by David Flanagan (O'Reilly Media, 2011) if you want to understand it.

■ **Note** TypeScript has no equivalent to structs in .NET.

Class Members

TypeScript classes provide for the same three types of class members you would expect to find in .NET:

- Properties and fields to store data
- Methods to define behavior
- Events to provide interactions between different objects and classes

Let's look at each of these in turn.

Properties and Fields

A common practice in .NET is to create a private field to store data and then a public property with `get` and `set` functions that allow you to implement any rules or constraints for retrieving or writing to the data in the field. In JavaScript, the option to create such a public property has only existed since ECMAScript 5 was released. As the majority of JavaScript VMs out there are still only ECMAScript 3–compliant, creating properties with getters and setters is not often done. In TypeScript, however, implementing public properties over private fields is allowed and enforced through the TLS extension in Visual Studio. The compiler just leaves them in when compiling down to ECMAScript 5 code and refactors them into methods when compiling into ECMAScript 3 code.

For example, to define a simple read-write field, we have this:

```
class SimpleWebSocket {
    serviceUrl: string;
}
```

If you wish to create a public property over a private field, you can include additional logic for reading and writing it in the getter and setter methods:

```
class SimpleWebSocket {
    private serviceUrl: string;
    get ServiceUrl(): string {
        // Fields and Properties are always referred to in class prefixed with 'this.'
        return this.serviceUrl;
    }

    set ServiceUrl(value: string) {
        this.serviceUrl = value;
    }
}
```

Points to note here:

- The `serviceUrl` private field is private only at design time. The TLS can enforce the intention of a private field only at design time, as JavaScript itself has no concept of a private member field. However, if the variable acts as a private one in TypeScript, it will do similarly in the compiled JavaScript.
- You cannot add a return type to a setter method. (It is assumed to be null.)
- Omitting the `get` or `set` property method does not mean that property is read-only or write-only in JavaScript as it would be in C#. As noted in the first point, you can still access the read-write private variable directly.

Whether you use fields or properties, you will always need to refer to them inside the class by using the `this.name` syntax, as demonstrated in the preceding code.

Methods

To define a method of a class, simply add a function in the class without the `function` keyword you would use if declaring a global function as demonstrated earlier in this chapter:

```
class SimpleWebSocket {
    Close(code: number, reason: string): void {
        console.log(code + " : " + reason);
        this.state = this.SocketState.CLOSED;
    }
}
```

As noted earlier, if you wish to overload a class method, you do so by stacking signatures for that method on top of each other and implementing a generalized form of the many functions:

```
class SimpleWebSocket {
    Close(code: string): void;
    Close(code: string, reason: string): void;
    Close(code: number): void;
    Close(code: number, reason: string): void;
    Close(code: any, reason?: string): void {
        var logEntry: string = code.toString();
        if (reason) { logEntry += (" : " + reason); }
        console.log(logEntry);
        this.state = this.SocketState.CLOSED;
    }
}
```

■ **Note** If you want to add a custom method to a built-in JavaScript object, you should do so on the object's `prototype` property. However, this breaks the TLS type analysis. See `http://typescript.codeplex.com/workitem/4` for the current workaround and status for this bug.

Arrow Functions

Another JavaScript gotcha that TypeScript helps us fix at design time concerns the `this` variable. You've seen previously that to reference class members from within other class members, you prefix them with `this`:

```
class SimpleWebSocket {
    state : number;

    Close(code: number, reason: string): void {
        console.log(code + " : " + reason);
        this.state = this.SocketState.CLOSED;
    }
}
```

However, when dealing with the DOM, it is possible to lose sight of what your `this` variable is referring to because its lexical scope has changed. Take, for example, any of the `window.onmouse*` events—`onmouseover`, `onmousemove`, and so forth.

When assigning any of these events a handler function, the function overrides `this` to refer not to your class, but *to the element your mouse has just interacted with*. In the following example, then, the function `beginMenuTest()` tries to alter the value of the `UITester` class property `menuTouches` twice. First it correctly resets it to zero,

but the second reference is within an anonymous function handling an `onmouseenter` event. Within the scope of the anonymous function, `this.menuTouches` refers to the `menuTouches` property of the menu object the mouse has just moved over—which probably doesn't exist.

```
class UITester {
    menuTouches : number;
    sidebarTouches : number;

    beginMenuTest(): void {
        this.menuTouches = 0;   // Right!!
        menu.onmouseenter = function (e) {
            this.menuTouches++;  // Wrong!!
        }
}
```

TypeScript provides the solution by allowing us to use *arrow functions* (which have a very similar syntax to lambda functions in C#) that retain a lexically scoped `this` variable.

In the `beginSidebarTest()` method, both references to `this.sidebarTouches` refer to the property of the `UITester` class:

```
    beginSidebarTest() : void {
        this.sidebarTouches = 0;  // Right!!
        sidebar.onmousemove = e => {
            this.sidebarTouches++;  // Still right!!
        }
    }
}
```

Note that the TLS warns you to use arrow functions only indirectly. You won't see any red squigglies highlighting something wrong with `beginMenuTest`, because there isn't. However, the TLS will infer this to be of type `UITester` in the first instance and of type `any` in the second instance (within the function handler), so IntelliSense will appear to stop working.

Constructors

For every class implementation, there is at least one constructor function that is executed automatically when an object of that class is created. Taking its cue from ECMAScript 6, class constructor functions are identified by using the `constructor` keyword:

```
class SimpleWebSocket {
    constructor (url: string) {
        this.serviceUrl = url;
        this.state = this.SocketState.OPEN;
        this.messagesSent = 0;
    }
}
```

You can also overload your constructor, but don't forget to treat it as you would any other overloaded method. At the bottom of your stack of signatures must be a generalized function providing the concrete implementation of the constructor used when compiled into JavaScript:

```
class SimpleWebSocket {
    constructor ();
    constructor (url: string);
    constructor (url?: string) {
        if (url !== null && url !== undefined) {
            this.ServiceUrl = url;
        }
        else {
            this.ServiceUrl = "http://localhost:80";
        }
        this.state = this.SocketState.OPEN;
        this.messagesSent = 0;
    }
}
```

Don't forget to reference your class properties and fields by using the `this` prefix.

Events

You saw in Chapter 1 how handler methods can be attached to events by using the DOM or jQuery:

```
// Using the DOM
document.getElementById("btnGo").addEventListener("click", displayDate);

// Using jQuery
$("#btnGo").click(displayDateJQ);
```

In C#, a class event is declared by using the `event` keyword and by then also declaring a delegate function to be called when the event has occurred:

```
// Declare the delegate (if using non-generic pattern).
public delegate void SampleEventHandler(object sender, SampleEventArgs e);

// Declare the event.
public event SampleEventHandler SampleEvent;
```

In TypeScript, these two declarations are combined into an optional class property defined to be of a certain function type but currently undefined:

```
class SimpleWebSocket {
    onOpen?: (ev: Event) => any;
}
```

Access Modifiers

You can add public and private accessors to the fields, properties, and methods inside a class:

- `public` (the default) indicates that the class member is available to all code in another module after it has been exported.
- `private` indicates that the class member is available only to other code in the same assembly.

For instance:

```
class SimpleWebSocket {
    private serviceUrl : string;
}
```

Note, however, that, as discussed earlier, the class member will not actually be accessible only to the class when compiled as JavaScript. It is only at design time that the TLS enforces the concept of private members.

Instantiating Classes

To create an object, you can instantiate a class by using the `new` keyword, exactly as you would in C#:

```
var TestSocket : SimpleWebSocket = new ⮨
    SimpleWebSocket("http://testthis.com");
```

After you've created the object, you can assign and retrieve that instance's properties and invoke its methods:

```
console.log("Opened a connection to " + TestSocket.ServiceUrl);
TestSocket.Send("Test the socket");
TestSocket.Close(200, "OK");
console.log(TestSocket.state.toString());
```

Check whether you can create an instance of a class by using an object literal.

Static Classes and Members

You can use the `static` keyword to indicate that a class property, field, or function is shared by all instances of a class, as you would in C# or by using the `shared` keyword in Visual Basic. To access the static member, use the name of the class without instantiating it:

```
class SimpleWebSocket {
    static origin: string;
```

```
    static ThrowError(code: number) {
        this.Close(code);   // NB. You can call non-static methods from a
static method.
    }
}

SimpleWebSocket.origin = "Program Name";
SimpleWebSocket.ThrowError(404);
```

Note that if you are referencing the static member from within a class function, you do not need to prefix it with `this`:

```
class SimpleWebSocket {
    constructor (url: string) {
        this.ServiceUrl = url;                      // non-static members.
'this' required
        this.state = this.SocketState.OPEN;
        this.messagesSent = 0;
        SimpleWebSocket.origin = "This Program";    // static member. No
'this' required.
    }

    static origin: string;
}
```

Unlike .NET, static methods do not have to call only static methods or reference only static fields and properties. Also, while there is no explicit way to label a class as static, it is perfectly correct to label every member of a class as static, which amounts to the same thing. There is just no way to have the TLS check that all members of a class are static. You have to do it yourself.

Object Literals vs. Anonymous Types

In .NET, anonymous types allow you to create objects without a class definition. The compiler then generates a class for you that has no name and only the properties you specify when creating the object. For instance:

```
var simpleAnonymousObject = new { This = 123, That = "A string" };
```

In TypeScript, a similar system occurs through the use of JSON-formatted object literals. A variable can be assigned an object literal—the `new` keyword is not required in TypeScript—and the TLS then infers the type of that variable.

```
var anon = { This : 123, That : "A String" };
```

Hover your cursor over the `anon` variable name and you'll see the inferred type, which in this case is

```
{
   This: number;
   That: string;
}
```

Inheritance

JavaScript mirrors the concept of class inheritance through the creation of a prototype chain on your objects. While TypeScript compiles the relationship between classes and base classes into similar code, the actual syntax to indicate that relationship in TypeScript is more akin to that of .NET, using the `extends` keyword to indicate an inheritance relationship.

```
class ComplexWebSocket extends SimpleWebSocket {
}
```

If you need to reference the base class from within the derived one—for example, to access the base class's (`constructor`) function, use the `super` keyword:

```
class ComplexWebSocket extends SimpleWebSocket {
    constructor (url: string, protocol: string) {
        super(url);            // calls the base class constructor function
        this.protocol = protocol;
    }

    protocol: string;

    ThrowError(code: number) {
        // some code
        super.Close(code);     // calls Close on the base class.
    }
}
```

In the case of instantiating an object of the derived type, the base class's `constructor` function will be called automatically if it isn't done so explicitly (as shown in the preceding code) within its own constructor.

■ **Note** If you follow the same practice as in .NET of writing one class per file, you'll need to make sure to add a reference to the file containing the parent class. TypeScript won't look around the project for the class you're referencing automatically. It must be told where to look.

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

class ComplexWebSocket extends SimpleWebSocket {

   ...

}
```

Overriding Methods from the Base Class

TypeScript allows you to override the method of a base class simply by writing another method in the derived class with the same method signature or of a valid subtype of the original signature. For instance, let's say the base class defines the method Send as follows:

```
Send(data: string): void {
    console.log(data);
    this.messagesSent++;
}
```

If you needed to override this method, you could add either of the following to the derived class, and all would be well. There is no need to mark it specifically as an overridden method as you would in C# or Visual Basic with an `override` or `Overrides` keyword, respectively:

```
Send(data: string): void {     //  <-- matches signature in base class
    // new code
}

Send(data: any): void {        //  <-- is a generalized form of the function
    // new code
}
```

At this point, within the derived class definition, you can call the new version of Send by calling `this.Send()` and the base class version of it by using `super.Send()`. However, you do not have access to `super.Send()` from an instantiation of the derived class.

```
var cws = new ComplexWebSocket("someUrl", "http");

//  Calls Send defined in ComplexWebSocket or base class version
//  if not defined in ComplexWebSocket
cws.Send("a message");

// Throws an error.
cws.super.Send("another msg");
```

If you want to override a function that is overloaded in the base class, note that you must list out all the signatures for that overloaded method in the derived class as well. You can't just override one form of the function; you must override the generalized form instead. For example, the `SimpleWebSocket` class defines a `Close` method with four overloads:

```
Close(code: string): void;
Close(code: string, reason: string): void;
Close(code: number): void;
Close(code: number, reason: string): void;
```

```
Close(code: any, reason?: string): void {
    var logEntry: string = code.toString();
    if (reason) { logEntry += (" : " + reason); }
    console.log(logEntry);
    this.state = this.SocketState.CLOSED;
}
```

To override it in the `ComplexWebSocket` class that derives from `SimpleWebSocket`, you'll need to include this:

```
Close(code: string): void;
Close(code: string, reason: string): void;
Close(code: number): void;
Close(code: number, reason: string): void;
Close(code: any, reason?: string): void {
   // ^^ or a further generalized form ^^ of the function signature
   // new code
}
```

Interfaces

In .NET, interfaces define a set of properties, methods, and events but do not provide their implementation. You cannot instantiate an interface as you can a class. Classes inherit from interfaces and must implement each item in the interface exactly as defined. In this way, interfaces perform a dual role in .NET:

- As a marker that a class has a certain kind of behavior—for instance, `IContainer`, `IEnumerable`, `IDisposable`.
- As a generalization of a group of classes without being a base type. For instance, you might give an interface as a function's return type indicating that any number of object types might be returned by the function but that they will all have a minimum level of common functionality as given in the interface. This idea is a cornerstone of dependency injection.

These same purposes are true of interfaces in TypeScript, but there are significant differences in how they are used.

- They are a design-time-only construct for the TLS to consume and provide useful type checking, IntelliSense, and refactoring services, and for us to use as a shortcut. They vanish after compilation, replaced with the object types they represent.
- They are not inherited by classes. They are object types for application to variables, parameters, and return types.
- TypeScript interfaces are open-ended. That is, rather like partial classes in .NET, they can be defined across several files and the combination of the parts is the final object type actually applied to variables assigned that interface.

Common-sense dictates that we'll use interfaces as a refactoring tool for function types, behavioral sets of functions a la `IComparable`, and full class definitions.

■ **Note** There's no convention set for naming interfaces in TypeScript, but we'll prefix them with `IFn` for function types and `I` otherwise, as is done in .NET.

Using Interfaces as Function Types

If one of the goals of TypeScript is to make JavaScript more maintainable, then using interfaces to represent function types will definitely help. It can certainly make things more legible, and the clearer code is, the easier it is to maintain. Consider the following code:

```
var sayHello: (input: string) : string = function (s: string) {
    return "Hello " + s;
}
var stringUtils: { (input: string): string; }[];
stringUtils.push(sayHello);
```

Even though `sayHello` is just a function that takes a string and returns one, and `stringUtils` is an array of such functions, it's not difficult to see how such types can quickly become difficult to write and interpret with more parameters, perhaps some of those being functions and arrays of functions everywhere. A parenthetical crisis!

Fortunately, we can refactor the function signature into an interface, and things become instantly more legible. And bracket free.

```
interface IFnStringManipulator {
    (input: string) : string;
}

var sayHello: IFnStringManipulator = function (s: string) {
    return "Hello " + s;
}
var stringUtils: IFnStringMaipulator[];
stringUtils.push(sayHello);
```

At this point, we should note that the tooling to go with this useful refactoring is a bit lacking at the moment. There's currently no support in Visual Studio for the Extract Interface refactoring available with .NET languages. (See the discussion at http://typescript.codeplex.com/discussions/400724.) Also, IntelliSense tells you that `sayHello` is of type `IFnStringManipulator` rather than presenting the actual function signature. Of course, you can right-click `IFnStringManipulator` and choose Go To Definition (F12) to see what it is, but the to-and-fro between files is a pain.

Using Interfaces as Object Types

As .NET developers, it's easy to understand why defining interfaces as behavioral sets of functions is a good idea. It's quite easy to translate those concepts into TypeScript. For instance:

```
interface IComparable {
    CompareTo(obj: any): number;
}

interface IEnumerable {
    [index: number]: any;
}

interface ICloneable {
    Clone(): any;
}
```

And it's not too difficult to check whether an object implements an interface either. You need to check that all the elements in the interface exist on the object.

```
function ImplementsICloneable (obj : any) :bool {
    return (obj && obj.Clone);
}
```

However, why define interfaces that represent whole classes? Why not just write classes? Well, the TLS understands and uses both classes and interfaces to provide IntelliSense and other type-related services, *but* there are (currently) very few third-party JavaScript libraries out there that also come in TypeScript, let alone in classes, for the TLS to consume and help us code more accurately. Hence declaration files exist, making full use of interfaces and ambient variable declarations to represent complete classes in these libraries where TypeScript classes are not available.

Rambling aside, interfaces representing object types can contain placeholders for the following:

- Constructors
- Fields
- Optional fields
- Methods
- Overloaded methods
- Indexers

As in .NET, you cannot add static members to interfaces, and currently you cannot specify that a field should be private and accessed through getter and setter methods. To demonstrate, here's the nearest interface equivalent for the `SimpleWebSocket` class we've used in this chapter.

```
// Entire Class Interface
interface IWebSocket {
    // use 'new' to indicate constructor
    new (url: string) : SimpleWebSocket;

    // fields
    state: number;
    messagesSent: number;

    // optional fields
    protocols?: string;

    // properties can be added
    // but getters and setters cannot be enforced
    ServiceUrl: string;

    // methods
    Send(data: string): void;
    onError(errorCode: number, message: string): void;

    // 'overloaded' method
    Close(code: string): void;
    Close(code: string, reason: string): void;
    Close(code: number): void;
    Close(code: number, reason: string): void;

    // indexer - socket[12] returns message #12 perhaps
    [index: number] : string;
}
```

Note how we use the `new` keyword to indicate a constructor function in the interface rather than `constructor`. Also the entry for the overloaded `Close` method does not include the generalized function signature we actually need to use to implement it.

Combining Interfaces

In .NET, a class can inherit more than one interface. In TypeScript, an interface represents an object type. Variables and parameters can't have more than one object type, so how can we create a variable based on `IEnumerable` and `ICloneable`, for instance? The answer is to create another interface that does inherit from both `IEnumerable` and `ICloneable`:

```
interface IEnumerableClone extends IEnumerable, ICloneable {
    // any overrides or additional items
}
```

Now you can set an object to be of the new derived type:

```
var BobaFett : IEnumerableClone;
```

An interface can inherit from zero or more base interfaces. Inheritance is expressed using the `extends` keyword followed by a comma-separated list of the base interfaces. As with classes, you can choose to use the base interface members in the new child interface or to hide them using new declarations. The TypeScript specification (section 7.1) lays out the rules for hiding base interface members:

- A base interface property is hidden by a property declaration with the same name.
- A base interface call signature is hidden by a call signature declaration with the same number of parameters and identical parameter types in the respective positions.
- A base interface construct signature is hidden by a construct signature declaration with the same number of parameters and identical parameter types in the respective positions.
- A base interface index signature is hidden by an index signature declaration with the same parameter type.

The following constraints must be satisfied by an interface declaration, or otherwise a compile-time error occurs:

- An interface cannot, directly or indirectly, be a base interface of itself.
- Inherited properties with the same name must have identical types.
- The declared interface must be a subtype of each of its base interfaces.

The net result is that if you're inheriting from two or more base interfaces that define items of the same name but different types, your child interface needs to redefine that item by using a more generalized form, much the same as you need to do when overloading a method.

Modules

Whether you're more familiar working with `System.Web` or `System.IO`, if you develop against the .NET Framework, you're working with namespaces all the time. TypeScript supports some of the ECMAScript 6 functionality for modules.

The basic module declaration is exactly the same as for classes or interfaces, but uses the `module` keyword instead of `class` or `interface`. However, modules can also be nested, unlike classes and interfaces. For instance:

```
module MyWebApp {
    module DataAccess {
        // code here
    }
    module BusinessLogic {
        // code here
    }
    module UI {
        // code here
    }

    // global code here
}
```

Of course, we don't often actually nest modules in .NET, preferring to write code in files pertaining to a single class or web page in a namespace. The preceding structure can be written as follows in TypeScript, using the additive names for the modules:

```
module MyWebApp.DataAccess {
        // code here
}

module MyWebApp.BusinessLogic {
       // code here
}

module MyWebApp.UI {
        // code here
}

// global code here
```

However you prefer to write them, modules can contain any of the language features you've seen in this chapter so far and two more:

- Classes
- Interfaces
- Global functions and variables
- Ambient declarations
- Import declarations
- Export declarations

In terms of structuring your code, they are effectively the same as any namespace in .NET, both logically and physically. If you wish to build modules across many files as you would namespaces (for instance, to write one class per file), you can do that too. Any elements within the same module can be accessed by each other. However, if they are located in separate files, you will need to add a reference from the top of one file to the other.

For example, you saw in the section on class inheritance that by keeping to a rule of *one class per file*, the child class must include a reference to that of the parent class before it will compile:

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

As long as the `<reference>`s are there, the TLS knows and the compiler will pick up the reference chain. However, we must remember that class names must be fully qualified when written inside a module. For instance:

```
In SimpleWebSocket.ts:

module MyWebApp.BusinessLogic {
    class SimpleWebSocket {
        // code
    }
}

In ComplexWebSocket.ts:

/// <reference path="SimpleWebSocket.ts" />
module MyWebApp.BusinessLogic {
    class ComplexWebSocket extends MyWebApp.BusinessLogic.SimpleWebSocket {
    }
}
```

Although, as it happens, referencing a class name in the same module is an exception to a golden rule.

If you want to instantiate any class, reference any interface, use any global method, or access any global variable within a module from a different file, a different module or from global code, *you will need to export it*.

TypeScript considers each declaration within a module as private unless you expose it to other code by exposing it using the `export` keyword. The exceptions to this are class members that become accessible when you mark the class declaration with `export`.

```
In SimpleWebSocket.ts:

module MyWebApp.BusinessLogic {

    export class SimpleWebSocket {
       ...
    }
```

```
    // functions global to module but not in a class
    // must be exported individually
    export function Initialise() {}
    export var CurrentVisitors : number;
}

In another file:

/// <reference path="SimpleWebSocket.ts" />

// requires class is exported
var TestSocket = new MyWebApp.BusinessLogic.SimpleWebSocket("http");
TestSocket.Send("Some info");
TestSocket.Close("123");
console.log(TestSocket.state);

// Static class property
MyWebApp.BusinessLogic.SimpleWebSocket.origin = "here";

// In MyWebApp.BusinessLogic module but not in class
MyWebApp.BusinessLogic.Initialise();
MyWebApp.BusinessLogic.CurrentVisitors++;
```

The TLS does a good job of letting you know you've forgotten to export something. Generally, you'll get a good case of the red squigglies if you're attempting to access code in a module that hasn't been exported.

However, you *are* stuck with using the fully qualified names for module-scoped functions, interfaces, classes, and variables, as shown in the preceding code. TypeScript has no direct equivalent of the C# `using` or Visual Basic `Imports` statement. It is possible, though, to use the TypeScript `import` statement to create an alias for the module. For example:

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

import BL = MyWebApp.BusinessLogic;

// class declaration
var TestSocket = new BL.SimpleWebSocket("http");

// Static class property
BL.SimpleWebSocket.origin = "here";

// Module-scoped function and property
BL.Initialise();
BL.CurrentVisitors++;
```

■ **Note** These import declarations for your own modules are evaluated lazily. They can reference modules that haven't been instantiated yet, but you can't then use them to reference items in that module until the module *has* been instantiated.

All of this brings us to the point that so far in this section we have been dealing solely with modules we have written for the same application. TypeScript divides modules into two categories:

- If a module does not contain `export` statements (but likely references those that do), it is an internal module.
- If a module does contain `export` statements, it is an external module.

External modules also describe third-party JavaScript modules that we've not written but wish to use. For example, if we are writing a Node.js application, we will need to import the (public/exported) contents of the main node module to access its basic functionality. Alternatively, we might want to use Modernizr (`modernizr.com`) to check a browser's support for new HTML5 APIs. Think of these types of external modules as you would the libraries you might add to your .NET projects using NuGet.

To use such external libraries, download the TypeScript files or declaration files (`.d.ts`) if they exist (or the JavaScript files if they don't) so the TLS can carry on type checking and providing IntelliSense to your project. Then you import the library into your TypeScript file using a slightly different syntax to that used earlier:

```
import ModuleAlias = module('ModuleName');
```

In this case, the `ModuleName` in question is the location (absolute or relative) of the file containing the target module to be imported minus the `.js` at the end:

```
import Modernizr = module('c:\src\lib\modernizr');
import svr = module('..\lib\node');
```

■ **Tip** Section 9.4.1 in the TypeScript specification contains the full rules for module name resolution.

After the module has been imported, you reference the module's contents in the same way as internal modules. This means that unless you're referencing external modules written in TypeScript, you'll want a declaration file for it as well to maintain the tooling.

■ **Tip** You'll find a number of community-written declaration files at https://github.com/borisyankov/DefinitelyTyped.

One final note about modules. The TypeScript compiler will transpile your modules into one of two styles of JavaScript module:

- CommonJS modules
- AMD modules

If you always write TypeScript applications with HTML or use TypeScript within web applications or Windows 8 applications, you may never need to use modules for any purpose other than code structure and not need to worry about this fact. However, if you begin to write applications using Node.js, or some JavaScript dependency loaders, you'll need to take note of which style module they use and consult Chapter 3 to make sure the compiler creates the module type you need.

Summary

In this chapter, you've looked at the new language features that TypeScript appends to JavaScript. You've seen how they mirror similar features in .NET and how they differ in certain cases as well.

In the next chapter, you'll see how to take control of the TypeScript compiler and how to incorporate TypeScript into your existing development projects.

CHAPTER 3

Working with TypeScript

So now you're up to speed with the new syntax and language features of TypeScript, it's time to put them to use and include TypeScript in your own projects. In this chapter, you'll look at how to get the most from the compiler and then how to include TypeScript in your Visual Studio projects.

The Compiler

Thus far, your interaction with the compiler has been to watch Visual Studio use it in the background when you've asked it to build your sample projects. In this section, you'll look at how to use it manually so you can tweak its operation if necessary. The Web Essentials Options dialog box for TypeScript support that you saw in Chapter 1 (and that's shown in Figure 3-1) has already given you a clue to the various options you can use, but there are some other points to consider.

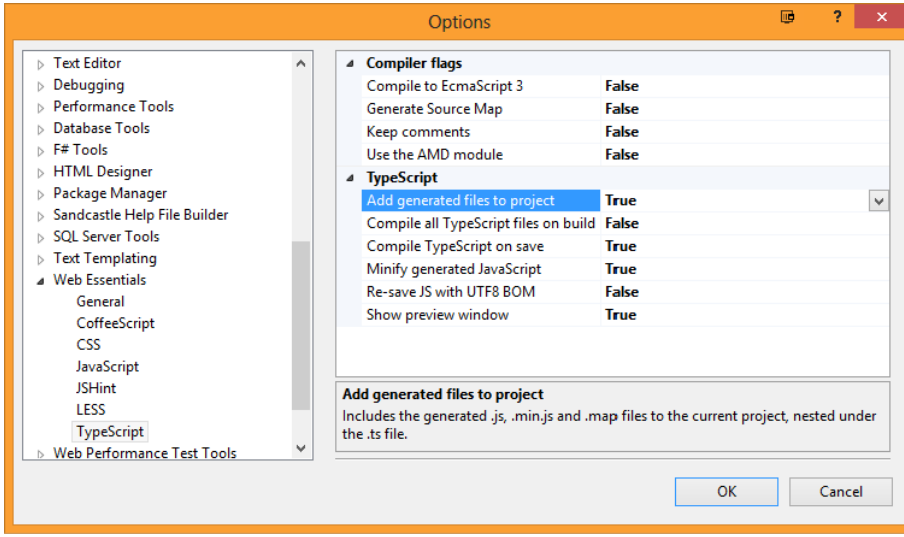

Figure 3-1. *The Web Essentials Options dialog box for TypeScript*

As you saw in Chapter 1, the TypeScript compiler is installed in the `%ProgramFiles (x86)%\Windows SDK\Typescript\<VERSION>` directory on 64-bit systems (`%ProgramFiles%\Windows SDK\Typescript\<VERSION>` on 32-bit) and is a single JavaScript file, `tsc.js`, that can be run using any ECMAScript 3–compliant JavaScript engine. Another file—`tsc.exe`—allows us to work with ithe compilerfrom the command line, and deals with the switches and file batches we want to compile.

■ **Note** The compiler's current file I/O infrastructure supports only Node.js and Windows Script Host file APIs, which is not great news if you're running on OS X or Linux/Unix. Get in touch with the team at the CodePlex site (`http://typescript.codeplex.com/discussions/404060`) to request better native support.

Assuming the installation directory has been added to your `PATH`, you can call the compiler from the command-line by using the following syntax:

```
> tsc @<file>
> tsc [options] <fileList>
```

There are ten options you can send to the compiler. The first two tell the compiler to generate something in addition to compiled JavaScript:

- `--declaration` requires the compiler to generate the `.d.ts` declaration file for the input files as well as the compiled JavaScript.
- `--sourcemap` requires the compiler to generate the `.map` source map files needed for step-through debugging of your TypeScript files.

For example, calling

```
tsc --declaration test.js
tsc --sourcemap test.js
```

will produce the files `test.d.ts` and `test.map`, respectively, as well as `test.js`.

- If you're using TypeScript as a Node.js package, you can call `tsc -w <fileList>` to have the compiler watch all the files in `<fileList>` and recompile them when they have been altered.

The next few options tweak the pre- and post-compile tasks for the compiler:

- `--nolib` requires the compiler to ignore its default and not to include `lib.d.ts` in its compilation even if you `<reference>` it explicitly in the file.

- `--out <outputFile>` requires the compiler to output all its compiled JavaScript into a single file called `<outputFile>` rather than the default of producing one JavaScript file for each TypeScript file in the `<fileList>` (with the same name: `a.ts` becomes `a.js`, `b.ts` becomes `b.js`, and so on).
- `-e` or `--exec` requires the compiler to execute the newly compiled script after it has been created.

Remember that scripts are run from top to bottom, so any code not in a class or module is run. There's no single entry point, like a `Main` function in a .NET command-line application. Also, if you use this option, be sure your global functions aren't expecting to run in a browser, or you'll get errors straight away.

```
tsc -e saveFaramir.ts
tsc --nolib shire.ts
tsc --out fellowship.js hobbit.ts human.ts elf.ts dwarf.ts
```

Another three influence the contents of the actual JavaScript emitted by the compiler:

- `-c` or `--comments` requires the compiler to include the comments you added to the TypeScript file in the compiled JavaScript.
- `--module <type>` requires the compiler to generate the specified `<type>` of modules during compilation. By default, it will generate `commonjs` modules (for Node.js scripts), but you can also specify `amd` modules (for example, for the RequireJS runtime module loader). See the section on modules in Chapter 2 for more information on the difference between the two.
- `--target <version>` requires the compiler to emit JavaScript compliant with the specified `<version>` of ECMAScript. By default, it will generate script compliant with ECMAScript 3 (`ES3`), but you can also specify `ES5` to have it emit ECMAScript 5–compliant script instead.

  ```
  tsc --comments howDidThisWorkAgain.ts
  tsc --module amd somethingForRequirejs.ts
  tsc --target ES5 newAndShiny.ts
  ```

- Finally, you can also use the `-h` or `--help` option to bring up a quick recap of these options onscreen.

If you are setting a lot of options and including a large number of files in the compilation, you can copy all the switches and the file list into a text file and

call `tsc @file.txt` to save typing it all out over and over again. `File.txt`, meanwhile, would have each switch and file to compile separated by a new line, like this:

```
--out fellowship.js
--module amd
--comments
hobbit.ts
human.ts
elf.ts
dwarf.ts
```

The File List

The file list you supply to the compiler is a space-separated list of all the files you want it to compile. This is much like the list you would send to the C# compiler (`csc.exe`), but with one difference. With `csc.exe`, you also need to include a list of references to the DLLs containing namespaces and classes your own C# code uses. This is not necessary with TypeScript. The `tsc` compiler will scan the top of each file in the list supplied for `<reference file="...">` statements and include those referenced (both `.d.ts` and `.ts`) in its list for compilation. It does the same for files referenced via an `import` statement in your code.

It will also check those files for `<reference>` and `import` statements, include those files, and so on, until it has completely walked through the entire reference tree and has a complete list of files to compile. The net result is that you don't need to include any declaration files in the file list for the compiler and perhaps fewer `.ts` files than you thought as well.

Also, unless you send it the `--nolib` option, the compiler will automatically include the `lib.d.ts` declaration file for the standard JavaScript runtime library and DOM API. The TLS does the same walk of `<references>` and `import` statements as it provides IntelliSense, so if IntelliSense appears to be working correctly across all your TypeScript files, you know your reference tree is intact as well, ready for compilation.

A QUICK COMPILER Q & A

Q: Can you include `.js` files in your compilation?

A: No. The compiler deals with only `.ts` or `.d.ts` files. Don't try to pass it any "raw" JavaScript files, because it will just reject them. If you need to reference something in a `.js` file, you can do one of the following:

- Create a declaration file for it (see the upcoming "Declaration Files" section in this chapter).
- Include an ambient declaration in your TypeScript file for the variable or function you need to use.
- Rename a copy of your `.js` file to `.ts` and see if it compiles.

Q: Why doesn't the compiler generate a file if I pass it a declaration file?

A: If your file contains only comments, interface definitions, or other type information, the compiler does not generate any code, and so it does not generate a file, empty or otherwise (http://typescript.codeplex.com/workitem/284).

Q: Why does the compiler slow down when working with Windows 8 apps?

A: It's not specifically Windows 8 apps. The compiler slows down when working with really large files such as `winrt.d.ts`, which you need for Win 8 development (http://typescript.codeplex.com/workitem/265).

Q: How do I pipe compiler errors from `tsc` to my own log file rather than to the command line? (http://stackoverflow.com/questions/12897695/typescript-tsc-redirection-of-messages)

A: The compiler writes errors to `stderr` rather than `stdout`, so you'll need to use the following command to pipe the errors:`> tsc file.js > err.log 2>&1`

Declaration Files

As soon as you start using Backbone.js, jQuery, jQuery UI, or any other third-party JavaScript library with your own TypeScript code, you'll need to start using declaration files as well. Add a quick annotation at the top of your file, for example

```
/// <reference path="lib\jquery-1.8.d.ts" />
```

and the TLS can incorporate the interfaces and variables for jQuery into its IntelliSense prompts and type inference/checking. It's a very neat system that demonstrates the need for ambient declarations (in case you're wondering). Trouble is, what happens when the declaration file doesn't exist for the library you want to use, be it a third-party library or one your own team has created? You have four choices at this point:

- Double-check that it's not available somewhere online.
- Use an ambient declaration in your code to identify the library's existence.
- Use the TypeScript compiler to generate a baseline declaration file.
- Roll your own declaration file.

Let's look at each of these in turn.

Has Someone Else Written It?

As you know by now, the TypeScript team has written four declaration files:

- `lib.d.ts` covers the standard JavaScript Runtime Library, HTML5 APIs, and the DOM API.
- `jquery.d.ts` covers the definitions for jQuery 1.7.

- `winjs.d.ts` covers the Windows Library for JavaScript (for Windows 8 JavaScript apps).
- `winrt.d.ts` covers the Windows Runtime Library (for Windows 8 apps).

Beyond these four, the TypeScript community has already done quite a bit of work on both improving these four files and creating many more besides. Currently, the unofficial repository for declaration files is https://github.com/borisyankov/DefinitelyTyped. Here you'll find updates for the four Microsoft declaration files and several dozen more libraries from AngularJS to QUnit to WebGL. And, of course, if you can't find the declaration file on DefinitelyTyped, then Google may still be your friend.

■ **Note** There is a discussion at http://typescript.codeplex.com/discussions/398580 about an official repository for declaration files and how best to make them available (NuGet, download, something else) and another at http://typescript.codeplex.com/workitem/267 about how best to correlate the version of the declaration file with the version of the library against which it is written. Both are ongoing as this book went to press.

Substitute with an Ambient Declaration

All declaration files contain interface definitions for the classes defined in a library and ambient declarations for global variables of those classes. Without these, both the compiler and TLS will complain as soon as you use one of those global variables in your code, and it will not compile.

If you are sufficiently secure in your knowledge of the library you want to use, you can add an ambient declaration for the global variable you want to use and assign it the `any` type. For instance, let's suppose you want to use jQuery but don't have the declaration file at hand. You can declare its global variable like so:

```
declare var $ : any;
```

and continue to use jQuery with no issue. You'll get no IntelliSense or type checking—everything will be of the `any` type—but that's how we worked with jQuery before anyway.

Use the Compiler to Generate a Declaration File

You saw earlier that the TypeScript compiler will generate a declaration file for you if you pass it the `--declaration` flag. For example:

```
> tsc --declaration SimpleLibrary.ts
```

Assuming `SimpleLibrary.ts` compiles correctly, the compiler will generate `SimpleLibrary.js` for inclusion in your project releases and `SimpleLibrary.d.ts` for use in your development work.

This sounds like an ideal solution, and it is—for libraries written in TypeScript. Unfortunately for JavaScript libraries, the compiler accepts only valid TypeScript files. If I wanted to generate my own declaration file for jQuery, I couldn't just download `jquery.js`, rename it to `jquery.ts`, and point the compiler at it: the compiler would throw many errors at it. And even if it did compile, the resultant declaration file would not contain much useful type information and perhaps lose some of the subtlety in the library. Because it is written in plain JavaScript, the compiler perceives all parameters to be of the `any` type rather than strings, numbers, and so forth. Remember: it is the TLS in the Visual Studio extension that infers types rather than the compiler.

All in all, unless you're referencing a library written in TypeScript, this is the most unsatisfactory solution of the four. Even Microsoft went down the route of writing a custom script to produce its declaration files based on Web IDL and standards definitions rather than pass the actual JavaScript libraries through the compiler.

■ **Note** Interestingly, some developers have decided that the best solution is to rewrite their favorite libraries in TypeScript so declaration files can be quickly generated to match the latest versions—for example, `backbone.ts` at https://github.com/jbaldwin/backbone.ts.

Roll Your Own

Like it or not, your best shot may be to write your own declaration file. Remember that a declaration file is just a collection of interfaces and ambient declarations for global variables, values, and functions. For example, `lib.d.ts` starts by declaring a handful of familiar number-related values before going on to define the interface for the `number` object type and then a class variable for it:

```
declare var NaN: number;
declare var Infinity: number;
declare function eval(x: string): any;
declare function parseInt(s: string, radix?: number): number;
declare function parseFloat(string: string): number;
declare function isNaN(number: number): bool;
declare function isFinite(number: number): bool;

...

interface Number {
    toString(radix?: number): string;
    toFixed(fractionDigits?: number): string;
    toExponential(fractionDigits?: number): string;
    toPrecision(precision: number): string;
}
declare var Number: {
    new (value?: any): Number;
    (value?: any): number;
```

```
    prototype: Number;
    MAX_VALUE: number;
    MIN_VALUE: number;
    NaN: number;
    NEGATIVE_INFINITY: number;
    POSITIVE_INFINITY: number;
}
```

One note, though: remember to include a prototype property inside your variable declaration to generate the correct OO properties for JavaScript.

TypeScript in Your Projects

It's time at last to look at how best to incorporate TypeScript into one of your own projects rather than experiment with it in one of its own. Necessity dictates being brief in this discussion about writing actual code in favor of how to set up your projects, so it goes without saying that you should also look at the eight sample apps available at http://typescriptlang.org/samples. Each demonstrates how TypeScript fits in with a certain kind of project and set of libraries:

- Hello World: Basic TypeScript
- Raytracer: DOM canvas
- TodoMVC: Backbone.js and jQuery
- ImageBoard: Node.js, Express, and MongoDB
- JQuery Parallax Starfield: jQuery (natch)
- D3 Visualization: D3.js (Data-Driven Documents) and HTML5 canvas
- Warship Combat: jQuery and jQuery UI
- Encyclopedia App: Windows 8 Store application

Be it a Web or Windows 8 Application project, where you previously had JavaScript, you can now have TypeScript generating JavaScript instead. It's just a matter of adding a new TypeScript file to the project instead of a JavaScript file, isn't it?

Mostly, yes. However, when using TypeScript with anything other than the accompanying HTML Application with TypeScript project template, you'll need to tweak your project's MSBuild (`.proj`) file to invoke the TypeScript compiler. You'll look at that next and follow up with a few additional considerations to make if you're adding TypeScript to either a Web Site project or a project that uses the ASP.NET 4.5 Web Optimization library.

If you're an MVC user, you may also want to have a look at https://github.com/devcurry/typescript-template-for-aspnet-mvc, which as the URL suggests is a GitHub project developing a Visual Studio project template for using TypeScript with MVC 4.

■ **Tip** As this book went to press, Chris Sells had just posted online some new sample project templates for Windows 8 applications built with TypeScript. Find them at https://www.sellsbrothers.com/Posts/Details/12724

In Your MSBuild File

The key to your projects correctly integrating TypeScript into their compilation is in adding a call to the TypeScript compiler before the actual build. You know from Chapter 1 that if it is installed, the Web Essentials extension compiles our TypeScript files for us on the fly, but when we build the project, Visual Studio will compile them again based on a different set of criteria from those expressed in the Web Essentials Options dialog box (shown earlier in Figure 3-1). The build criteria can be found in the MSBuild file for your project—typically the `.jsproj`, `.csproj`, or `.vbproj` file in the root directory of your project.

Let's have a look at the MSBuild file for a TypeScript project and see what it does:

1. Create a new HTML Application with TypeScript project in VS called CompilerTest.
2. Add a second TypeScript file to the project. Call it `second.ts`.
3. In the Solution Explorer window, right-click the project name and then select Unload Project.
4. Right-click the project name again and click Edit CompilerTest.csproj file.
5. To reload the project back into Visual Studio after you've finished looking at this, right-click the project name in the Solution Explorer window and then select Reload Project.

The relevant parts of the file are as follows.

The TypeScript Files and Their Dependencies

The project's MSBuild file lists each TypeScript file within a `<TypeScriptCompile>` element. Each of these is a child of an `<ItemGroup>` element, although there may not be a 1:1 correspondence between the two. Each JavaScript file is listed within a `<Content>` element, indicating that the JavaScript file must be included when the project is published. Each `<Content>` element also has a child `<DependentUpon>` element documenting the relationship between the two.

```
<ItemGroup>
  <Content Include="app.js">
    <DependentUpon>app.ts</DependentUpon>
  </Content>
```

```
  <Content Include="second.js">
    <DependentUpon>second.ts</DependentUpon>
  </Content>
  <TypeScriptCompile Include="second.ts" />
  <TypeScriptCompile Include="app.ts" />
</ItemGroup>
```

Note that the second `<TypeScriptCompile>` element may be wrapped in its own `<ItemGroup>` parent element.

Compiler Instructions

Toward the end of the file, you'll find the instructions for TypeScript compilation within a `<Target>` element.

```
<Target Name="BeforeBuild">
```

In it, MSBuild is told to output the complete command sent to the `tsc` executable file and then execute the command. If the application is in debug mode, MSBuild adds the `--sourcemap` option to the compiler call so the TypeScript file can be stepped through rather than having to debug the JavaScript and then work backward to the issue in the TypeScript. MSBuild derives the file list sent to `tsc` by enumerating all the files listed in `<TypeScriptCompile>` elements. As these instructions target the `BeforeBuild` step, all TypeScript compilation will occur before any C# or VB.NET compilation.

```
<PropertyGroup Condition="'$(Configuration)' == 'Debug'">
  <TypeScriptSourceMap> --sourcemap</TypeScriptSourceMap>
</PropertyGroup>

<Target Name="BeforeBuild">
  <Message Text="Compiling TypeScript files" />
  <Message Text="Executing tsc$(TypeScriptSourceMap)
    @(TypeScriptCompile ->'&amp;quot;%(fullpath)&amp;quot;', ' ')" />
  <Exec Command="tsc$(TypeScriptSourceMap)
    @(TypeScriptCompile ->'&amp;quot;%(fullpath)&amp;quot;', ' ')" />
</Target>
```

Note that the `<Exec>` element assumes that the directory for the `tsc` compiler has been added to your `PATH`. If it hasn't for some reason, you'll need to add the full path to the `exec` command (and then remember to update it whenever you install the latest version of TypeScript).

```
<Target Name="BeforeBuild">
  <Exec Command="&amp;quot;$(PROGRAMFILES)\Microsoft SDKs\TypeScript\<VERSION>\
tsc&amp;quot; @(TypeScriptCompile ->'&amp;quot;%(fullpath)&amp;quot;', ' ')" />
</Target>
```

Tweaking Your Existing Project File

If you are adding TypeScript files into an existing project, you need to know that Visual Studio will add information to your MSBuild file listing the TypeScript/JavaScript files and their dependencies, but it will not add the compiler instructions for you. You'll need to do this yourself:

1. Add the `<Exec>` element into your project's `BeforeBuild` step.
2. Add the `<TypeScriptSourceMap>` element into your project's debug build `PropertyGroup` element.

If your project has neither a `BeforeBuild` step nor a debug build `PropertyGroup`, add them in their entirety as given in the previous section.

In fact, if you're in the project file anyway, you might find it useful to refactor it slightly and tell it to compile every TypeScript file in the project.

```
<ItemGroup>
  <AvailableItemName Include="TypeScriptCompile" />
</ItemGroup>
<ItemGroup>
  <TypeScriptCompile Include="$(ProjectDir)\**\*.ts" />
</ItemGroup>
```

This preemptive inclusion of every `.ts` file into the `TypeScriptCompile` group replaces all the `<TypeScriptCompile>` elements with this single one. You'll want to remove the others if you go for this approach.

You can also add any of the options seen earlier in the chapter to the compiler call. For example, to generate ECMAScript 5 code, to include your comments in the JavaScript, and so forth, the `<Exec>` element is the place to do it:

```
<Exec Command="tsc$(TypeScriptSourceMap) -c --Target ES5 @(TypeScriptCompile
->'&amp;quot;%(fullpath)&amp;quot;', ' ')" />
```

For non-TypeScript-specific projects, you can use the `<None>` item group to list the `.ts` files but not include them in releases:

```
<ItemGroup>
  <Content Include="app.js">
    <DependentUpon>app.ts</DependentUpon>
  </Content>
  <Content Include="second.js">
    <DependentUpon>second.ts</DependentUpon>
  </Content>
  <None Include="second.ts" />
  <None Include="app.ts" />
</ItemGroup>
```

Using this approach requires you to include the `AvailableItemName` and preemptive `TypeScriptCompile` elements shown in the earlier code.

The Web Site Project

The preceding approach works for all Visual Studio projects except for one: the Web Site project. With respect to TypeScript, there are two main differences between Web Site projects and Web Application projects.

The first is that TypeScript files aren't recognized file types for Web Site projects. As Figure 3-2 shows, the Add New Item dialog box for Web Site projects has no TypeScript option.

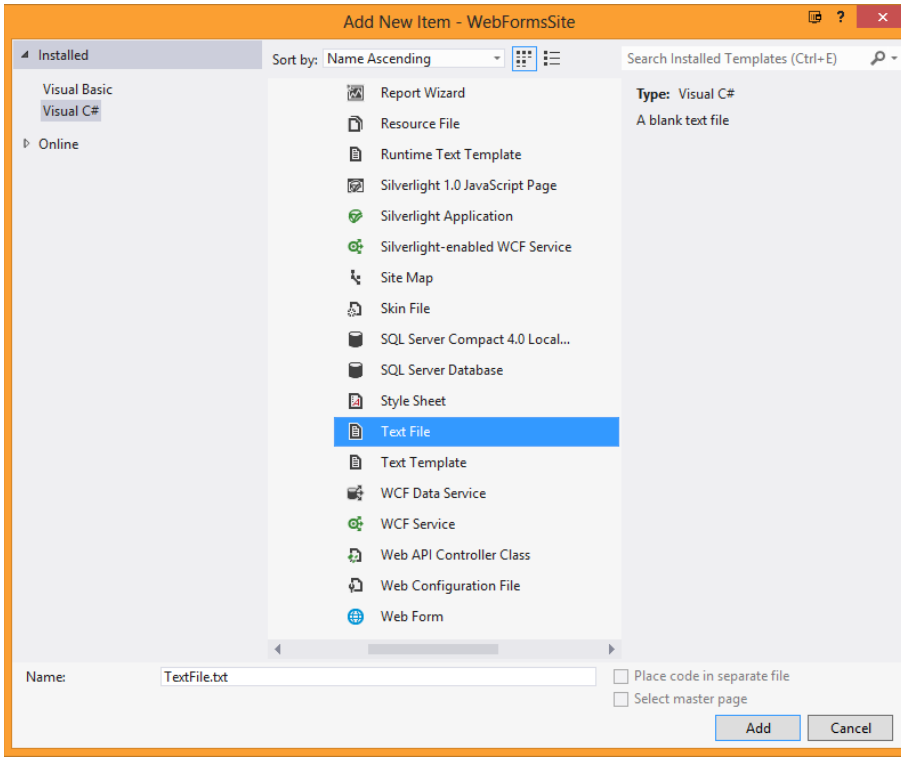

Figure 3-2. *A Web Site project's Add New Item dialog box has no TypeScript File option*

To add a TypeScript file to a Web Site project, simply add a new text file but give it a `.ts` extension. Visual Studio picks up the `.ts` extension and from then on treats it as a TypeScript file (visually at least). The fact that Visual Studio doesn't think of it as a valid file type for a Web Site project doesn't matter, because there's no MSBuild file to update manually anyway.

In fact, this is the main problem with TypeScript in Web Sites projects. They have no MSBuild (`.proj`) file. Instead, MSBuild is passed the `.sln` file containing the root directory of the site and from there it generates the `.proj` file on the fly. All of which means, we can't add the all-important compiler instructions to the projects' build instructions in the same way we could with other projects.

Fortunately, there are three workarounds:

- If possible, install the Web Essentials extension for Visual Studio and make sure the Compile TypeScript on Save option (see Figure 3-1 earlier) is set to True.
- Write a batch file that calls the TypeScript compiler first and then MSBuild itself on the solution file. For example:

```
@echo off
setLocal EnableDelayedExpansion
for /f %%a in ('dir /b WebSiteFolder/*.ts') do ( set str=!str!
"%%a" )
tsc !str!
msbuild WebSiteProject.sln
```

- You can get MSBuild to generate a hard copy of the `.proj` file it creates on the fly when building the project. Open a Visual Studio command prompt window and browse to the folder containing the `.sln` file for your project. Then run the two following commands:

```
> set msbuildemitsolution = 1
> msbuild <ProjectName>.sln
```

You'll find that two `.metaproj` files have been generated in the same folder as the `.sln` file: one for the solution as a whole and the other for the Web Site project. You may then be able to add the `BeforeBuild` step and TypeScript file list to the solution's `.metaproj` file and call MSBuild on that.

In Your Web Sites

After your project is compiling your TypeScript to your preference, you still have a couple of points to consider if you're a web developer.

You can't embed TypeScript within a web page. TypeScript lives inside its own files. If your current project has web pages with `<script>` blocks containing actual JavaScript rather than references to files, you'll need to ask yourself whether there is any way to refactor that JavaScript out to an external file. Chapter 1 provides a very simple example of doing this, so all the event wire-up and scripting is done after the page has loaded. If it's not possible (perhaps your server-side code injects script into the page), remember that your TypeScript code will still be able to interact with any functions or variables on the page as long as you include ambient declarations for them in your `.ts` file.

■ **Note** There's nearly always a way to separate a web page from its JavaScript. Try googling Unobtrusive JavaScript for hints and tips.

If you're using the Bundling feature in ASP.NET 4.5's System.Web.Optimization feature, you may want to use that to gather your compiled JavaScript (not the TypeScript files) into one file rather than using the compiler's own `--out` option. Indeed, this approach offers a lot more flexibility if you want to compile several libraries into their own files.

```
BundleTable.Bundles.Add(
   new ScriptBundle("~/ts/AllScript.js")
         .IncludeDirectory("~/ts/", "*.js", true));
```

Remember, however, to reference the bundled JavaScript file in your pages and not the individual JavaScript files.

■ **Note** You can also find a bundle transform for ASP.NET that takes a TypeScript bundle and compiles it to JavaScript at https://github.com/wouterdevinck/TypeScript-BundleTransform/.

In the Development Life Cycle

Before wrapping up, it's worth saying that whatever you can do in JavaScript, you can probably also do in TypeScript, and some more on top of that. Declaration files for popular JavaScript libraries are still being written, and the TypeScript project itself is still a way from wrapping up a v1.0 release. By that time, it would not be surprising to see a complete and cogent application development life cycle based solely in HTML and TypeScript. Consider the following:

- The UI design phase is language-neutral anyway.
- The Class and Module features in TypeScript allow for more-accurate code design, development, and maintenance already.
- The TypeScript debugging story is already getting better through the generation of source map files that allow step-through debugging in Visual Studio.
- Unit testing in TypeScript is already possible with a QUnit declaration file in the works. Other efforts also show that unit testing is possible using Jasmine and Chutzpah. (See the appendix for links.)
- Including TypeScript files in source control is the same as for any other type of file. Just remember not to check in the JavaScript files that the compiler generates from them. After all, you don't check in your project's bin and obj folders, do you? Do you?

Summary

In this chapter, you've looked at the more practical issues of TypeScript development. To wit:

- Compiler options and what they do
- Where to source and how to write your own declaration files
- How to incorporate TypeScript compilation into your own Visual Studio projects

I hope that you've enjoyed this whistle-stop tour and that you'll start using TypeScript in earnest. Now really is the best of times to start.

APPENDIX A

Webliography

In this appendix, you'll find all the URLs mentioned in the book plus some more for quick reference.

Core Links

- TypeScript home

  http://typescriptlang.org

- TypeScript project on CodePlex

  http://typescript.codeplex.com

- TypeScript team blog

  http://blogs.msdn.com/b/typescript/

- TypeScript sample applications

  http://typescriptlang.org/samples

- TypeScript installer

  www.microsoft.com/en-gb/download/details.aspx?id=34790

- ECMAScript 6 (Harmony) specification

  http://wiki.ecmascript.org/doku.php?id=&idx=harmony

- Visual Studio Gallery for Web Essentials 2012 extension

  http://visualstudiogallery.msdn.microsoft.com/

- TypeScript support for Sublime Text, vi, and eMacs

  http://blogs.msdn.com/b/interoperability/archive/2012/10/01/sublime-text-vi-emacs-typescript-enabled.aspx

GitHub Projects

- Community repository for TypeScript declaration files

  https://github.com/borisyankov/DefinitelyTyped

- TypeScript extension for WebMatrix

  http://extensions.webmatrix.com/packages/TypeScript4WebMatrix/

- Github repository for TypeScript extension for WebMatrix

  http://macawnl.github.com/TypeScript4WebMatrix/

- Autocompile TypeScript files on save for Sublime Text

  https://github.com/alexnj/SublimeOnSaveBuild

- Workaround for `node.js` while the TypeScript compiler is not exposed as an explicit public API for module access

  https://github.com/eknkc/typescript-require

- `backbone.js` written in TypeScript

  https://github.com/jbaldwin/backbone.ts

- ASP.NET MVC 4 project template with TypeScript support

  https://github.com/devcurry/typescript-template-for-aspnet-mvc

- TypeScript-aware bundle transform

  https://github.com/wouterdevinck/TypeScript-BundleTransform/

Notable Bugs and Discussions

- TypeScript compiler is not exposed as an explicit public API for module access in `node.js` package

  http://typescript.codeplex.com/workitem/97

- An official TypeScript declaration file community site

  http://typescript.codeplex.com/workitem/191

  http://typescript.codeplex.com/discussions/398580

- How best to reconcile a declaration file with the particular version of the JavaScript library the file is describing

  http://typescript.codeplex.com/workitem/267

- Native Compiler support for more file APIs in addition to node and Windows Script Host

  http://typescript.codeplex.com/discussions/404060

- Compiler doesn't generate a `.js` file if `.ts` file has no compilable code

  http://typescript.codeplex.com/workitem/284

- Compiler loses performance working with large files

  http://typescript.codeplex.com/workitem/265

- Will TypeScript support destructuring / more of the ECMAScript Harmony specification?

  http://typeScript.codeplex.com/workitem/15

- Request for Visual Studio to support the 'Extract Interface' refactoring

  http://typescript.codeplex.com/discussions/400724

Miscellaneous

- A quick guide to JavaScript

  http://www.w3schools.com/js/default.asp

- A collection of JavaScript gotchas

  http://www.codeproject.com/Articles/182416/A-Collection-of-JavaScript-Gotchas

- Quoted article in Chapter 1

  http://readwrite.com/2012/10/03/microsofts-typescript-fills-a-long-standing-void-in-javascript

- List of languages like TypeScript, Dart, and CoffeeScript that compile to JavaScript

  https://github.com/jashkenas/coffee-script/wiki/List-of-languages-that-compile-to-JS

- The Dart language home page

  http://www.dartlang.org

- The CoffeeScript language home page

  http://coffeescript.org

- Mads Kristensen's blog (creator of Web Essentials plugin for Visual Studio)

  http://madskristensen.net

- TypeScript questions on Stack Overflow

  http://stackoverflow.com/questions/tagged/typescript

- TypeScript Templates for Windows 8 Applications

  http://www.sellsbrothers.com/posts/Details/12724

Index

T, U, V

W, X, Y, Z

TypeScript Revealed

Dan Maharry

Apress®

TypeScript Revealed

ISBN-13 (pbk): 978-1-4302-5725-7

ISBN-13 (electronic): 978-1-4302-5726-4

President and Publisher: Paul Manning
Lead Editor: Jonathan Hassell
Technical Reviewer: Todd Meister
Editorial Board: Steve Anglin, Mark Beckner, Ewan Buckingham, Gary Cornell, Louise Corrigan, Morgan Ertel, Jonathan Gennick, Jonathan Hassell, Robert Hutchinson, Michelle Lowman, James Markham, Matthew Moodie, Jeff Olson, Jeffrey Pepper, Douglas Pundick, Ben Renow-Clarke, Dominic Shakeshaft, Gwenan Spearing, Matt Wade, Tom Welsh
Coordinating Editor: Mark Powers
Copy Editor: Sharon Wilkey
Compositor: SPi Global
Indexer: SPi Global
Artist: SPi Global
Cover Designer: Anna Ishchenko

Distributed to the book trade worldwide by Springer Science+Business Media New York, 233 Spring Street, 6th Floor, New York, NY 10013. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail rights@apress.com, or visit www.apress.com.

Apress and friends of ED books may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Special Bulk Sales–eBook Licensing web page at www.apress.com/bulk-sales.

Any source code or other supplementary materials referenced by the author in this text is available to readers at www.apress.com/9781430257257. For detailed information about how to locate your book's source code, go to www.apress.com/source-code/.

For Jane and Evie, who make me smile and laugh

Contents

About the Author

Dan Maharry is an experienced technical author with over a dozen books to his name and many more as technical reviewer and editor for a variety of publishers including Wrox Press, Apress, Microsoft Press, and O'Reilly Media.

Dan contributes code and documentation to various open source projects and tries to blog and to speak from time to time to user groups about cool stuff.

He's also a .NET developer with past stints for the dotCoop TLD registry and various web development houses under his belt.

He listens to a lot of new music as he does the above.

You can find him online at `http://hmobius.com`. His blog is available at `http://blog.hmobius.com`.

About the Technical Reviewer

Todd Meister has been working in the IT industry for more than 15 years. He's been a technical editor for over 75 titles, ranging from SQL Server to the .NET Framework. Besides technical editing, he is the senior IT architect at Ball State University in Muncie, Indiana. He lives in central Indiana with his wife, Kimberly, and their five excellent children.

Acknowledgments

Many thanks to Jon and Mark at Apress for inviting me to write this book, and thanks also to Anders and the TypeScript team for making the fruits of Project Strada available for us to consume.

And, of course, thanks to the two ladies in my life who keep me sane and drag me away from the screen into reality. To Jane and Evie, I love you both.