THE EXPERT'S VOICE® IN TYPESCRIPT

Pro TypeScript

Application-Scale JavaScript Development

Steve Fenton

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

Atwood's Law: any application that can be written in JavaScript will eventually be written in JavaScript.

—Jeff Atwood

TypeScript is a language created by Microsoft and released under an open-source Apache 2.0 License (2004). The language is focused on making the development of JavaScript programs scale to many thousands of lines of code. The language attacks the large-scale JavaScript programming problem by offering better design-time tooling, compile-time checking, and dynamic module loading at runtime.

As you might expect from a language created by Microsoft, there is excellent support for TypeScript within Visual Studio, but other development tools have also added support for the language, including WebStorm, Eclipse, Sublime Text, Vi, IntelliJ, and Emacs among others. The widespread support from these tools as well as the permissive open-source license makes TypeScript a viable option outside of the traditional Microsoft ecosystem.

The TypeScript language is a typed superset of JavaScript, which is compiled to plain JavaScript. This makes programs written in TypeScript highly portable as they can run on almost any machine—in web browsers, on web servers, and even in native applications on operating systems that expose a JavaScript API, such as WinJS on Windows 8 or the Web APIs on Firefox OS.

The language features found in TypeScript can be divided into three categories based on their relationship to JavaScript (see Figure 1). The first two sets are related to versions of the ECMA-262 ECMAScript Language Specification, which is the official specification for JavaScript. The ECMAScript 5 specification forms the basis of TypeScript and supplies the largest number of features in the language. The ECMAScript 6 specification adds modules for code organization and class-based object orientation, and TypeScript has included these since its release in October 2012. The third and final set of language features includes items that are not planned to become part of the ECMAScript standard, such as generics and type annotations. All of the TypeScript features can be converted into valid ECMAScript 5 and most of the features can be converted into the ancient ECMAScript 3 standard if required.

Figure 1. *TypeScript language feature sources*

Because TypeScript is such a close relative of JavaScript, you can consume the myriad existing libraries and frameworks written in JavaScript. Angular, Backbone, Bootstrap, Durandal, jQuery, Knockout, Modernizr, PhoneGap, Prototype, Raphael, Underscore, and many more are all usable in TypeScript programs. Correspondingly, once your TypeScript program has been compiled it can be consumed from any JavaScript code.

TypeScript's similarity to JavaScript is beneficial if you already have experience with JavaScript or other C-like languages. The similarity also aids the debugging process as the generated JavaScript correlates closely to the original TypeScript code.

If you still need to be convinced about using TypeScript or need help convincing others, I summarize the benefits of the language as well as the problems it can solve in the following. I also include an introduction to the components of TypeScript and some of the alternatives. If you would rather get started with the language straight away, you can skip straight to Chapter 1.

Who This Book Is For

This book is for programmers and architects working on large-scale JavaScript applications, either running in a browser, on a server, or on an operating system that exposes a JavaScript API. Previous experience with JavaScript or another C-like language is useful when reading this book, as well as a working knowledge of object orientation and design patterns.

Structure

This book is organized into nine chapters and four appendices.

Chapter 1: TypeScript Language Features: describes the language features in detail, from simple type annotations to important structural elements, with stand-alone examples of how to use each one.

Chapter 2: The Type System: explains the details of working within TypeScript's structural type system and describes the details on type erasure, type inference, and ambient declarations.

Chapter 3: Object Orientation in TypeScript: introduces the important elements of object orientation and contains examples of design patterns and SOLID principles in TypeScript. This chapter also introduces the concept of mixins with practical examples.

Chapter 4: Understanding the Runtime: describes the impact of scope, callbacks, events, and extensions on your program.

Chapter 5: Running TypeScript in a Browser: a thorough walk-through including working with the Document Object Model, AJAX, session and local storage, IndexedDB, geolocation, hardware sensors, and web workers as well as information on packaging your program for the web.

Chapter 6: Running TypeScript on a Server: an explanation of running programs on a JavaScript server with examples for Node and a basic end-to-end application example written in Express and Mongoose.

Chapter 7: Exceptions, Memory, and Performance: describes exceptions and exception handling with information on memory management and garbage collection. Includes a simple performance testing utility to exercise and measure your program.

Chapter 8: Using JavaScript Libraries: explains how to consume any of the millions of JavaScript libraries from within your TypeScript program, including information on how to create your own type definitions and convert your JavaScript program to TypeScript.

Chapter 9: Automated Testing: a walk-through of automated testing in your TypeScript program with examples written using the Jasmine framework.

Appendix 1: JavaScript Quick Reference: an introduction to the essential JavaScript features for anyone who needs to brush up on their JavaScript before diving into TypeScript.

Appendix 2: TypeScript Compiler: explains how to use the compiler on the command line and describes many of the flags you can pass to customize your build.

Appendix 3: Bitwise Flags: dives into the details of bitwise flags including the low-level details of how they work as well as examples using TypeScript enumerations.

Appendix 4: Coding Katas: introduces the concept of coding katas and provides an example for you to try, along with techniques you can use to make katas more effective.

The TypeScript Components

TypeScript is made up of three distinct but complementary parts, which are shown in Figure 2.

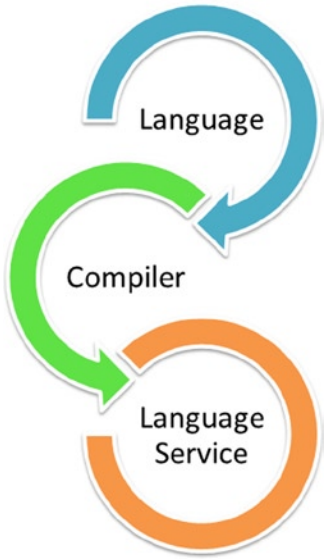

Figure 2. *The TypeScript components*

The language consists of the new syntax, keywords, and type annotations. As a programmer, the language will be the component you will become most familiar with. Understanding how to supply type information is an important foundation for the other components because the compiler and language service are most effective when they understand the complex structures you use within your program.

The compiler performs the type erasure and code transformations that convert your TypeScript code into JavaScript. It will emit warnings and errors if it detects problems and can perform additional tasks such as combining the output into a single file, generating source maps, and more.

The language service provides type information that can be used by development tools to supply autocompletion, type hinting, refactoring options, and other creative features based on the type information that has been gathered from your program.

Compile or Transpile?

The term *transpiling* has been around since the last century, but there is some confusion about its meaning. In particular, there has been some confusion between the terms *compilation* and *transpilation*. Compilation describes the process of taking source code written in one language and converting it into another language. Transpilation is a specific kind of compilation and describes the process of taking source code written in one language and transforming it into another language *with a similar level of abstraction*. So you might compile a high-level language into an assembly language, but you would transpile TypeScript to JavaScript as they are similarly abstracted.

Other common examples of transpilation include C++ to C, CoffeeScript to JavaScript, Dart to JavaScript, and PHP to C++.

Which Problems Does TypeScript Solve?

Since its first beta release in 1995, JavaScript (or LiveScript as it was known at the time it was released) has spread like wildfire. Nearly every computer in the world has a JavaScript interpreter installed. Although it is perceived as a browser-based scripting language, JavaScript has been running on web servers since its inception, supported on Netscape Enterprise Server, IIS (since 1996), and recently on Node. JavaScript can even be used to write native applications on operating systems such as Windows 8 and Firefox OS.

Despite its popularity, it hasn't received much respect from developers—possibly because it contains many snares and traps that can entangle a large program much like the tar pit pulling the mammoth to its death, as described by Fred Brooks (1975). If you are a professional programmer working with large applications written in JavaScript, you will almost certainly have rubbed up against problems once your program chalked up a few thousand lines. You may have experienced naming conflicts, substandard programming tools, complex modularization, unfamiliar prototypal inheritance that makes it hard to re-use common design patterns easily, and difficulty keeping a readable and maintainable code base. These are the problems that TypeScript solves.

Because JavaScript has a C-like syntax, it looks familiar to a great many programmers. This is one of JavaScript's key strengths, but it is also the cause of a number of surprises, especially in the following areas:

- Prototypal inheritance
- Equality and type juggling
- Management of modules
- Scope
- Lack of types

Typescript solves or eases these problems in a number of ways. Each of these topics is discussed in this introduction.

Prototypal Inheritance

Prototype-based programming is a style of object-oriented programming that is mainly found in interpreted dynamic languages. It was first used in a language called Self, created by David Ungar and Randall Smith in 1986, but it has been used in a selection of languages since then. Of these prototypal languages, JavaScript is by far the most widely known, although this has done little to bring prototypal inheritance into the mainstream. Despite its validity, prototype-based programming is somewhat esoteric; class-based object orientation is far more commonplace and will be familiar to most programmers.

TypeScript solves this problem by adding classes, modules, and interfaces. This allows programmers to transfer their existing knowledge of objects and code structure from other languages, including implementing interfaces, inheritance, and code organization. Classes and modules are an early preview of JavaScript proposals and because TypeScript can compile to earlier versions of JavaScript it allows you to use these features independent of support for the ECMAScript 6 specification. All of these features are described in detail in Chapter 1.

Equality and Type Juggling

JavaScript has always supported dynamically typed variables and as a result it expends effort at runtime working out types and coercing them into other types on the fly to make statements work that in a statically typed language would cause an error.

The most common type coercions involve strings, numbers, and Boolean target types. Whenever you attempt to concatenate a value with a string, the value will be converted to a string, if you perform a mathematical operation an attempt will be made to turn the value into a number and if you use any value in a logical operation there are special rules that determine whether the result will be true or false. When an automatic type conversion occurs it is commonly referred to as *type juggling*.

In some cases, type juggling can be a useful feature, in particular in creating shorthand logical expressions. In other cases, type juggling hides an accidental use of different types and causes unintended behavior as discussed in Chapter 1. A common JavaScript example is shown in Listing 1.

Listing 1. Type juggling

```
var num = 1;
var str = '0';

// result is '10' not 1
var strTen = num + str;

// result is 20
var result = strTen * 2;
```

TypeScript gracefully solves this problem by introducing type checking, which can provide warnings at design and compile time to pick up potential unintended juggling. Even in cases where it allows implicit type coercion, the result will be assigned the correct type. This prevents dangerous assumptions from going undetected. This feature is covered in detail in Chapter 2.

Management of Modules

If you have worked with JavaScript, it is likely that you will have come across a dependency problem. Some of the common problems include the following:

- Forgetting to add a script tag to a web page
- Adding scripts to a web page in the wrong order
- Finding out you have added scripts that aren't actually used

There is also a series of issues you may have come across if you are using tools to combine your scripts into a single file to reduce network requests or if you minify your scripts to lower bandwidth usage.

- Combining scripts into a single script in the wrong order
- Finding out that your chosen minification tool doesn't understand single-line comments
- Trying to debug combined and minified scripts

You may have already solved some of these issues using module loading as the pattern is gaining traction in the JavaScript community. However, TypeScript makes module loaders the normal way of working and allows your modules to be compiled to suit the two most prevalent module loading styles without requiring changes to your code. The details of module loading in web browsers are covered in Chapter 5 and on the server in Chapter 6.

Scope

In most modern C-like languages, the curly braces create a new context for scope. A variable declared inside a set of curly braces cannot be seen outside of that block. JavaScript bucks this trend by being functionally scoped, which means blocks defined by curly braces have no effect on scope. Instead, variables are scoped to the function they are declared in, or the global scope if they are not declared within a function. There can be further complications caused by the accidental omission of the `var` keyword within a function promoting the variable to the global scope. More complications are caused by *variable hoisting* resulting in all variables within a function behaving as if they were declared at the top of the function.

Despite some tricky surprises with scope, JavaScript does provide a powerful mechanism that wraps the current lexical scope around a function declaration to keep values to hand when the function is later executed. These closures are one of the most powerful features in JavaScript. There are also plans to add block level scope in the next version of JavaScript by using the `let` keyword, rather than the `var` keyword.

TypeScript eases scope problems by warning you about implicit global variables, provided you avoid adding variables to the global scope.

Lack of Types

The problem with JavaScript isn't that it has no types because each variable does have a type; it is just that the type can be changed by each assignment. A variable may start off as a string, but an assignment can change it to a number, an object, or even a function. The real problem here is that the development tools cannot be improved beyond a reasonable guess about the type of a variable. If the development tools don't know the types, the autocompletion and type hinting is often too general to be useful.

By formalizing type information, TypeScript allows development tools to supply specific contextual help that otherwise would not be possible.

Which Problems Are Not Solved

> *TypeScript is not a crutch any more than JSLint is a crutch. It doesn't hide JavaScript (as CoffeeScript tends to do).*
>
> — Ward Bell

TypeScript remains largely faithful to JavaScript. The TypeScript specification adds many language features, but doesn't attempt to change the ultimate style and behavior of the JavaScript language. It is just as important for TypeScript programmers to embrace the idiosyncrasies of the runtime as it is for JavaScript programmers. The aim of the TypeScript language is to make large-scale JavaScript programs manageable and maintainable. No attempt has been made to twist JavaScript development into the style of C#, Java, Python, or any other language (although it has taken inspiration from many languages).

Prerequisites

To benefit from the features of TypeScript, you'll need access to an integrated development environment that supports the syntax and compiler. The examples in this book were written using Visual Studio 2013, but you can use WebStorm/PHPStorm, Eclipse, Sublime Text, Vi, Emacs, or any other development tools that support the language; you can even try many of the simpler examples on the TypeScript Playground provided by Microsoft (2012).

From the Visual Studio 2013 Spring Update (Update 2), TypeScript is a first class language in Visual Studio. If you are using an older version you can download and install the TypeScript extension from Microsoft (2013). Although the examples in this book are shown in Visual Studio, you can use any of the development tools that were listed at the very start of this introduction.

It is also worth downloading and installing NodeJS (which is required to follow the example in Chapter 6) as it will allow you to access the Node Package Manager and the thousands of modules and utilities available through it. For example, you can use grunt-ts to watch your TypeScript files and compile them automatically each time you change them if your development tools don't do this for you.

Node is free and can be downloaded for multiple platforms from

http://nodejs.org/

TypeScript Alternatives

TypeScript is not the only alternative to writing to plain JavaScript. CoffeeScript is a popular alternative with a terse syntax that compiles to sensible JavaScript code. CoffeeScript doesn't offer many of the additional features that TypeScript offers, such as static type checking. It is also a very different language to JavaScript, which means you need to translate snippets of code you find online into CoffeeScript to use them. You can find out more about CoffeeScript on the official website

http://coffeescript.org/

Another alternative is Google's Dart language. Dart has much more in common with TypeScript. It is class-based, object oriented and offers optional types that can be checked by a static checker. Dart was originally conceived as a replacement for JavaScript, which could be compiled to JavaScript to provide wide support in the short term. It seems unlikely at this stage that Dart will get the kind of browser support that JavaScript has won, so the compile-to-JavaScript mechanism will remain core to Dart's future in the web browser. You can read about Dart on the official website for the language

https://www.dartlang.org/

There are also converters that will compile from most languages to JavaScript, including C#, Ruby, Java, and Haskell. These may appeal to programmers who are uncomfortable stepping outside of their primary programming language.

It is also worth bearing in mind that for small applications and web page widgets, you can defer the decision and write the code in plain JavaScript. With TypeScript in particular, there is no penalty for starting in JavaScript as you can simply paste your JavaScript code into a TypeScript file later on to make the switch.

Summary

TypeScript is an application-scale programming language that provides early access to proposed new JavaScript features and powerful additional features like static type checking. You can write TypeScript programs to run in web browsers or on servers and you can re-use code between browser and server applications.

TypeScript solves a number of problems in JavaScript, but respects the patterns and implementation of the underlying JavaScript language, for example, the ability to have dynamic types and the rules on scope.

You can use many integrated development environments with TypeScript, with several providing first class support including type checking and autocompletion that will improve your productivity and help eliminate mistakes at design time.

Key Points

- TypeScript is a language, a compiler, and a language service.
- You can paste existing JavaScript into your TypeScript program.
- Compiling from TypeScript to JavaScript is known specifically as transpiling.
- TypeScript is not the only alternative way of writing JavaScript, but it bears the closest resemblance to JavaScript.

CHAPTER 1

TypeScript Language Features

> *What if we could strengthen JavaScript with the things that are missing for large scale application development, like static typing, classes [and] modules… that's what TypeScript is about.*
>
> —Anders Hejlsberg

TypeScript is a superset of JavaScript. That means that the TypeScript language includes the entire JavaScript language plus a collection of useful additional features. This is in contrast to the various subsets of JavaScript and the various lint tools that seek to reduce the available features to create a smaller language with fewer surprises. This chapter will introduce you to the extra language features, starting with simple type annotations and progressing to more advanced features and structural elements of TypeScript. This chapter doesn't cover the features included in the ECMAScript 5 language specification so if you need a refresher on JavaScript take a look at Appendix 1.

The important thing to remember is that all of the standard control structures found in JavaScript are immediately available within a TypeScript program. This includes:

- Control flows
- Data types
- Operators
- Subroutines

The basic building blocks of your program will come from JavaScript, including if statements, switch statements, loops, arithmetic, logical tests, and functions. This is one of the key strengths of TypeScript—it is based on a language (and a family of languages) that is already familiar to a vast and varied collection of programmers. JavaScript is thoroughly documented not only in the ECMA-262 specification, but also in books, on developer network portals, forums, and question-and-answer websites.

Each of the language features discussed in this chapter has short, self-contained code examples that put the feature in context. For the purposes of introducing and explaining features, the examples are short and to the point; this allows the chapter to be read end-to-end. However, this also means you can refer back to the chapter as a reference later on. Once you have read this chapter, you should know everything you will need to understand the more complex examples described throughout the rest of the book.

JavaScript Is Valid TypeScript

Before we find out more about the TypeScript syntax, it is worth stressing this important fact: All JavaScript is valid TypeScript, with just a small number of exceptions, which are explained below. You can take existing JavaScript code, add it to a TypeScript file, and all of the statements will be valid. There is a subtle difference between valid code and error-free code in TypeScript; because, although your code may work, the TypeScript compiler will warn you about any potential problems it has detected.

If you transfer a JavaScript listing into a TypeScript file you may receive errors or warnings even though the code is considered valid. A common example comes from the dynamic type system in JavaScript wherein it is perfectly acceptable to assign values of different types to the same variable during its lifetime. TypeScript detects these assignments and generates errors to warn you that the type of the variable has been changed by the assignment. Because this is a common cause of errors in a program, you can correct the error by creating separate variables, by performing a type assertion, or by making the variable dynamic. There is further information on type annotations later in this chapter, and the type system is discussed in detail in Chapter 2.

Unlike some compilers that will only create output where no compilation errors are detected, the TypeScript compiler will still attempt to generate sensible JavaScript code. The code shown in Listing 1-1 generates an error, but the JavaScript output is still produced. This is an admirable feature, but as always with compiler warnings and errors, you should correct the problem in your source code and get a clean compilation. If you routinely ignore warnings, your program will eventually exhibit unexpected behavior. In some cases, your listing may contain errors that are so severe the TypeScript compiler won't be able to generate the JavaScript output.

Listing 1-1. Using JavaScript's "with" statement

```
// Not using with
var radius = 4;
var area = Math.PI * radius * radius;

// Using with
var radius = 4;
with (Math) {
    var area = PI * radius * radius;
}
```

■ **Caution** The only exceptions to the "all JavaScript is valid TypeScript" rule are the `with` statement and vendor specific extensions, such as Mozilla's `const` keyword.

The JavaScript `with` statement in Listing 1-1 shows two examples of the same routine. Although the first calls `Math.PI` explicitly, the second uses a `with` statement, which adds the properties and functions of `Math` to the current scope. Statements nested inside the `with` statement can omit the `Math` prefix and call properties and functions directly, for example the `PI` property or the `floor` function.

At the end of the `with` statement, the original lexical scope is restored, so subsequent calls outside of the `with` block must use the `Math` prefix.

The `with` statement is not allowed in strict mode in ECMAScript 5 and in ECMAScript 6 classes and modules will be treated as being in strict mode by default. TypeScript treats `with` statements as an error and will treat all types within the `with` statement as dynamic types. This is due to the following:

- The fact it is disallowed in strict mode.
- The general opinion that the `with` statement is dangerous.
- The practical issues of determining the identifiers that are in scope at compile time.

So with these minor exceptions to the rule in mind, you can place any valid JavaScript into a TypeScript file and it will be valid TypeScript. As an example, here is the area calculation script transferred to a TypeScript file.

■ **Note** The ECMAScript 6 specification, also known as "ES6 Harmony," represents a substantial change to the JavaScript language. The specification is still under development at the time of writing.

In Listing 1-2, the statements are just plain JavaScript, but in TypeScript the variables `radius` and `area` will both benefit from type inference. Because `radius` is initialized with the value `4`, it can be inferred that the type of `radius` is `number`. With just a slight increase in effort, the result of multiplying `Math.PI`, which is known to be a `number`, with the `radius` variable that has been inferred to be a `number`, it is possible to infer the type of `area` is also a `number`.

Listing 1-2. Transferring JavaScript in to a TypeScript file

```
var radius = 4;
var area = Math.PI * radius * radius;
```

With type inference at work, assignments can be checked for type safety. Figure 1-1 shows how an unsafe assignment is detected when a string is assigned to the radius variable. There is a more detailed explanation of type inference in Chapter 2.

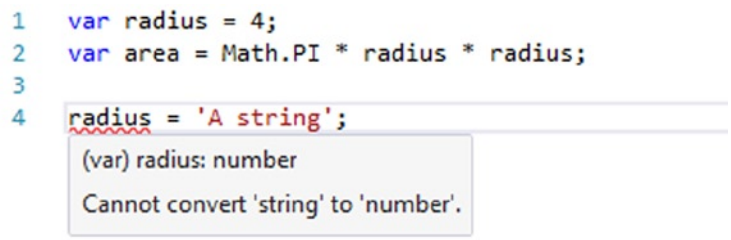

Figure 1-1. *Static type checking*

Variables

TypeScript variables must follow the JavaScript naming rules. The identifier used to name a variable must satisfy the following conditions.

The first character must be one of the following:

- an uppercase letter
- a lowercase letter
- an underscore
- a dollar sign
- a Unicode character from categories—*Uppercase letter* (Lu), *Lowercase letter* (Ll), *Title case letter* (Lt), *Modifier letter* (Lm), *Other letter* (Lo), or *Letter number* (Nl)

Subsequent characters follow the same rule and also allow the following:

- numeric digits
- a Unicode character from categories—*Non-spacing mark* (Mn), *Spacing combining mark* (Mc), *Decimal digit number* (Nd), or *Connector punctuation* (Pc)
- the Unicode characters U+200C (Zero Width Non-Joiner) and U+200D (Zero Width Joiner)

You can test a variable identifier for conformance to the naming rules using the JavaScript variable name validator by Mathias Bynens.

http://mothereff.in/js-variables

■ **Note** The availability of some of the more exotic characters can allow some interesting identifiers. You should consider whether this kind of variable name causes more problems than it solves. For example this is valid JavaScript:
`var ಠ_ಠ = 'Dignified';`

Variables are functionally scoped. If they are declared at the top level of your program they are available in the global scope. You should minimize the number of variables in the global scope to reduce the likelihood of naming collisions. Variables declared inside of functions, modules, or classes are available in the context they are declared as well as in nested contexts.

In JavaScript it is possible to create a global variable by declaring it without the `var` keyword. This is commonly done inadvertently when the `var` keyword is accidentally missed; it is rarely done deliberately. In a TypeScript program, this will cause an error, which prevents a whole category of hard to diagnose bugs in your code. Listing 1-3 shows a valid JavaScript function that contains an implicit global variable, for which TypeScript will generate a `"Could not find symbol"` error. This error can be corrected either by adding the `var` keyword, which would make the variable locally scoped to the `addNumbers` function, or by explicitly declaring a variable in the global scope.

Listing 1-3. Implicit global variable

```
function addNumbers(a, b) {
    // missing var keyword
    total = a + b;
    return total;
}
```

Types

TypeScript is optionally statically typed; this means that types are checked automatically to prevent accidental assignments of invalid values. It is possible to opt out of this by declaring dynamic variables. Static type checking reduces errors caused by accidental misuse of types. You can also create types to replace primitive types to prevent parameter ordering errors, as described in Chapter 2. Most important, static typing allows development tools to provide intelligent autocompletion.

Figure 1-2 shows autocompletion that is aware of the variable type, and supplies a relevant list of options. It also shows the extended information known about the properties and methods in the autocompletion list. Contextual autocompletion is useful enough for primitive types—but most reasonable integrated development environments can replicate simple inference even in a JavaScript file. However, in a program with a large number of custom types, modules, and classes, the deep type knowledge of the TypeScript Language Service means you will have sensible autocompletion throughout your entire program.

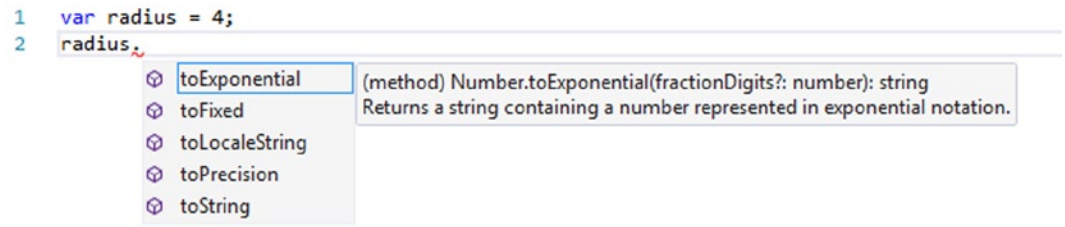

Figure 1-2. *TypeScript autocompletion*

Type Annotations

Although the TypeScript language service is expert at inferring types automatically, there are times when it isn't able to determine the type. There will also be times where you will wish to make a type explicit either for safety or readability. In all of these cases, you can use a type annotation to specify the type.

For a variable, the type annotation comes after the identifier and is preceded by a colon. Figure 1-3 shows the combinations that result in a typed variable. The most verbose style is to add a type annotation and assign the value. Although this is the style shown in many examples in this chapter, in practice this is the one you will use the least. The second variation shows a type annotation with no value assignment; the type annotation here is required because TypeScript cannot infer the type when there is no value present. The final example is just like plain JavaScript; a variable is declared and initialized on the same line. In TypeScript the type of the variable is inferred from the value assigned.

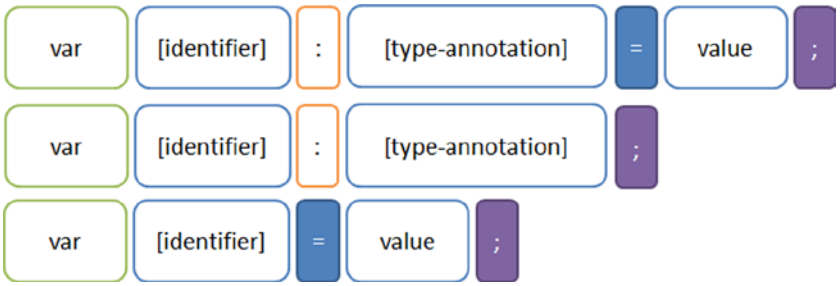

Figure 1-3. *Typed variable combinations*

To demonstrate type annotations in code, Listing 1-4 shows an example of a variable that has an explicit type annotation that marks the variable as a string. Primitive types are the simplest form of type annotation, but you are not restricted to such simple types.

Listing 1-4. Explicit type annotation

```
var name: string = 'Steve';
```

The type used to specify an annotation can be a primitive type, an array type, a function signature, or any complex structure you want to represent including the names of classes and interfaces you create. If you want to opt out of static type checking, you can use the special any type, which marks a variable's type as dynamic. No checks are made on dynamic types. Listing 1-5 shows a range of type annotations that cover some of these different scenarios.

Listing 1-5. Type annotations

```
// primitive type annotation
var name: string = 'Steve';
var heightInCentimeters: number = 182.88;
var isActive: boolean = true;

// array type annotation
var names: string[] = ['James', 'Nick', 'Rebecca', 'Lily'];

// function annotation with parameter type annotation and return type annotation
var sayHello: (name: string) => string;

// implementation of sayHello function
sayHello = function (name: string) {
    return 'Hello ' + name;
};

// object type annotation
var person: { name: string; heightInCentimeters: number; };

// Implementation of a person object
person = {
    name: 'Mark',
    heightInCentimeters: 183
};
```

■ **Note** Although many languages specify the type before the identifier, the placement of type annotations in TypeScript after the identifier helps to reinforce that the type annotation is optional. This style of type annotation is also inspired by *type theory*.

If a type annotation becomes too complex, you can create an interface to represent the type to simplify annotations. Listing 1-6 demonstrates how to simplify the type annotation for the `person` object, which was shown at the end of the previous example in Listing 1-5. This technique is especially useful if you intend to reuse the type as it provides a re-usable definition. Interfaces are not limited to describing object types; they are flexible enough to describe any structure you are likely to encounter. Interfaces are discussed in more detail later in this chapter.

Listing 1-6. Using an interface to simplify type annotations

```
interface Person {
    name: string;
    heightInCentimeters: number;
}

var person: Person = {
    name: 'Mark',
    heightInCentimeters: 183
}
```

Primitive Types

Although the primitive types seem limited in TypeScript, they directly represent the underlying JavaScript types and follow the standards set for those types. String variables can contain a sequence of UTF-16 code units. A Boolean type can be assigned only the true or false literals. Number variables can contain a double-precision 64-bit floating point value. There are no special types to represent integers or other specific variations on a number as it wouldn't be practical to perform static analysis to ensure all possible values assigned are valid.

The any type is exclusive to TypeScript and denotes a dynamic type. This type is used whenever TypeScript is unable to infer a type, or when you explicitly want to make a type dynamic. Using the any type is equivalent to opting out of type checking for the life of the variable.

■ **Caution** Before version 0.9 of TypeScript, the Boolean type was described using the `bool` keyword. There was a breaking change in the 0.9 TypeScript language specifications, which changed the keyword to `boolean`.

The type system also contains three types that are not intended to be used as type annotations but instead refer to the absence of values.

- The `undefined` type is the value of a variable that has not been assigned a value.
- The `null` type can be used to represent an intentional absence of an object value. For example, if you had a method that searched an array of objects to find a match, it could return `null` to indicate that no match was found.
- The `void` type is used only on function return types to represent functions that do not return a value or as a type argument for a generic class or function.

Arrays

TypeScript arrays have precise typing for their contents. To specify an array type, you simply add square brackets after the type name. This works for all types whether they are primitive or custom types. When you add an item to the array its type will be checked to ensure it is compatible. When you access elements in the array, you will get quality autocompletion because the type of each item is known. Listing 1-7 demonstrates each of these type checks.

Listing 1-7. Typed arrays

```
interface Monument {
    name: string;
    heightInMeters: number;
}

// The array is typed using the Monument interface
var monuments: Monument[] = [];

// Each item added to the array is checked for type compatibility
monuments.push({
    name: 'Statue of Liberty',
    heightInMeters: 46,
    location: 'USA'
});
```

```
monuments.push({
    name: 'Peter the Great',
    heightInMeters: 96
});

monuments.push({
    name: 'Angel of the North',
    heightInMeters: 20
});

function compareMonumentHeights(a: Monument, b: Monument) {
    if (a.heightInMeters > b.heightInMeters) {
        return -1;
    }
    if (a.heightInMeters < b.heightInMeters) {
        return 1;
    }
    return 0;
}

// The array.sort method expects a comparer that accepts two Monuments
var monumentsOrderedByHeight = monuments.sort(compareMonumentHeights);

// Get the first element from the array, which is the tallest
var tallestMonument = monumentsOrderedByHeight[0];

console.log(tallestMonument.name); // Peter the Great
```

There are some interesting observations to be made in Listing 1-7. When the `monuments` variable is declared, the type annotation for an array of `Monument` objects can either be the shorthand: `Monument[]` or the longhand: `Array<Monument>`—there is no difference in meaning between these two styles. Therefore, you should opt for whichever you feel is more readable. Note that the array is instantiated after the equals sign using the empty array literal (`[]`). You can also instantiate it with values, by adding them within the brackets, separated by commas.

The objects being added to the array using `monuments.push(...)` are not explicitly `Monument` objects. This is allowed because they are compatible with the `Monument` interface. This is even the case for the Statue of Liberty object, which has a `location` property that isn't part of the `Monument` interface. This is an example of structural typing, which is explained in more detail in Chapter 2.

The array is sorted using `monuments.sort(...)`, which takes in a function to compare values. When the comparison is numeric, the comparer function can simply return `a - b`, in other cases you can write custom code to perform the comparison and return a positive or negative number to be used for sorting (or a zero if the values are the same).

The elements in an array are accessed using an index. The index is zero based, so the first element in the `monumentsOrderedByHeight` array is `monumentsOrderedByHeight[0]`. When an element is accessed from the array, autocompletion is supplied for the `name` and `heightInMeters` properties. The `location` property that appears on the Statue of Liberty object is not supplied in the autocompletion list as it isn't part of the `Monument` interface.

To find out more about using arrays and loops, refer to Appendix 1.

Enumerations

Enumerations represent a collection of named elements that you can use to avoid littering your program with hard-coded values. By default, enumerations are zero based although you can change this by specifying the first value, in which case numbers will increment from the specified value. You can opt to specify values for all identifiers if you wish to. In Listing 1-8 the `VehicleType` enumeration can be used to describe vehicle types using well-named identifiers throughout your program. The value passed when an identifier name is specified is the number that represents the identifier, for example in Listing 1-8 the use of the `VehicleType.Lorry` identifier results in the number 5 being stored in the `type` variable. It is also possible to get the identifier name from the enumeration by treating the enumeration like an array.

Listing 1-8. Enumerations

```
enum VehicleType {
    PedalCycle,
    MotorCycle,
    Car,
    Van,
    Bus,
    Lorry
}

var type = VehicleType.Lorry;

var typeName = VehicleType[type]; // 'Lorry'
```

In TypeScript enumerations are open ended. This means all declarations with the same name inside a common root will contribute toward a single type. When defining an enumeration across multiple blocks, subsequent blocks after the first declaration must specify the numeric value to be used to continue the sequence, as shown in Listing 1-9. This is a useful technique for extending code from third parties, in ambient declarations and from the standard library.

Listing 1-9. Enumeration split across multiple blocks

```
enum BoxSize {
    Small,
    Medium
}

//...

enum BoxSize {
    Large = 2,
    XLarge,
    XXLarge
}
```

■ **Note** The term *common root* comes from graph theory. In TypeScript this term relates to a particular location in the tree of modules within your program. Whenever declarations are considered for merging, they must have the same fully qualified name, which means the same name at the same level in the tree.

Bit Flags

You can use an enumeration to define bit flags. Bit flags allow a series of items to be selected or deselected by switching individual bits in a sequence on and off. To ensure that each value in an enumeration relates to a single bit, the numbering must follow the binary sequence whereby each value is a power of two, e.g.,

```
1, 2, 4, 8, 16, 32, 64, 128, 256, 512, 1,024, 2,048, 4,096, and so on
```

Listing 1-10 shows an example of using an enumeration for bit flags. By default when you create a variable to store the state, all items are switched off. To switch on an option, it can simply be assigned to the variable. To switch on multiple items, items can be combined with the bitwise OR operator (|). Items remain switched on if you happen to include them multiple times using the bitwise OR operator. Bitwise Flags are explained in detail in Appendix 3.

Listing 1-10. Flags

```
enum DiscFlags {
        None = 0,
        Drive = 1,
        Influence = 2,
        Steadiness = 4,
        Conscientiousness = 8
}

// Using flags
var personality = DiscFlags.Drive | DiscFlags.Conscientiousness;

// Testing flags

// true
var hasD = (personality & DiscFlags.Drive) == DiscFlags.Drive;

// true
var hasI = (personality & DiscFlags.Influence) == DiscFlags.Influence;

// false
var hasS = (personality & DiscFlags.Steadiness) == DiscFlags.Steadiness;

// false
var hasC = (personality & DiscFlags.Conscientiousness) == DiscFlags.Conscientiousness;
```

Type Assertions

In cases in which TypeScript determines that an assignment is invalid, but you know that you are dealing with a special case, you can override the type using a type assertion. When you use a type assertion, you are guaranteeing that the assignment is valid in a scenario where the type system has found it not to be—so you need to be sure that you are right, otherwise your program may not work correctly. The type assertion precedes a statement, as shown in Listing 1-11. The `avenueRoad` variable is declared as a `House`, so a subsequent assignment to a variable declared as `Mansion` would fail. Because we know that the variable is compatible with the `Mansion` interface (it has all three properties required to satisfy the interface), the type assertion `<Mansion>` confirms this to the compiler.

Listing 1-11. Type assertions

```
interface House {
    bedrooms: number;
    bathrooms: number;
}

interface Mansion {
    bedrooms: number;
    bathrooms: number;
    butlers: number;
}

var avenueRoad: House = {
    bedrooms: 11,
    bathrooms: 10,
    butlers: 1
};

// Errors: Cannot convert House to Mansion
var mansion: Mansion = avenueRoad;

// Works
var mansion: Mansion = <Mansion>avenueRoad;
```

Although a type assertion overrides the type as far as the compiler is concerned, there are still checks performed when you assert a type. It is possible to force a type assertion, as shown in Listing 1-12, by adding an additional `<any>` type assertion between the actual type you want to use and the identifier of the variable.

Listing 1-12. Forced type assertions

```
var name: string = 'Avenue Road';

// Error: Cannot convert 'string' to 'number'
var bedrooms: number = <number> name;

// Works
var bedrooms: number = <number> <any> name;
```

Operators

All of the standard JavaScript operators are available within your TypeScript program. The JavaScript operators are described in Appendix 1. This section describes operators that have special significance within TypeScript because of type restrictions or because they affect types.

Increment and Decrement

The increment (`++`) and decrement (`--`) operators can only be applied to variables of type `any`, `number`, or `enum`. This is mainly used to increase index variables in a loop or to update counting variables in your program, as shown in Listing 1-13. In these cases you will typically be working with a `number` type. The operator works on variables with the `any` type, as no type checking is performed on these variables.

Listing 1-13. Increment and decrement

```
var counter = 0;

do {
    ++counter;
} while (counter < 10);

alert(counter); // 10
```

When incrementing or decrementing an enumeration, the number representation is updated. Listing 1-14 shows how incrementing the `size` variable results in the next element in the enumeration and decrementing the `size` variable results in the previous element in the enumeration. Beware when you use this method as you can increase and decrease the value beyond the bounds of the enumeration.

Listing 1-14. Increment and decrement of enumerations

```
enum Size {
    S,
    M,
    L,
    XL
}

var size = Size.S;
++size;
console.log(Size[size]); // M

var size = Size.XL;
--size;
console.log(Size[size]); // L

var size = Size.XL;
++size;
console.log(Size[size]); // undefined
```

Binary Operators

The operators in the following list are designed to work with two numbers. In TypeScript, it is valid to use the operators with variables of type `number` or `any`. Where you are using a variable with the `any` type, you should ensure it contains a number. The result of an operation in this list is always a number.

Binary operators: `- * / % << >> >>> & ^ |`

The plus (+) operator is absent from this list because it is a special case; a mathematical addition operator as well as a concatenation operator. Whether the addition or concatenation is chosen depends on the type of the variables on either side of the operator. As Listing 1-15 shows, this is a common problem in JavaScript programs in which an intended addition results in the concatenation of the two values, resulting in an unexpected value. This will be caught in a TypeScript program if you try to assign a string to a variable of the `number` type, or try to return a string for a function that is annotated to return a `number`.

Listing 1-15. Binary plus operator

```
// 6: number
var num = 5 + 1;

// '51': string
var str = 5 + '1';
```

The rules for determining the type resulting from a plus operation are

- If the type of either of the arguments is a `string`, the result is always a `string`.
- If the type of both arguments is either `number` or `enum`, the result is a `number`.
- If the type of either of the arguments is `any`, and the other argument is not a `string`, the result is `any`.
- In any other case, the operator is not allowed.

When the plus operator is used with only a single argument, it acts as a shorthand conversion to a number. This unary use of the plus operator is illustrated in Listing 1-16. The unary minus operator also converts the type to `number` and changes its sign.

Listing 1-16. Unary plus and minus operators

```
var str: string = '5';

// 5: number
var num = +str;

// -5: number
var negative = -str;
```

Bitwise Operators

Bitwise operators in TypeScript accept values of all types. The operator treats each value in the expression as a sequence of 32 bits and returns a number. Bitwise operators are useful for working with Flags, as discussed in the earlier section on Enumerations.

The full list of bitwise operators is shown in Table 1-1.

Table 1-1. *Bitwise Operators*

Operator	Name	Description
&	AND	Returns a result with a 1 in each position that both inputs have a 1.
|	OR	Returns a result with a 1 in each position where either input has a 1.
^	XOR	Returns a result with a 1 in each position where exactly one input has a 1.
<<	Left Shift	Bits in the left hand argument are moved to the left by the number of bits specified in the right hand argument. Bits moved off the left side are discarded and zeroes are added on the right side.

(*continued*)