TypeScript is a language created and maintained 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. In fact, Microsoft has written both the Azure Management Portal (1.2 million lines of code) and the Visual Studio Code editor (300,000 lines of code) in TypeScript. 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 plenty of other development tools have also added support for the language, including VS Code, WebStorm, Eclipse, Sublime Text, Vi, Atom, 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 in the flavor of your choosing. 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.

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. Subsequent versions of the ECMAScript specification are rolled into TypeScript releases, often as early previews that feature down-level compilation to older versions of the specication. 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 the additional features of TypeScript can be output to a number of widely supported versions of JavaScript.

TypeScript has several native frameworks, including Angular, Ionic, RxJs 5, and Dojo 2. Additionally, because TypeScript is such a close relative of JavaScript, you can consume the myriad of existing libraries and frameworks written in JavaScript; Aurelia, Backbone, Bootstrap, Durandal, jQuery, Knockout, Modernizr, PhoneGap, Prototype, Raphael, React, Underscore, Vue, and many more are all usable in TypeScript programs. Correspondingly, once your TypeScript program has been compiled it can be consumed from any JavaScript program too.

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. Source maps can also be generated to aid debugging, with browser developer tools displaying your TypeScript code during in-browser debugging.

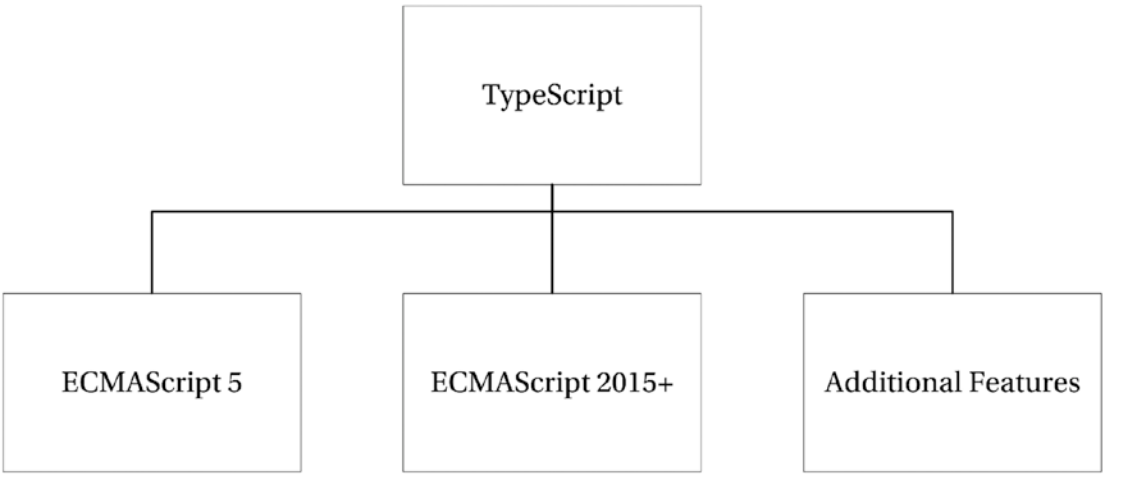

Figure 1. *TypeScript language feature sources*

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 sections. 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 Part 1.

The TypeScript Components

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

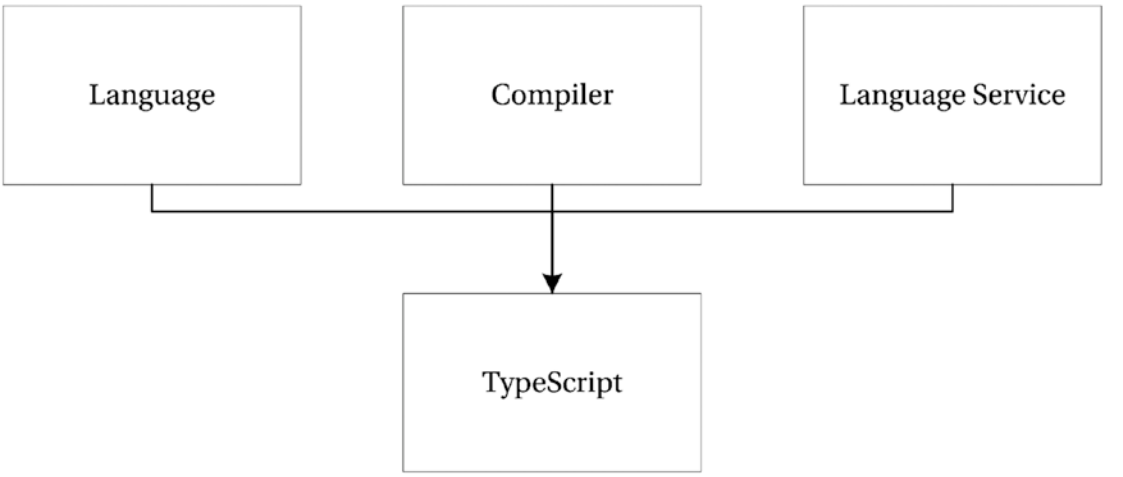

Figure 2. *The TypeScript components*

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 on Node since 2011. JavaScript can even be used to write native applications on operating systems such as Windows.

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 reuse common design patterns easily, and difficulty keeping a readable and maintainable code base. TypeScript solves problems such as these.

Because JavaScript has a C-like syntax, it looks familiar to most programmers. This is one of JavaScript's key strengths, but it is also the cause of many 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 several 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, namespaces, 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 newer ECMAScript specifications. All these features are described in detail in Part 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.v

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 Part 1. A common JavaScript example is shown in Listing 1.

Listing 1. Type juggling

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

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

// result is 20
const 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 Part 3.

The introduction of types in TypeScript does not preclude the use of dynamic types. You can choose when to use types, and when to go without them. TypeScript does not force you to do anything, it is just there to help.

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 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 default way of working and allows your modules to be compiled to suit the most prevalent module loading styles without requiring changes to your code. The details of module loading in web browsers are covered in Part 6 and on the server in Part 7.

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 has bucked this trend traditionally 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, thus promoting the variable into 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.

The introduction of the const and let variable declarations have gone some way to solving this problem, and if you are starting from a clean sheet you can avoid the older var variable declaration altogether.

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.

TypeScript eases scope problems by warning you about implicit global variables, provided you avoid adding variables to the global scope. This safety net is demonstrated in Listing 2.

Listing 2. Accidental global scope error

```
function process() {
    // Error! Cannot find name 'accidentalGlobal;
    accidentalGlobal = 5;
}
```

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).*

TypeScript remains largely faithful to JavaScript. The TypeScript specification adds many language features, but it 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, Ruby, 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 guide were written using Visual Studio 2017, but you can use VS Code, 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. I often use the TypeScript Playground when answering questions on the language.

From the Visual Studio 2013 Spring Update (Update 2), TypeScript is a first-class language in Visual Studio. Prior to this, an additional extension needed to be installed. Although the examples in this guide are shown in Visual Studio, you can use any of the development tools that were listed above. In particular, Visual Studio Code is a free cross-platform editor with native TypeScript support - so you can write your TypeScript code on any machine you have on hand, regardless of the operating system.

It is also worth downloading and installing Node (which is required to follow many of the examples) 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 task runners such as Grunt and Gulp 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 the official Node website.

https://nodejs.org/

To avoid being greatly sidetracked, I have avoided using any task runners to perform additional operations outside of Visual Studio, but once you have mastered TypeScript you will almost certainly want to add a task runner, such as Gulp or Grunt, to your development workflow. I have referenced some web dependencies from the node_modules folder in the examples in this guide; but on a real-world project I would use a task runner to lift and shift the website dependencies into a different folder that I would be happy to deploy to a web server. The node_modules folder often contains a great deal of files that I would not deploy to a web server.

TypeScript Alternatives

TypeScript is not the only alternative to writing to plain JavaScript.

The strongest TypeScript alternative is Babel, a compiler that exposes the latest ECMAScript features with plugins for down-level compilation and polyfills to make your program work in current browsers. The aim of the Babel project is to make the latest features available much sooner than they otherwise would be, but Babel doesn't introduce compile-time type checking. Babel also features in many TypeScript workflows, where Babel is executed after the TypeScript compiler, rather than using TypeScript to perform the down-level compilation. You can read about Babel on the official website:

https://babeljs.io/

For a number of years, CoffeeScript was 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. In 2017, however, CoffeeScript reached the top three "most dreaded languages" in the Stack Overflow developer survey (next to VBA, and Visual Basic 6). In the same survey, TypeScript landed in the top three most loved languages. You can find out more about CoffeeScript on the official website:

http://coffeescript.org/

Another alternative is Google's Dart language. Dart has much 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, and was only intended to be compiled to JavaScript to provide wide support in the short term while native support for Dart was added to browsers.

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 likely remain core to Dart's future in the web browser. The decision by Google to adopt TypeScript for the Angular project may be indicative of their commitment to Dart, although it is still described as a long-term project. 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. Equally, there is little penalty for writing small programs in TypeScript, especially if you already have a workflow in place to generate combined or minified files each time you save your application.

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 reuse code between browser and server applications.

TypeScript solves many problems in JavaScript, but it respects the patterns and implementation of the underlying JavaScript language, for example, the ability to have dynamic types.

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 has gained incredible traction in its first five years.