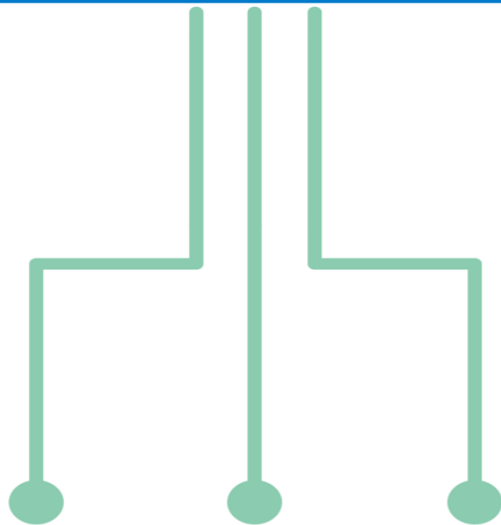


Essential TypeScript

The TypeScript logo, consisting of a solid blue square with the letters 'TS' in a white, bold, sans-serif font.

TS



Jess Chadwick

Essential TypeScript

It's JavaScript... only better.

Jess Chadwick

This book is for sale at <http://leanpub.com/essentialetypescript>

This version was published on 2016-05-08



This is a [Leanpub](#) book. Leanpub empowers authors and publishers with the Lean Publishing process. [Lean Publishing](#) is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

© 2016 Jess Chadwick

Tweet This Book!

Please help Jess Chadwick by spreading the word about this book on [Twitter](#)!

The suggested tweet for this book is:

Reading "Essential TypeScript" by Jess Chadwick and it's great! Definitely worth the read.

The suggested hashtag for this book is [#EssentialTypeScript](#).

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

<https://twitter.com/search?q=#EssentialTypeScript>

To my parents, who instilled a love of learning and teaching.

To my wife and kids, who put up with my late nights and grumpy mornings.

Contents

1. Preface	1
What You Should Know	1
Source Code Examples	2
Errors, Omissions, and Contributions	2
2. Introduction	3
What is TypeScript?	3
Defining “JavaScript”	8
Writing your first TypeScript function	12
3. Getting Started	16
Choosing your TypeScript editor	16
Installing TypeScript in Visual Studio	17
Installing the TypeScript Command Line Interface	18
Configuring the TypeScript compiler	19
4. ECMAScript 2015 Features	24
Optional Parameters	25
Template Strings	27
Let and const	30
For..of Loops	32
Arrow Functions	34
Destructuring	38
The Spread Operator	44
Computed Properties	47
5. Type Fundamentals	49
Introducing JavaScript Types	49
Understanding Type Inference	54
Specifying JavaScript Types	57
Union Types	60
Function Overloads	62
6. Custom Types	64
Defining custom types with Interfaces	64

CONTENTS

Using interfaces to describe functions	69
Extending interface definitions	71
Defining constant values with enums	75
Anonymous Types	79
Using interfaces to dynamically access object properties	81
7. Classes	83
Understanding Prototypical Inheritance	83
Defining a class	90
Applying static properties	93
Making properties smarter with accessors	96
Inheriting behavior from a base class	101
Implementing an abstract class	105
Controlling visibility with access modifiers	107
Implementing interfaces	110
8. Generics	114
Creating generic functions	114
Creating generic classes	116
Creating generic constraints	119
9. Modules	123
Organizing Your Code with Namespaces	124
Using namespaces to encapsulate private members	128
Understanding the difference between Internal and External Modules	134
Switching from internal to external modules	136
Importing modules using Require syntax	138
Importing modules using ECMAScript 2015 syntax	139
Loading External Modules	141
10. Real-World Application Development	145
Converting existing JavaScript code to TypeScript	153
Generating Declaration Files	165
Referencing third-party libraries	168
Converting to External Modules	170
Debugging TypeScript with source maps	175
11. Decorators	179
Implementing Class Decorators	183
Implementing Property Decorators	187
Implementing Decorator Factories	191
12. Appendix: Syntax Examples	195
ECMAScript 6 Features	195

CONTENTS

Custom Types	201
------------------------	-----

1. Preface

Put simply, TypeScript is a superset of JavaScript, which means that it builds on top of JavaScript's mature foundations, adding a variety of helpful syntax and tooling onto the language. It brings the power and productivity of static typing and object-oriented development techniques to the core JavaScript language. In fact, TypeScript is designed to be compiled into fully-compatible JavaScript.

And, as JavaScript's reach continues to expand, so do the opportunities to leverage TypeScript to create JavaScript-based applications.

In this book, I will show you everything you need to know in order to create full-fledged JavaScript applications using the TypeScript programming language, starting by revisiting some JavaScript fundamentals and proceeding all the way to demonstrating how to convert an entire existing JavaScript codebase to take full advantage of what TypeScript has to offer.

One of the nicest things about TypeScript is that its tooling is fully cross-platform, which means that you can develop TypeScript applications using Windows, Mac, or even Linux. Microsoft even offers an online editor to play around with in order to get your feet wet without having to install anything at all!

What You Should Know

Before getting into the content of the book, I want to take a minute to set some expectations - especially when it comes to what you should already know in order to get the most out of this book.

As I mentioned previously, TypeScript is a superset of JavaScript, which means that it is an extension of JavaScript, adding new features and syntax on top of the core language.

In this book, I'm going to focus the unique features that TypeScript adds to the language rather than explaining the fundamentals of JavaScript itself and I'll be depending on the assumption that you've already got a decent familiarity with the JavaScript language to the point where you should be able to pretty easily recognize where JavaScript ends and TypeScript begins.

In other words, if you're comfortable using JavaScript to enhance a website to make an AJAX call and update parts of the webpage when the user clicks a button, for example - even if you prefer to use a library like jQuery to help you - you'll probably be able to follow along with this book just fine.

If, however, you're relatively new to JavaScript development or don't have much experience with the language, I strongly suggest you spend some time to learn JavaScript itself first, then come back to this book to see how TypeScript can make your JavaScript experience even more productive.

Source Code Examples

Since this is a book about software development, it's going to have a lot of code. While I expect you to follow along by running the same code on your local machine, I don't necessarily expect you to have to copy and paste every character that I show. And, I don't know about you, but whenever I am having trouble running some sample code that I'm writing along with a book or training course, I always like to have a full, working version of that code that I can refer to in order to help me figure out what I need to change in order to get my version of the code running.

So, to make that possible, I've captured a snapshot of the code I show you at the end of every section and it is available to you through my GitHub repository at: github.com/jchadwick/EssentialTypeScript¹.

You can either browse the repository on the GitHub website or make a local clone of the repository and start looking through it! Note that this repository also contains snapshots at the end of each section, each one represented as a Tag in the repository. That means that if you want to see the code for a specific section, you can switch to that tag in your local repository or select that section from the dropdown on the GitHub site. On the GitHub site, this will change the context of the whole source code tree and you can then use the website to browse through the code as it stands at the end of that section. Also note that you can download a zip archive of any tag as well by visiting the Releases tab and clicking on the "zip" link for the tag that you'd like to download.

The full source code is there if you need it – I encourage you to refer to it as much and as often as you need to in order to get the most out of this book!

Errors, Omissions, and Contributions

Oh, and guess what else is on GitHub – *this very book*! That's right - if you head on over to github.com/jchadwick/EssentialTypeScriptBook² you'll find the repository with this full text. From there you're welcome to submit Issues or even fork the whole thing, make your own changes, and submit a pull request!

¹<https://github.com/jchadwick/EssentialTypeScript>

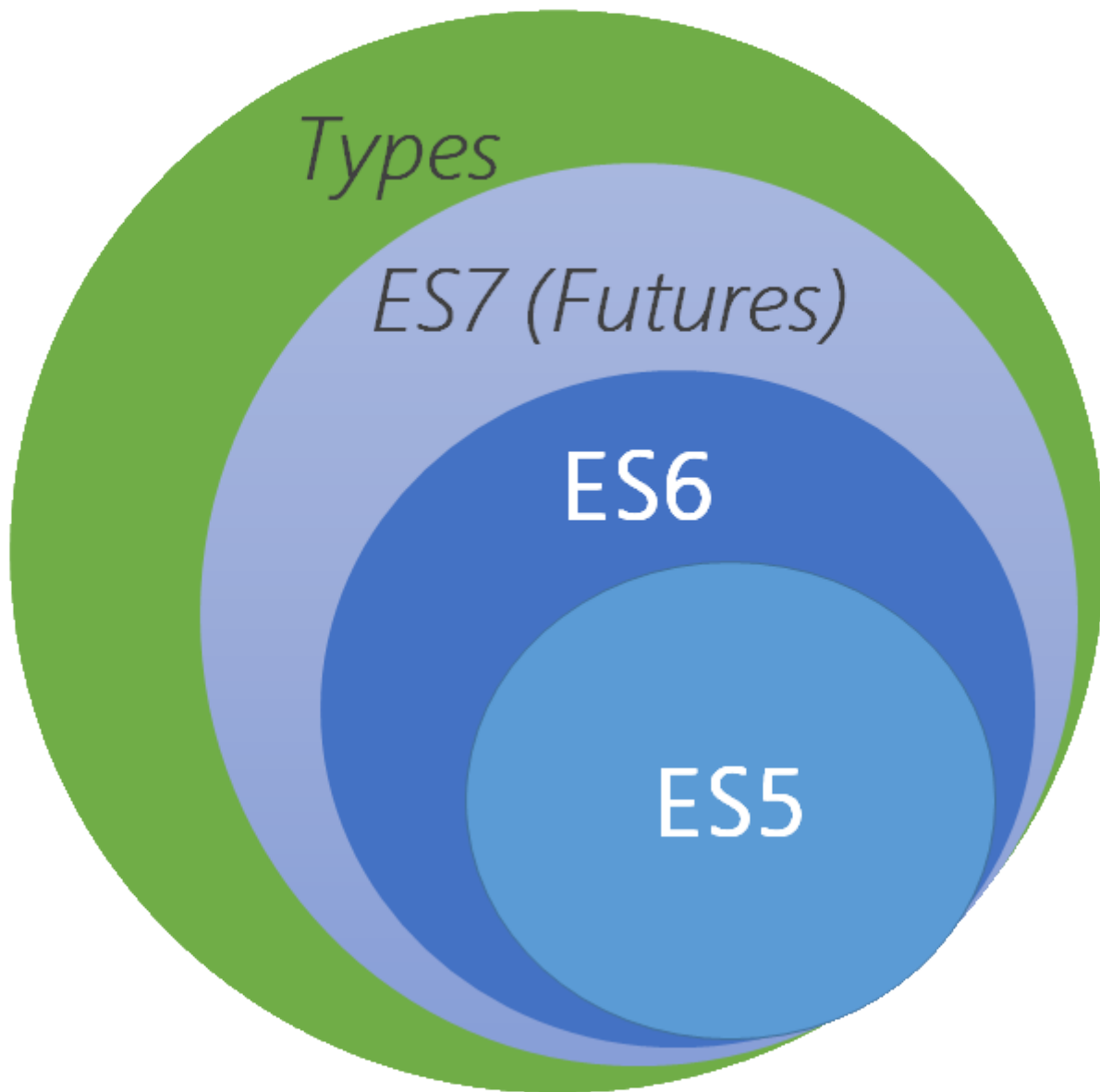
²<https://github.com/jchadwick/EssentialTypeScriptBook>

2. Introduction

What is TypeScript?

In this book, I'm going to show you everything you need to know in order to write applications using TypeScript. In this section, I'm going to start with the basics and explain what, exactly, TypeScript is.

Put simply, TypeScript is a superset of the JavaScript programming language that adds the concept of static typing to the core features of JavaScript.



Superset of JavaScript

This is a big deal because JavaScript is - and always has been - a dynamic language.

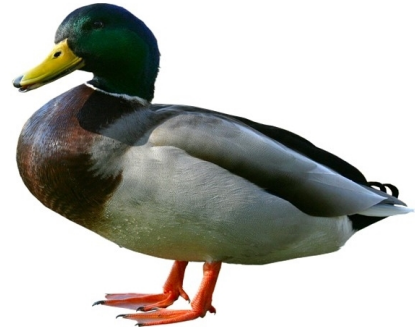
In order to illustrate why this is a big deal, I'll quickly define what it means to be a static language or a dynamic language and how they seem, at first, to be completely opposite and incompatible things.

Dynamic Types	Static Types
Type System	Type System
Forgiving	Rigid
Great for web browser object model	Promotes stability and maintainability

Both dynamic and static languages rely on a type system - that is, definitions of data structures and their behaviors - to ensure that programs are correct. It's just that the two kinds of languages validate those types in different ways.

Dynamic languages aim to be much more forgiving at development time, relying on the concept of “duck typing” to validate that a particular object can be used in a certain way.

```
{  
  name: "Bill",  
  quack: function() {  
    // Quack!  
  }  
}
```



```
function sayQuack(target) {  
  target.quack();  
}
```

Duck typing

“Duck typing” refers to the idea that if it looks like a duck, walks like a duck, and quacks like a duck, it must be a duck... In other words, if my code expects an object that has a method called “quack” on it, and I get an object that has a method named “quack”, well that’s good enough for me – I don’t need to validate anything else about that object. The net result of this approach is that tools don’t have enough information to catch errors before the application runs, for example, if I’ve accidentally typed the method name “quake” rather than “quack”.

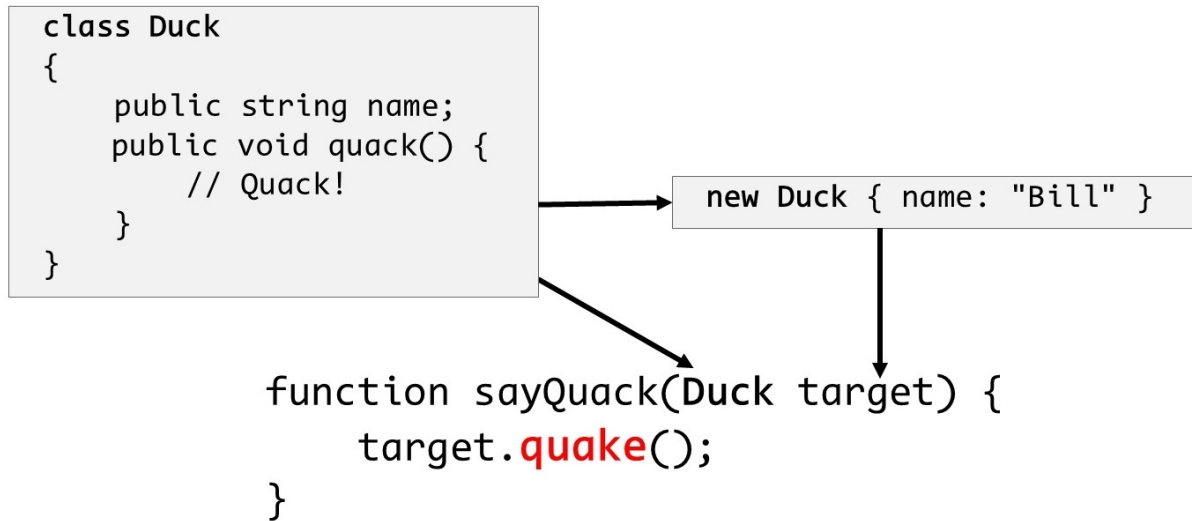
This means that errors are only ever caught while the application is running... after it’s too late to do anything about it.

Statically-typed languages, on the other hand, are much more rigid. They aim to catch development errors before the code is even executed and they do this by imposing restrictions on how you can interact with objects, forcing you to very clearly specify everything about the object that you’re going to interact with.

```
function sayQuack(target) {  
  target.quake();  
}
```

Common typing error

In static typing world, you can’t just call a “quack” method on any object - you first need to explicitly define a type that has that quack method, as well as any parameters that need to be passed into that method not to mention the value that the quack method will return to its callers. Only then can you use that instance as a parameter or create an instance of that class to pass around to other objects or methods.



Static typing in action

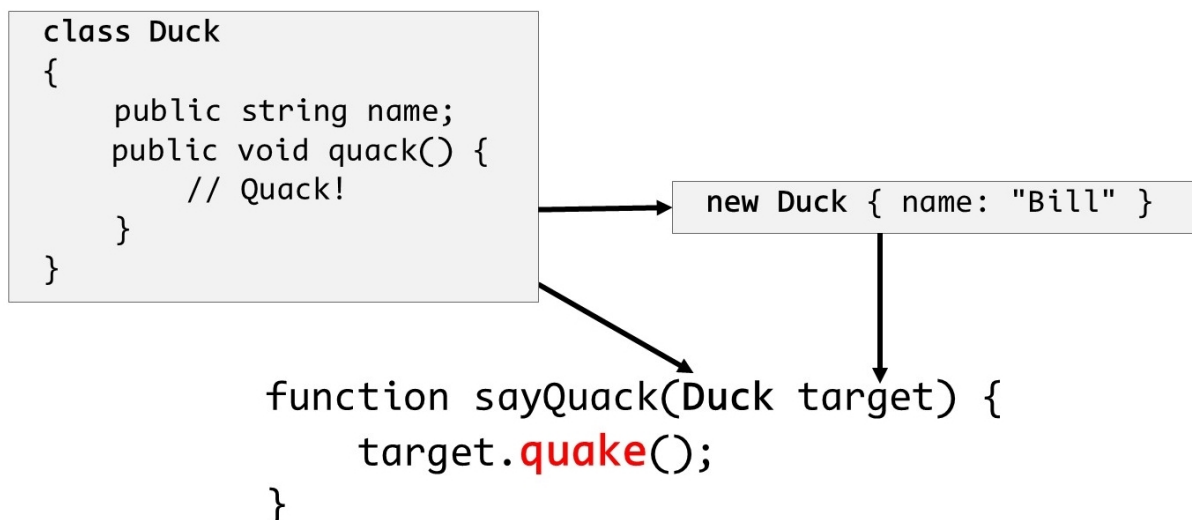
```

class Duck
{
    public string name;
    public void quack() {
        // Quack!
    }
}

```

Example of a static type

With all this information explicitly provided, the tooling is able to easily tell me when I've misspelled the call to the "quack" method well before my application is running – heck, with modern tools it'll likely take less than a second for me to find out that I've done something wrong!



Static typing in action

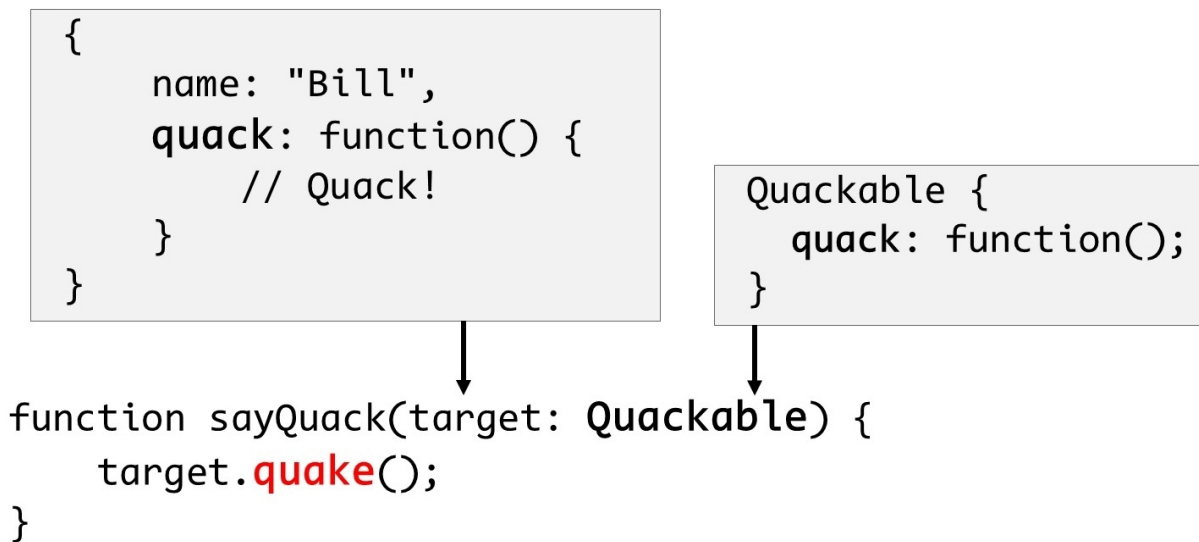
If it sounds like I'm making a case for one approach over the other, I'm not – each of the typing

approaches lends itself to a different set of scenarios so it quite simply depends on what you're looking to do with the language.

But here's the thing: who's to say you can't use both at the same time?

For better or worse, JavaScript is a dynamically typed language. This makes it particularly suitable for dealing with the web browser object model that it was originally designed to work with. But, JavaScript has been asked to do far more these days than what it was originally designed to do, and in some of those cases some people start to find its dynamic nature more of a curse than a gift. In these scenarios, being able to apply static typing can have a hugely beneficial effect on the stability and maintainability of the code.

Enter TypeScript.



TypeScript Typing

TypeScript is a superset of JavaScript – that is, it extends JavaScript with static typing capabilities; it doesn't fundamentally change the way that JavaScript works and it especially doesn't take away any of JavaScript's powerful dynamic capabilities. TypeScript simply allows developers to opt-in to the static typing approach whenever it will benefit them.

Let's go back to the quacking example from earlier: JavaScript is a dynamic language, which means that I can write code that calls a "quack" method on any object and JavaScript isn't going to yell at me until runtime when it turns out that quack method doesn't actually exist. In some cases, that may be what I want. But, in other cases – and that case could be 5 lines further down in the same piece of code – maybe I want to be completely sure that the object actually has a quack method before I call it, and I don't want to wait until runtime to find out. In that case, I'll use TypeScript's language features to make that very clear right in my code and TypeScript will validate that assertion before my code executes.

This all comes at a cost, of course - you've got to define types in order to use them, and that means

writing more code. You just have to weigh the cost of writing that code with the value of finding out about these kinds of issues before your code executes. For many people and many scenarios, having this piece of mind is well worth the extra work of defining the types.

And, if finding out about code issues as early as possible isn't enough, having to explicitly define the types of objects you work with also tends to make your code easier to understand and maintain, especially if you have multiple developers working on the same codebase. In cases such as these, static types tend to have an illuminating effect, helping developers understand what they can and can't do with a particular object.

But perhaps the best thing about TypeScript is that you don't always have to use static types. While introducing static typing may help deal with a lot of different scenarios, there are still other scenarios that benefit from maintaining a completely dynamic approach, and TypeScript supports those, too. With TypeScript, static typing is strongly encouraged, but whenever you feel the need to ditch the safety of type checking in order to take full advantage of JavaScript's dynamic roots, just let TypeScript know and it will leave you without that type-checking safety net for as long as you like.

At this point, you're likely starting to form an opinion about whether or not TypeScript sounds like a good fit for you. If this sounds like just the thing you've been looking for, I think you're going to love this book! If you're still hesitant about whether or not TypeScript offers enough value to make up for the extra work you'll have to do to specify all your types, I urge you to at least read the rest of this chapter to get the full picture before making up your mind.

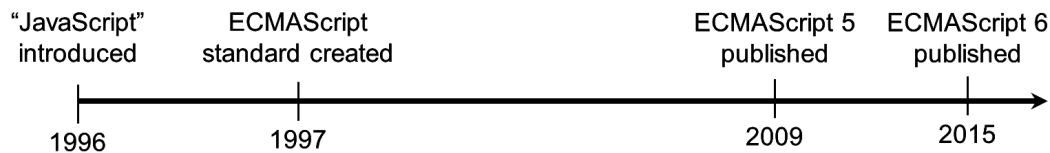
Either way, head on to the next section where I'll get a little deeper into TypeScript's relationship with JavaScript and how you can still benefit from TypeScript even if you never take advantage of its static typing features!

Defining "JavaScript"

I know I've already said it a few times, but I'm going to go ahead and say it again because it's just that important: TypeScript is a superset of JavaScript. And, since JavaScript is going to play such a big part in this book, I think it's pretty important to define what it is that I'm referring to when I use the term JavaScript.

I'll start with a real quick history lesson:

History of JavaScript



JavaScript Timeline

The JavaScript language was first introduced in 1996 in the Netscape Navigator browser. Soon, other browsers started implementing syntax and APIs incredibly similar to - but not exactly compatible with - Netscape's implementation and within about a year or so, the ECMA International standards body attempted to get everyone on the same page by publishing a standard which defined a common specification for all implementations to follow; this standard was called ECMAScript.

As such, you'll often hear the JavaScript language referred to as ECMAScript, or simply "ES" followed by the version number of the standard - for example, "ES5", "ES6", or "ES2015". And, by the way, the ECMA group decided to reset the version numbering in the sixth version, but not until after the community had already taken to calling it "ES6", so just keep in mind that "ES6", "ES2015", and "ECMAScript 2015" all refer to the same exact thing. Yeah, confusing, I know...

Semantics aside, and especially throughout this book, it's generally fine to use the terms JavaScript and ECMAScript interchangeably when referring to the various implementations of the language, as most implementations follow the standard close enough that the names can be used pretty synonymously.

Fast-forward to today, where we've got many more browsers and other platforms that leverage JavaScript, as well as several more versions of the ECMAScript standard. However, just because a standards body publishes a version of a standard doesn't mean every feature in that standard is immediately available in all implementations and platforms. The reality is that some implementations already had some of the features prior to the feature being officially approved and, despite being part of the approved standard, some other features may not actually be available in all implementations for quite a while.

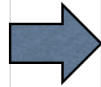
As a JavaScript developer, it can be incredibly frustrating to be aware of a convenient or powerful new feature that is available in the language you're using, but not being able to leverage that feature out of fear that it won't be implemented in one or more of the browsers or platforms that you must support.

ECMAScript Browser Compatibility

A screenshot of the ECMAScript Browser Compatibility table. The table lists various ECMAScript features on the left and shows their support status across different browsers (Chrome, Firefox, Safari, Edge, IE, Opera, etc.) in the columns. The cells are color-coded: green for supported, yellow for partially supported, and red for not supported.

TypeScript

```
class Duck {  
  name: string = 'Bill';  
  quack(): void {  
    // Quack!  
  }  
}
```



ECMAScript 5

```
var Duck = (function () {  
  function Duck() {  
    this.name = 'Bill';  
  }  
  Duck.prototype.quack = function () {  
    // Quack!  
  };  
  return Duck;  
})();
```

Transpilation

Enter the world of transpilers, or tools that transform code from one language into another language. In our example, this means the ability to write code that leverages next-generation ECMAScript features, yet compiles down to JavaScript that is fully compatible with the current generation ECMAScript implementations that are widely available today.

There are handful of popular transpilers available and each of them expose the next-generation ECMAScript features in a different way. Some transpilers attempt to be true to the ECMAScript standard, allowing you to write your code as if the latest version of the standard were available everywhere. Other transpilers use a syntax that is effectively a completely different language altogether.

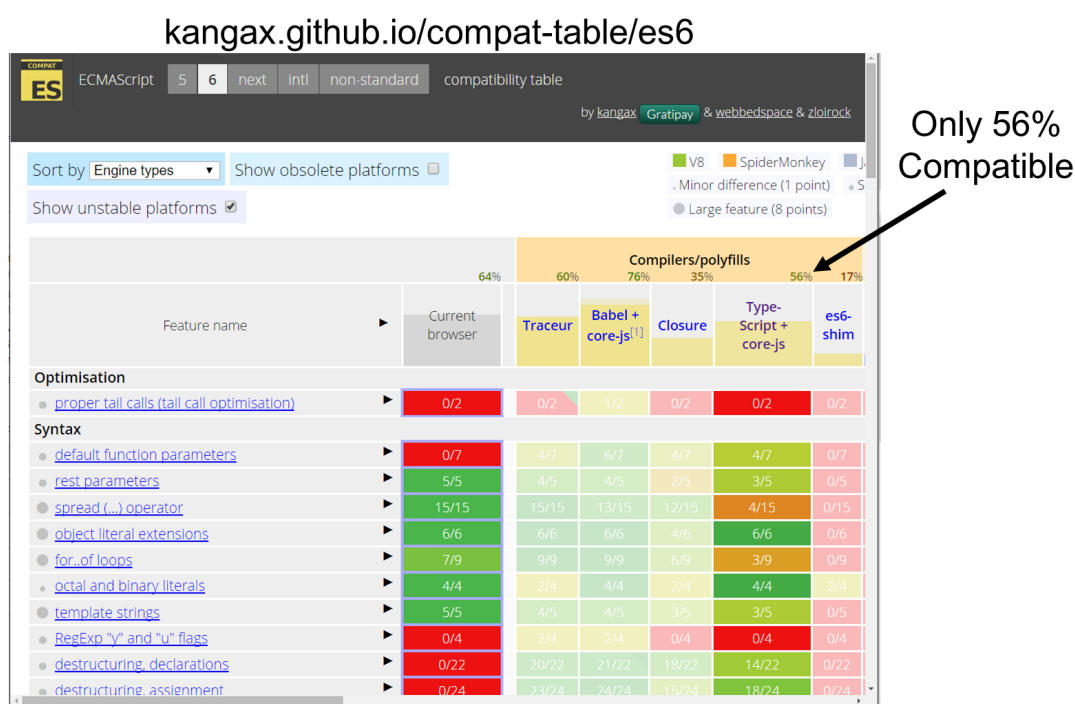
Regardless of what syntax you use to write your code, however, all transpilers compile that code into the JavaScript that is most compatible with all implementations that you're looking to run your application on.

As of the time of this writing, ECMAScript 2015 has only been recently ratified, which means that it will still be a while until all of its features are available everywhere you want them to be. The good news is that, in addition to all of the static typing features that I'll be showing throughout this book, TypeScript also implements a handful of ECMAScript 2015 features as well. In fact, I'll spend the entire first chapter of this book just showing you how to implement these ECMAScript features with TypeScript before I even get into showing you how to leverage TypeScript's static typing features.

And in this way, TypeScript definitely falls under the "transpiler" category. More importantly, it

falls into the class of transpiler that aims to remain true to the ECMAScript standard rather than the kind of transpiler that reinvents its own language. So, when I say that “TypeScript is a superset of JavaScript”, I’m referring to this idea that TypeScript starts with being as compatible with the latest ECMAScript standard as possible, and then adds even more features on top of that foundation.

The chart I showed earlier is from an excellent website I like to refer that contains an exhaustive list of all of the ECMAScript standards and which tools, runtimes, and implementations currently support them. TypeScript is among the tools that is compared on this list and as of the time of this writing, this site states that TypeScript implements about a half of the total ECMAScript 2015 standard. While that may seem unimpressive - especially compared to some of the other tools such as Babel - it’s important to keep in mind that the real value TypeScript offers is adding static typing support to JavaScript – as an added perk, it also implements some ES6 features as well.



TypeScript: Not a full ECMAScript 2015 implementation

Also keep in mind that TypeScript also happens to implement some of the future features defined in the next ECMAScript standard, ECMAScript 7, and that standard is not even close to being finalized yet. I think you can expect TypeScript to continue to growing, remaining as compatible with current, approved standards while also implementing proposed features. In other words, in addition to all of the benefits of static typing that TypeScript adds, it also helps you leverage many JavaScript features that may not otherwise be available to you. Personally, I think that’s a great deal.

For more details of the ECMAScript 2015 features that TypeScript supports and how to leverage them, check out the “ECMAScript 2015 Features” chapter of this book.

But, enough about JavaScript and ES6 - head on to the next section to get a glimpse into how you

use TypeScript to implement all this static typing that I've been raving about!

Writing your first TypeScript function

Over the past few sections, I've described what TypeScript is in theory, but now it's time to see it in action. In this section I'm going to take you on a whirlwind tour of the most important features of TypeScript - the features that you'll find yourself using on a daily basis. If you find that I'm going too fast in this section, don't worry - I'll spend the rest of this book diving into all the details so that you won't miss a thing.

In later sections, I'll show you how to install TypeScript onto your local system, but I'm going to skip that for now and start by showing you the simplest possible way to start learning TypeScript: TypeScript's free, online TypeScript editor. This editor is a powerful tool that gives you full syntax highlighting, auto-complete capability, and compile TypeScript right in the browser

You can get to the editor by opening up your favorite browser and going to typescriptlang.org¹, then clicking the "play" menu item at the top of the TypeScript homepage.

Once the page has loaded, I'll clear all the code that's already in the editor and replace it with this snippet of JavaScript code:

```
1  function speak(value) {  
2      document.write(value);  
3  }  
4  
5  var greeted = "World";  
6  var greeting = "Hello, ";  
7  var whatToSay = greeting + greeted;  
8  
9  speak(whaToSay);
```

Right off the bat, the TypeScript editor is already helping spot problems in this code: turns out I spelled the variable name wrong in the last line! That's simple enough to fix - I'll just add the missing character.

With that fixed, let's look at what this snippet is doing: we've got a function called `speak` which simply writes out the variable to the document, and a couple of variables that we're adding together to produce the full text that we are passing to the `speak` function.

Seems straight-forward enough – what could go wrong, right? As it turns out, plenty. For instance, what happens when I change the value of the `greeted` variable to a different type, such as a number?

¹<https://typescriptlang.org>

```
var greeted = 42;
```

Actually, that's fine, because JavaScript is smart enough to convert that number back into a string when the number is added to a string. And, since the result will always be a string type and never be ambiguous, TypeScript is happy as well. But, what if I turned BOTH values into numbers? Then the resulting value would be a number!

```
var greeted = 42;  
var greeting = 1;
```

Now, we've gone from passing a string value into the `speak()` function to passing a number value. True to JavaScript's dynamic nature, it doesn't care - it just goes ahead and converts it to a string in order to render it to the browser. TypeScript is also happy, since the number can be converted to a string.

Here's the problem: that's not what I intended to happen. I only ever intended for the `speak` method to accept a string value, but JavaScript doesn't let me express that intent. The only way I can ever guarantee that the parameter value is a string is to write code to check it at runtime.

Here's where TypeScript comes in handy. If I only ever want strings to be passed in, I can say so by explicitly saying that the `value` parameter is a string using this syntax:

```
speak(value: string)
```

As soon as I add this type, TypeScript immediately notices that I am trying to pass a number value as a parameter to the `speak` function on the last line of code and underlines that code in red to indicate the error. When I hover over the error, TypeScript tells me that I'm trying to pass in a number when the function expects a string.

TypeScript is raising this error because it thinks that the variable I'm passing is a number. TypeScript thinks the variable is a number because it was initialized as the result of adding two numbers together. To make this error go away, I need to tell TypeScript that the `whatToSay` variable is, indeed, a string. I can do that using the same syntax that I just used to define the string type on the method parameter: right after the variable name, and before the assignment.

```
var whatToSay: string = greeting + greeted;
```

Of course, that didn't actually solve the problem, it just gave TypeScript more information to figure out where the problem was. Remember - the real problem is that I'm adding two numbers together which produce a number as a result. What I need to do is change this addition to produce a string rather than a number, and I can do that by explicitly calling the `toString()` method on one of the number values to turn it into a string.

```
var whatToSay: string = greeting.toString() + greeted;
```

And, finally, the error goes away completely.

It's interesting to note that what I'm left with now doesn't make a whole lot of sense, either: why in the world am I converting two numbers to strings and adding them together? The answer is that this is just a demo, but it does illustrate an important point: TypeScript may help a lot, but it's not going to solve all your problems - you're still responsible for writing code that makes sense... but that's a different book altogether!

Of course, the real solution is for me to never have made them numbers in the first place...

```
var greeted = "World";  
var greeting = "Hello, ";
```

Now that I've shown how to specify the type of variables and function parameters, let's talk about function return values. Take a look at what TypeScript says when I hover over the speak function: notice that void at the end? That's TypeScript saying that this method doesn't return any value. And, when I add a return value, that type changes:

```
function speak(value: string) {  
    document.write(value);  
    return value;  
}
```

If I hover over the method again I can see that TypeScript has now determined that the method returns a string value. Likewise, let's see what happens when I change the value that I'm returning to the length of the variable:

```
return value.length;
```

Now when I hover over the function I can see that TypeScript thinks that it now returns a number.

When I want to explicitly define which value the function returns, all I have to do is use the same syntax at the end of the function signature as I did with the variable definition.

For example, let's say that I want to make sure that the speak() function always returns a string type. I'll simply add that to the end of the function signature:

```
function speak(value: string): string {
```

With the return type specified, TypeScript can now tell me when I'm not returning a value that matches that type, which is the case right now since I'm actually returning the length of the value rather than the value itself. Luckily, that's pretty easy to fix:

```
function speak(value: string): string {  
    document.write(value);  
    return value;  
}
```

Once I fix that, TypeScript is happy again and all the errors go away.

And that's your whirlwind introduction to TypeScript's static typing functionality. Of course, there is far more to learn about TypeScript - and trust me, I'll show it all to you in the other chapters - but everything else you're going to see throughout this book rests on top of these fundamentals.

Now that I've given you the high-level overview of what TypeScript is and how it might be beneficial to you, head on to the next chapter where I'll show you how to install and configure the tools you'll need in order to work with TypeScript on your local machine as opposed to an editor on a website.

3. Getting Started

Choosing your TypeScript editor

In the last chapter, I described what TypeScript is and how it relates to JavaScript.

In this section, I'm going to talk a little bit about the tools you'll need in order to work with TypeScript.

Actually, I don't really need an entire section because the answer is simple: since TypeScript is just a superset of JavaScript, you'll almost certainly end up using the same tools to work with TypeScript as you already use to work with JavaScript today!

TypeScript is a compiled (or "*transpiled*") language, so you will need at least two tools to work with TypeScript: a text editor, and the TypeScript compiler.

Technically speaking, you can use any text editor you want, however you're probably going to want an editor that understands TypeScript well enough to offer some nice features such as syntax highlighting and auto-complete functionality built in.

Lucky for us, TypeScript has been around long enough and gained enough popularity that your favorite JavaScript editor almost certainly has some kind of TypeScript support either built-in or available as an extension. For most of the popular text editors such as Sublime Text, TextMate, Atom, Notepad++, Visual Studio Code, it's as simple as searching in their library of extensions and installing the feature.

Then, of course, there are full-fledged Integrated Development Environments (or IDEs) such as Visual Studio, WebStorm, or Eclipse. Just like basic text editors, your favorite IDE will likely have great TypeScript support, but some of them may even have the TypeScript compiler built right in.

In the following sections I'm going to show you a couple of ways of installing TypeScript take make sure that one of them will be a good fit for you:

- The next section will be for you Visual Studio developers who like to have everything easily accessible right there in an nice IDE and integrated into the Visual Studio projects and solution structure that you already know and love. If this describes you, then you may never need to open up a command line to use TypeScript.
- The section after that, however, will be for everyone else. In that section, I'll show you how to install the Node Package Manager (or NPM) and use it to install and execute the TypeScript command line interface while editing your TypeScript files in your text editor of choice, whether it's TextMate, Atom, or the new cross-platform editor, Visual Studio Code.

In the rest of this book after these next two sections I will be using Visual Studio Code to edit my TypeScript files and the command-line interface to compile them. However, after you've installed TypeScript using one of these next two sections, you will be able to follow along with me throughout the rest of the course regardless of whether what text editor you choose or even which OS you're running, be it Windows, Mac, or even Linux.

So, enough talk - choose one of the next two sections that applies to you and let's get installing!

Installing TypeScript in Visual Studio

I mentioned in the last section that you can choose any editor that you wanted in order to work with TypeScript and that some editors had support for TypeScript already built-in. Considering that Microsoft created both the Visual Studio IDE and TypeScript, it should come as no surprise that Visual Studio offers great support for TypeScript built right in... well... kind of.

Let me clarify that a little bit. Visual Studio has been around a long time and there are several different versions still in wide-spread use today, and each of them offers varying levels of support for TypeScript.

If you're using Visual Studio 2010 or older, I'm sorry, but there is no built-in TypeScript or extension that I'm aware of. If you really wanted to, you could jump to the next section and install the command line version of TypeScript, then use Visual Studio 2010 as your HTML and TypeScript editor, but frankly if that is your situation then there are a number of much better TypeScript editors available, and most of them are free.

If you're using Visual Studio 2013, you didn't get TypeScript support right out of the box, but you can install the TypeScript extension by visiting typescripqlang.org¹, then clicking the "Download" button at the top, and selecting "TypeScript for VS2013".

This will bring you to the "TypeScript for Visual Studio 2013" extension page where you can download the extension and run it to enable full TypeScript support in Visual Studio 2013.

On the other hand, the latest Visual Studio release - Visual Studio 2015 - actually has full TypeScript support right out of the box, but it's actually a little outdated so you're going to want to install the most recent version of TypeScript on top of what Visual Studio already provides. You'll follow the same steps to install the extension as you did with Visual Studio 2013, except you'll choose the "TypeScript for VS 2015" option instead.

Once the extension is installed you should have everything you need in order to follow along with the course.

¹<http://www.typescripqlang.org>



Keep in mind that I'll be using the command line interface and Visual Studio Code in order to demonstrate all my examples, however if you are familiar with Visual Studio you should be able to follow along without any problem at all.

In that case, you're welcome to read the next section where I install the command line interface, but if you don't want to, that's fine - it's not required, and you can just skip on to the next section where I show you how to configure the TypeScript compiler in your project.

Installing the TypeScript Command Line Interface

If you're reading this section, either you're running on a Mac or Linux machine or you're a Windows user and you've decided that you're going to use a lightweight code editor instead of a full-blown IDE like Visual Studio.

Regardless of how you got here, I think you made a good choice. Many people find the command line interface rather daunting, but I find that once you get used to it it's actually quite a nice and lightweight option.

In order to be truly cross-platform compatible, the TypeScript command line interface is actually built in NodeJS. Don't worry - you won't have to learn how to be a NodeJS developer in order to use the TypeScript command line, but you will need to install the Node Package Manager (or "NPM"), which comes included in the NodeJS installer.

To install NodeJS, simply visit nodejs.org² and then choose the appropriate download for your environment. Feel free to install either the "mature" or "latest" version - as of this writing, TypeScript will work on either.

Once you've installed NPM, it's time to use it to install TypeScript. In order to do so, simply jump into a command prompt: in Mac you can open Spotlight and search for "terminal", and in Windows you can open the Start menu and search for either "Command Prompt" or "PowerShell" - either one will work fine, but personally I like PowerShell because it offers a resizable window and nicely-colored text highlighting.

Keep in mind that all of the commands I'll show you in this book should be exactly the same, regardless of which OS or command prompt you're using.

Once in the command prompt, type the following:

```
npm install -g typescript
```

This command tells the Node Package Manager to install the TypeScript compiler (and the "-g" option tells it to install it globally so the command will be available to you regardless of which project you're working in. n Once it's done, you can verify that everything worked by typing the command "tsc".

²<http://nodejs.org>

```
tsc
```

Technically speaking, that's all you need in order to compile TypeScript files into JavaScript, however in order to follow along with this book you'll also need to open up a web page, and for that you'll need a web server. You could probably get away with just saving the files to your local machine and opening them directly in your browser, but since we're already at the command prompt and we already have NPM installed, let me introduce you to one of my trade secrets called "lite-server".

Lite server is a little lightweight web server that not only serves your files for you, it also injects a little script into each of your page that listens for changes to those files. When the server detects a change, it refreshes the browser automatically. Trust me, web development is much more enjoyable when you don't have to keep hitting "refresh" all the time.

To install lite-server, simply use the "npm install" command like you did for typescript, but install "lite-server" instead, like this:

```
npm install -g lite-server
```

Now that you've got all that installed, let's move on to the next section to create our new project and configure it for TypeScript.

Configuring the TypeScript compiler

In this chapter, I've shown you how to get TypeScript installed on your local machine, and now it's time to show you how to start up a new TypeScript project.

Now, you can use TypeScript anywhere that JavaScript is supported – on the client, server... heck, I can even code my Pebble smartwatch using JavaScript! However, I assume that the majority of the time you're using TypeScript it'll be to build web applications in the browser.

As such, our project starts by simply creating a new folder with an index.html file to host our generated JavaScript.

So, go ahead and create a folder to hold our project. Throughout this book I'll be showing you how to build a Todo application, so let's call the folder "TypeScriptTodo".

```
mkdir TypeScriptTodo  
cd TypeScriptTodo
```

Once I've created the folder, I'll open it up in my favorite text editor, Visual Studio Code.

```
code .
```

Note that if you're using the Visual Studio IDE instead of following these steps, you can do this instead:

Open Visual Studio

Choose "File > New Project"

Choose the "Web Application" project type

Then, choose the "Empty Web" template and enter "TypeScriptTodo" as your project's name

Once I have the folder open in my editor, I'll create a new HTML file named "index.html" and populate it with some markup:

index.html

```
<!doctype html>
<html lang="en">
<head>
  <title>TypeScript Todo App</title>
  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/b\
ootstrap.min.css">
</head>
<body>
  <div id="container" class="container">
  </div>
  <script type="text/javascript" src="app.js"></script>
</body>
</html>
```

Other than the fact that I've linked to the Bootstrap UI framework stylesheet on line 5 to make my site look a little bit nicer, this is nothing but some boilerplate HTML. Notice how I have a link to a JavaScript file called `app.js` on line 10, even though this file doesn't exist yet.

That file will be the JavaScript output from this new TypeScript file I'll create, named `app.ts`.

Now with all this in place, it's time to get the TypeScript compiler working!

Note that if you're in Visual Studio, all that you should have to do at this point is save your project and it should automatically generate the JavaScript file for you. You won't have to do any of this command line stuff (but it's still interesting to read about!).

Technically speaking, the simplest way to build compiler your TypeScript files is to jump into your command prompt and simply run the TSC command, passing in the names of all the TypeScript files you want to compile, like this:

```
tsc app.ts
```

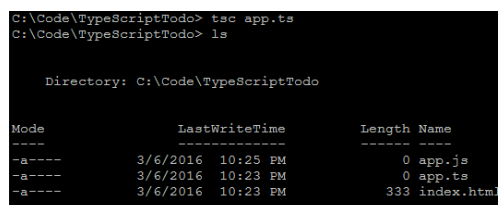
Even though the command doesn't emit any output, I can see that it works because I now have an `app.js` file in the same folder.

I can even have the TypeScript compiler watch for changes to my files and recompile them automatically by passing in the `-w` option, like this:

```
tsc -w app.ts
```

When I pass in the `-w` option, TypeScript compiles my file, then tells me that it's waiting for changes.

Then, I can go back to my editor, make some more changes...

A terminal window showing the execution of the TypeScript compiler. The prompt is 'C:\Code\TypeScriptTodo>'. The first command is 'tsc app.ts', which produces no visible output. The second command is 'ls', which produces a directory listing for 'C:\Code\TypeScriptTodo'. The listing shows three files: 'app.js' (0 bytes), 'app.ts' (0 bytes), and 'index.html' (333 bytes). All files were last written on 3/6/2016 at 10:23 PM.

Mode	LastWriteTime	Length	Name
-a----	3/6/2016 10:23 PM	0	app.js
-a----	3/6/2016 10:23 PM	0	app.ts
-a----	3/6/2016 10:23 PM	333	index.html

TypeScript compiler output

```
app.ts
```

```
var todo: string = "Pick up drycleaning";
```

And then look at the `app.js` file again...

```
app.js
```

```
var todo = "Pick up drycleaning";
```

and see that my changes have been compiled.

However, if I run the `tsc` command again I can see that the TypeScript compiler actually supports a whole lot more options than just `-w`.

And, if I wanted to make use of any of those options, I'd have to type them as command line parameters every time, which would get pretty old after a while.

In fact, if you're using the full-blown Visual Studio IDE, you don't have a command line, so where would you put these options!?

Luckily, there is a solution that solves both of these problems: the TypeScript configuration file, or `tsconfig.json`.

This file is just a simple JSON file, so just add a new file to the folder named `tsconfig.json` and, since the TypeScript configuration object is just a JSON object with configuration values, start by defining an empty JSON object:

tsconfig.json

```
{  
}
```

Then, in the Visual Studio Code editor and the full-blown Visual Studio IDE, you can hit **Ctl-Space** to see the configuration options you have available to you.

Choose the first one, “**compilerOptions**”. Then, hit **Ctl-Space** again to see what compiler options are available:

tsconfig.json

```
{  
  "compilerOptions": {  
  }  
}
```

There are a whole lot of configuration options - and we’ll see a lot of them in this book - but let’s start with the simplest one: the ECMAScript version that we want to target.

In order to do that, choose the “**target**” option.

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es3"  
  }  
}
```

Once it’s added, we can delete the default value and hit **Ctl-space** again to see what other versions of ECMAScript we can target, and since we’re looking for ES5 compatibility, go ahead and pick the “**es5**” option.

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5"  
  }  
}
```

I'll explore later how changing the target ECMAScript version changes the JavaScript that TypeScript generates, but for now I'm just going to save the file and go back to the command line to see how just having this configuration file in place changes how the compiler behaves.

When it starts, the TypeScript compiler looks for this `tsconfig.json` file in the root folder. If it finds it, it treats the whole folder - and all of its sub-folders - as one big project, and automatically detects any TypeScript files.

So, the biggest difference now is that we no longer have to tell TypeScript which files to compile - just type in `"tsc -w"` and it will find them, compile them, and wait for changes to any of them:

```
tsc -w
```

I can tell by the fact that the compiler isn't showing any errors that it's working, but just for the heck of it I can go change my `app.ts` file a bit to test it out...

app.ts

```
let todo: string = 'Dryclean cape';
```

And I can go back to the command prompt and see from the output that everything is still working fine.

And that is how you create and configure your very first TypeScript project! Be sure to hold on to this folder because we're going to be using it to build our sample application using TypeScript throughout the rest of this book. Also, make sure to keep that `"tsc"` task running in the background the whole time you're working on the code samples in this book, because I will be!

Now that you're all set up, why don't you head on to the next chapter where I'll show you all of the latest ECMAScript features that TypeScript makes available to you so that you can begin using this new project to implement tomorrow's ECMAScript language features today.

4. ECMAScript 2015 Features

In the last chapter, I mentioned that in addition to adding static typing to the JavaScript language, TypeScript also provides the ability to leverage a handful of ECMAScript 2015 and next-generation ECMAScript features that aren't yet widely supported and compile them down to ES5-compatible JavaScript that will run in all of today's browsers. In this chapter I'm going to walk through each of those ECMAScript features in detail, showing you how each feature works and why you might want to use it in your application.

Notice how I said "might want to use it" – as you read these next few sections, you may notice that I'll use the term "syntactic sugar" quite a bit. When I say that something is "syntactic sugar", what I mean is that they do not change the functionality of your application at all. They are simply nice, clean ways to express logic that would otherwise take many more keystrokes to write out using the previous ES5 language syntax.

Keep in mind, however, that even though these features are a very nice way to write clean, concise JavaScript - and I will be using most of them in the sample code that I write throughout this book - nothing I'm about to show you is required in order to leverage TypeScript's static typing abilities. What's more, each of these features are independent of each other which means that you don't need to learn one feature in order to leverage another feature. Just learn the features that seem like they might have some value to you and ignore the rest.

So, if you are really eager to jump directly into the unique static typing approach that only TypeScript offers, you can feel free to skip any section in this chapter - or even the entire chapter - and then come back and read them later. They'll be here waiting for you!

Optional Parameters

The first ECMAScript 2015 feature I'll demonstrate is called **optional parameters**. Just as its name implies, the optional parameter syntax allows you to specify a default value for a given parameter in the case that a value is not explicitly passed in.

Take this function, which counts down from an initial number to a final number at a given interval, and it accepts all three of these values as parameters.

```
var container = document.getElementById('container');

function countdown(initial, final, interval) {

    var current = initial;

    while(current > final) {
        container.innerHTML = current;
        current -= interval;
    }

}
```

The nice thing about this function is that it is very customizable: since all of the variables are exposed as parameters on the function I am able to pass in any values I like for the initial value, the final value, and the interval.

However, let's say that as I start using this function throughout my application I find that I'm always counting down to 0 with an interval of 1. In other words, I'm almost always passing in 0 and 1 as the values for the last two parameters, like this:

```
countdown(10, 0, 1)
```

Notice how I said "almost always", meaning that there are still some cases where I do want to be able to specify how far down to count or how quickly to do it by passing in values for the last two parameters rather than simply hard-coding them.

This is the perfect scenario for optional parameters.

I can specify that a parameter is "optional" by simply adding "equals" after the parameter name and then specifying the default value that should be used if the caller omits that parameter:


```
1  function countdown(initial, final = 0, interval = 1) {  
2  
3      var current = initial;  
4  
5      while(current > final) {  
6          container.innerHTML = current;  
7          current -= interval;  
8      }  
9  
10 }
```

If we take a look at the ES5-compatible JavaScript code that is generated - especially lines 3 & 4 - we can see that the compiler has added a check for each parameter to see if the caller of the function provided a value for that parameter by looking to see if the value is “void” or not. When it sees that the parameter was omitted, it initializes the value of that variable to the default value.

With this change in place, I can now call the countdown function by only passing in one parameter:

```
countdown(10)
```

And when I do this, the final parameter will default to 0 and the interval parameter with default to 1.

Then, in the cases where I want to specify the final parameter, I can do so:

```
countdown(10, 4)
```

And the interval parameter will still default to 0.

Of course, I can still call the function by passing in all three parameters, just as before:

```
countdown(10, 4, 2)
```

As this simple example shows, optional parameters and default values are a nice bit of syntactic sugar that can really help make your code cleaner and easier to read.

Template Strings

In this section I'm going to demonstrate one of my favorite ECMAScript features: **template strings**. Template strings are an incredibly simple but surprisingly powerful bit of syntactic sugar to help you in those scenarios when you want to construct a string that includes the values of your variables.

Here's a really simple example:

```
var todo = {
  id: 123,
  name: "Pick up drycleaning",
  completed: true
}

var displayName = `Todo #${todo.id}`;
```

The first thing to notice is the syntax. In JavaScript, there are two ways to define a string literal value: you can wrap the text in single or double quotes. In order to define a string template you use the backtick symbol - that one right next to your 1 key that you've never used for anything else. Inside the template you can write whatever string literal you want, just like a normal string literal, except for when you want to inject a value - that's when you open up an expression by typing "dollar open bracket". You close the expression by closing the bracket. Inside of those brackets you can put whatever JavaScript expression you like, and the result will be rendered at that place in the string.

If you take a look at the ES5 that this gets compiled down to, it makes a whole lot more sense:

```
"Hello, " + name;
```

Looking at this compiled code with such a simple example also makes this syntax seems a little silly, but it gets a whole lot more interesting when you consider a more real-world example. For instance, let's assume we want to use this chunk of HTML as a template to render the details of a Todo:

```
<div todo='[[Todo ID]]' class="list-group-item">
  <i class="[[ If Todo is complete, then "hidden" ]] text-success glyphicon gl\
yphicon-ok"></i>
  <span class="name">[[Name]]</span>
</div>
```

Previously, I might implement this template by reading this from the DOM and then doing a search-and-replace operation to insert the various bits of logic and data such as the Todo ID, Todo Name, and dynamically applying the "hidden" class to hide the "success" icon on the second line when the todo is not yet completed.

So, let's avoid all of that by converting this into a string template instead.

The first thing to do is wrap the HTML in backticks.

```

container.innerHTML = `
    <div todo='[[Todo ID]]' class="list-group-item">
        <i class="[[ If Todo is complete, then "hidden" ]] text-success glyphico\
n glyphicon-ok"></i>
        <span class="name">[[Name]]</span>
    </div>
`

```

Notice how the string template spans multiple lines. You don't have to do anything special, just hit enter and keep on typing.

Also, the fact that this syntax leverages the backtick symbol to open and close the template means that you're free to use the single- and double-quotes inside of the template, just like any other character.

Then, rather than doing those search-and-replace operations, we can insert the values that we want directly into the template using the “dollar-bracket” syntax, like this:

```

var todo = {
    id: 123,
    name: "Pick up drycleaning",
    completed: true
}

container.innerHTML = `
    <div todo='${todo.id}' class="list-group-item">
        <i class="[[ If Todo is complete, then "hidden" ]] text-success glyphico\
n glyphicon-ok"></i>
        <span class="name">${todo.name}</span>
    </div>
`

```

One of the cool things about string templates is that you don't have to restrict yourself to simple variable replacements. You can actually put more complex statements in that will be evaluated.

For example, I can figure out whether or not to render the “hidden” class on the icon element dynamically by introducing a conditional statement right there in the expression, like this:

```

container.innerHTML = `
    <div todo='${todo.id}' class="list-group-item">
        <i class="${ todo.completed ? "" : "hidden" } text-success glyphicon gly\
phicon-ok"></i>
        <span class="name">${todo.name}</span>
    </div>
`

```

When the value of `todo.completed` is true, the expression will evaluate to an empty string and no CSS class will be applied; when the value is false, the expression will output the string “hidden”, which will be rendered along with the rest of the CSS classes in the template.

app.js

```

container.innerHTML = "\n\t    <div todo='" + todo.id + "' class=\"list-group-it\
em}\">\n\t        <i class=\"[[ If Todo is complete, then \"hidden\" ]] text-suc\
cess glyphicon glyphicon-ok\"></i>\n\t        <span class=\"name\">" + todo.name\
+ "</span>\n\t    </div>\n\t";

```

Once again, inspecting the compiled JavaScript reveals that - even with inline conditional statements and everything - this syntax eventually just ends up as a series of concatenated strings and expressions.

Nonetheless, string templates remains one of my favorite ECMAScript 2015 features: I find myself using it more every day, and I expect you will, too.

Let and const

In this section, I'm going to introduce you to two more ways to declare variables in addition to using the `var` keyword: **the `let` and `const` keywords**.

In order to really appreciate the value of the `let` and `const` keywords, it's important to really understand how the `var` keyword works and how it has a few undesirable behaviors that can be really troublesome to JavaScript developers.

To demonstrate, take a look at this simple for loop which creates a variable inside the for loop scope, then tries to access that variable outside of the scope:

```
for(var x = 0; x <= 5; x++) {  
    var counter = x;  
}
```

```
console.log(counter);
```

If I execute this code in the browser, I can see that prints out the value of the variable “counter” and guess what - it works because I can see the number 5 printed to the console!

If you come from a language such as Java or C# or many other similar languages, you'd think this was crazy because you'd expect that everything in that for loop - everything inside those brackets - would be contained in its own scope so that it couldn't be accessed by the outside. As this demo illustrates, that's clearly not the case in JavaScript.

Now let's replace the `var` keyword with the `let` keyword instead:

```
for(var x = 0; x <= 5; x++) {  
    let counter = x;  
}
```

```
console.log(counter);
```

With this change, the editor immediately alerts us to the error on that last line that references the counter variable, telling us that it can't find a variable named `counter`. What's more, if you executed this code in a browser using strict mode, you'd actually get an error at runtime in your browser that the counter variable is not defined – all of this exactly what you'd expect if you're used to the block scope in languages other than JavaScript.

Where can you use the `let` keyword? Well, you can - and you should - feel free to use the `let` keyword pretty much anywhere that you're currently used to using the `var` keyword today. Everywhere, of course, except the places where you're actually depending on JavaScript's interesting behavior that I just demonstrated... but if you're doing that, you might want to rethink that approach, too!

Now that you've seen the `let` keyword in action, let's take a look at its sibling, `const`. As you might expect if you've used this keyword in other languages such as C#, the `const` keyword means to create a variable and initialize it to a certain value, then never let that value change.

To demonstrate, I'll change the `let` keyword from the previous example to `const` instead. Everything continues to behave as before (including the fact that the `counter` variable is not accessible outside of the `for` loop).

```
for(var x = 0; x < 5; x++) {  
    const counter = x;  
    counter = 1;  
}
```

Then, if I try to assign the `counter` variable to another value, the editor yells at me and tells me I can't do that.

```
for(var x = 0; x < 5; x++) {  
    const counter = x;  
    counter = 1;  
}
```

Likewise, if I choose to ignore the editor warning and run this code in the browser in strict mode, the browser will give me a runtime exception that I can't assign a value to a constant variable after it's been initialized.

The way that JavaScript deals with variables has always been a sticking point with a lot of developers and it's certainly been the source of plenty of bugs in my development career. But now that we have keywords such as `let` and `const` at our disposal, we can use them to better express our real intent and these bugs will certainly become a thing of past.

For..of Loops

In this section I'm going to show you a concise new way to loop over arrays of objects. Prior to ECMAScript 2015, you had to use a clunky `for/in` syntax in order to iterate over an array.

```
1 var array = [ "Pick up drycleaning", "Clean Batcave", "Save Gotham" ];
2
3 for(var index in array) {
4     var value = array[index];
5     console.log(`${index}: ${value}`);
6 }
```

As this example shows, when you use the `for/in` keywords to iterate over an array, you're actually iterating over the index values - in other words, 0, 1, 2, etc. So, in order to actually get the values out of the array you need to use those indexes to reference that values in the array.

That's what's happening in line 4 of this example - I'm using the `index` to grab the value out of that index in the array.

Then on line 5 I log the value to the console.

Running this in a browser helps to see what's going on a little better:

```
0: Pick up drycleaning
1: Clean Batcave
2: Save Gotham
```

That's the `for/in` keyword - the new feature in ECMAScript 2015 is `for/of`, which iterates over the values of the array directly and cuts out the middle man. Here's that same example using the `for/of` syntax:

```
var array = [
    "Pick up drycleaning",
    "Clean Batcave",
    "Save Gotham"
];

for(var value of array) {
    console.log(value);
}
```

If I run this snippet in the browser, I can see that it just prints out all of the values without having use the index to pull them out of the array.

```
Pick up drycleaning  
Clean Batcave  
Save Gotham
```

The `for/of` syntax is just one more example of syntactic sugar: since you always had the `for/in` syntax at your disposal and it doesn't allow you to do anything that you couldn't do before, but it does allow you to write cleaner, more concise code and over time may save you plenty of lines of code in your application.

Arrow Functions

In the “[let and const](#)” section I showed you how JavaScript can have some wacky behavior when it comes to dealing with variables and their scope. However, there is perhaps no wackier behavior in JavaScript than the `this` keyword. Rather than give a big long technical explanation as to why, let me just show you an example in which the `this` keyword introduces some unexpected behavior:

```
1  var container = document.getElementById('container');
2
3  function Counter(el) {
4
5      this.count = 0;
6
7      el.innerHTML = this.count;
8
9      el.addEventListener('click',
10         function () {
11             this.count += 1;
12             el.innerHTML = this.count;
13         })
14  }
15
16  new Counter(container);
```

This snippet looks a little long, but really all it does is executes the function on line 10 which increments the counter every time the user clicks the element by incrementing the field “`this.count`” on line 11.

Just by looking at this example, I’d expect that the count would start at 0, and then every time I clicked on the element, it’d add one more to the count: 1, 2, 3, and so on. But, I have a little surprise for you. Let’s switch to the command line and execute the “`lite-server`” command to run our site in a browser and try it out...

```
lite-server
```

When the browser opens, I can see the value 0 printed. So far, so good... But, if I click...

Now I see NaN which means that the value has been lost!

Long story short, the reason that the value got lost is because of the call to `this.count` on line 11. Even though it looks like `this.count` refers to the `Counter` object that this function lives in, when it’s executed as an event handler for a browser click event, the `this` keyword actually refers to the global browser scope, which is definitely not the same thing.



Closure fail

If you're like me, this scenario has caused you many headaches in the past. And, the way you need to work around it is to first save a reference to the `this` object outside of the function where you still have access to it, and then reference that temporary variable instead of the `this` keyword inside of the function, like this:

```
let _this = this;

el.addEventListener('click',
  function () {
    _this.count += 1;
    el.innerHTML = _this.count;
  })
```

Now, this works, but personally I've always hated this approach because it just seems awkward and unnecessary to save a reference to `this` just to be able to access it later on.

Luckily, ECMAScript 2015 introduces a great new feature called **arrow functions** to help address this very problem, and the syntax of this feature is actually really simple and it may even be familiar to you already.

If you've used other languages such as C# and Java, you've probably already used an arrow function before – in those languages they're called “lambda expressions”.

You can convert pretty much any JavaScript function into an arrow function by simply removing the `function` keyword from the front and then inserting “equals, greater-than” (or “arrow”, hence the name of the feature) after the parameter list. And.. That's it!

So, if I undo the ugly hack that I just introduced, I can rewrite the same code like this:

```
el.addEventListener('click',
  () => {
    this.count += 1;
    el.innerHTML = this.count;
  })
```

And if I look at the generated JavaScript I can see that it's actually creating that ugly “`_this`” variable for me in the background to create ES5-compatible JavaScript.

Now when I run this code in the browser and click on the DOM element, I can see that the counter is incremented, meaning that the call to “this.count” actually points to the variable that I expect it to.

Fixing the reference to the `this` keyword is the only functional change that this syntax provides, but it does offer a little more syntactic sugar as well.

For example, this example doesn’t require it, but if I wanted to pass in parameters to the function, I can just put them in the parentheses - just like a normal function signature, only without the “function” keyword.

```
el.addEventListener('click',  
  (event) => {  
    this.count += 1;  
    el.innerHTML = this.count;  
  })
```

And - even better - if the body of my arrow function had only one expression, like this:

```
el.addEventListener('click',  
  (event) => {  
    el.innerHTML = (this.count += 1);  
  })
```

I could do away with the brackets entirely and just execute that one expression inline.

```
el.addEventListener('click', (event) => el.innerHTML = (this.count += 1))
```

Now, isn’t that nice and concise?

Note that arrow functions expressions written in this way actually return the output of the expression. In other words, let’s say I wanted to loop through an array of items using the `Array.filter()` method.

I could write the arrow function across multiple lines like this:

```
var filtered = [ 1, 2, 3 ].filter((x) => {  
  return x > 0;  
});
```

Or, get rid of the “return” keyword and just compact it into one expression, like this:

```
var filtered = [ 1, 2, 3 ].filter((x) => x > 0);
```

Also note that in this case - when the arrow function accepts only one parameter - I can even omit the parentheses around the parameter name:

```
var filtered = [ 1, 2, 3 ].filter(x => x > 0);
```

I don't know about you, but I like that syntax a whole lot more than the verbose alternative. In fact, I mentioned earlier in the chapter that template strings were my favorite ECMAScript 2015 feature, but actually I'm not too sure, because I really love arrow functions. And I'm willing to bet that once you've used them for a while you'll love them, too!

Destructuring

In this chapter I've been demonstrating all of the language features that are new in ECMAScript 2015. The features I've shown so far - features such as arrow functions and string templates - are the ones that I use on a regular basis in just about any TypeScript file that I work in. The last few features I'm about to show, however, I tend to use a whole lot less (if at all) and I expect that you'll have the same limited need for them as I do. Nevertheless, the language offers them, so it's probably worth your time to learn them for the occasions that they may be useful to you.

The first of these features is something called **destructuring**, which is the ability to assign values to multiple variables from a single object with a single statement. The easiest way to think of destructuring is that it is the reverse of creating a bunch of variables and combining them into an array.

In its simplest form, destructuring looks like this:

```
var array = [123, "Pick up drycleaning", false];  
var [id, title, completed] = array;
```

Destructuring is the thing that happens in line 2, where we assign the values of the variables `id`, `title`, and `completed` all at once using the values contained in the array. The first value in the array gets assigned to the first variable, the second value to the second variable, etc.

It's actually a lot easy to understand if you just take a look at the generated JavaScript to see exactly what this translates to:

```
var array = [123, "Pick up drycleaning", false];  
var id = array[0], title= array[1], completed = array[2];
```

That means that if I were to execute this code in a browser:

- the variable `id` would contain the value 123
- the variable `title` would contain the value "Pick up drycleaning"
- And the variable `completed` would contain the value `false`

The benefits of this syntax might not be immediately obvious with this example, so let me show one example where the destructuring syntax is a little bit cleaner than the ES5 alternative.

Let's say that you have two variables, "a" and "b" and you want to flip their values. Using ES5, You can't just assign the value of variable "b" to variable "a" because you'd lose the value of variable "a" in the process. So, you'd have to introduce a third temporary value like so:

```
var a = 1;
var b = 5;

var temp = a;
a = b;
b = temp;
```

With destructuring, this becomes simple:

```
var a = 1;
var b = 5;

[a, b] = [b, a];
```

Destructuring doesn't just work on arrays - it actually works with objects as well, although with a slightly different syntax. In the case of objects, the values aren't assigned by their location in the object, but by matching the name of the property with the name of variable.

For example, let's look at a more real-world object:

```
var todo = {
  id: 123,
  title: "Pick up drycleaning",
  completed: false
};
```

Given this todo object, I can easily assign its three properties - id, title, and completed - to variables of the same names:

```
var { id, title, completed } = todo;
```

Looking at the generated code, you can see that this gets broken down into individual assignments, based on the variable name:

```
var id = todo.id, title= todo.title, completed = todo.completed;
```

What's more, it doesn't matter at all what order the properties are defined in. I happened to assign them in the same order that they were declared on the object in this example, but that doesn't have to be the case. I can assign them in any order I'd like.

For instance, this code will achieve the same exact outcome:

```
var { completed, title, id } = todo;
```

And, if I try to assign a variable with a name that doesn't match the name of a property on the object, TypeScript will yell at me.

For instance, if I try to assign a new variable named `createDate`...

```
var { completed, title, id, createDate } = source; // object has no property "c\
reateDate"
```

TypeScript will tell me that the object doesn't have a property called "createDate".

Note that TypeScript will still generate the code anyway, just as it would if it were an ECMAScript 2015 compiler without any type information:

```
var id = todo.id, title= todo.title, completed = todo.completed, createDate= tod\
o.createDate;
```

And here's another cool thing about destructuring: it even works when mapping function return values. In other words, all of the examples I've shown have involved declaring an object and that pulling values from that object, but that object could just as well be the result of a function call rather than a declared object.

For example, if I wrap that `todo` object in a function and return it from that function like this...

```
function getTodo(id) {
  return {
    id: 123,
    title: "Pick up drycleaning",
    completed: false
  };
}
```

Then I can still initialize those same variables by calling that function and piping the response right into the declaration, like this:

```
var { completed, title, id } = getTodo(123);
```

Now, you might be thinking, "That's nice, but my variable names usually don't line up that well! What if I wanted to assign the property to a variable with a different name?" Well, that's pretty simple, too - just create a mapping from the property name that you want to pull from by adding a colon after the name of the property, then supply the name of the variable you want to map to.

For example, if you wanted to map the value of the property named `completed` on the source object to, say, a variable named `isCompleted`, you'd just use this syntax:

```
var { completed: isCompleted, title, id } = getTodo(123);
```

All of this is nice, but perhaps the most effective use of destructuring is to reduce a long list of method parameters down into a single option that contains all of those parameters as properties instead.

To demonstrate, let's revisit the countdown function from the [Optional Parameters](#) section:

```
function countdown(initial, final = 0, interval = 1) {  
  
    var current = initial;  
  
    while(current > final) {  
        console.log(current);  
        current -= interval;  
    }  
  
}
```

Now, this function has only got 3 parameters, which isn't too many, but let's combine them all into an object anyway, just to see this approach in action. In order to implement this approach manually, I'd have to write this code:

```
function countdown(options) {  
  
    var options = options === undefined ? {} : options,  
        initial = options.initial === undefined ? 10 : options.initial,  
        final = options.final === undefined ? 0 : options.final,  
        interval = options.interval === undefined ? 1 : options.interval;  
  
    var current = initial;  
  
    while(current > final) {  
        console.log(current);  
        current -= interval;  
    }  
  
}
```

Notice how when I convert the parameters into an object, I keep the default values that I applied in the previous section, but I have to do it the long way, playing it safe by checking each property –

and the options object itself – to see if they are defined before trying to access them. Then, fall back to the default value if they are not defined.

A lot of people prefer to pass around objects rather than lists of parameters, but clearly in some cases it can make the code very unreadable!

Luckily, we can apply the destructuring syntax directly to the object parameter itself, allowing us to use this design pattern without having to write all of that code, like this:

```
function countdown({ initial, final, interval }) {  
  
    var current = initial;  
  
    while(current > final) {  
        console.log(current);  
        current -= interval;  
    }  
  
}
```

I can even pull in that current variable by assigning the initial property to two different variables:

```
function countdown({ initial, final, interval, initial: current }) {  
  
    while(current > final) {  
        console.log(current);  
        current -= interval;  
    }  
  
}
```

With this in place, the only thing left to do is add those default values back into the mix, which I can do with the same syntax I used before:

```
function countdown({ initial, final: final = 0, interval: interval = 1, initial:\
current }) {

    while(current > final) {
        console.log(current);
        current -= interval;
    }

}
```

And there it is: the destructuring syntax in all its glory! Of course, while some people consider this code much easier to read and maintain than a list of parameters, others think it looks utterly horrible and wouldn't consider using it. Regardless of how you feel about it, it's another tool in your ECMAScript 2015 toolbox that is there when you need it!

The Spread Operator

In this section I'm going to show you the **spread operator** - another ECMAScript 2015 function that I tend to use pretty infrequently, but in the right cases it can make some really elegant-looking code.

Rather than start off with the syntax that the spread operator uses, first I'll demonstrate one of the best use cases for it: creating a function that takes any number of arguments.

To do this using ES5 syntax, you'd write a function like this:

```
function add() {  
    var values = Array.prototype.splice.call(arguments, [1]),  
        total = 0;  
  
    for(var value of values) {  
        total += value;  
    }  
  
    return total;  
}
```

This function accepts any number of arguments and then uses the very long-winded call to `Array.prototype.splice.call()` to convert them into an array that can then be iterated over.

Other than these first few awkward lines of code, the rest of the function is actually pretty straight forward: loop through the argument values (using the new ECMAScript 2015 `for/of` syntax I showed you earlier) and add them all together, then return the total.

I say that these lines of code are awkward for two reasons:

1. It's very long and somewhat difficult to determine at a glance why we are calling a method on the Array prototype object (by the way, it's because the arguments object isn't actually an array, but we can trick the `Array.splice` method to thinking it is if we call it this way)
2. At a glance, it looks like this function doesn't accept any arguments since the signature is empty. Only once you actually get into the implementation do you see the reference to the arguments object.

The spread operator offers us a solution to both of these problems.

Instead of converting the argument list into an array of values and assign it to the variable called `values` within the function, we simply use the variable named `values` as the argument and then apply the spread operator - represented by three periods - in front of it, like this:

```
function add(...values) {  
    var total = 0;  
  
    for(var value of values) {  
        total += value;  
    }  
  
    return total;  
}
```

With the spread operator in place, everything is much cleaner and we still get the original behavior of being able to pass in any number of arguments we like (including no arguments at all, in which case the array will just be empty)!

```
add(1, 2, 3, 4, 5)
```

And, while the spread operator has to be the last argument in the list, it doesn't have to be the only argument - we can still define other arguments in front of it, like this:

```
function calculate(action, ...values) {  
    var total = 0;  
  
    for(var value of values) {  
  
        switch(action) {  
            case 'add':  
                total += value;  
                break;  
  
            case 'subtract':  
                total -= value;  
                break;  
        }  
    }  
  
    return total;  
}
```

Here I've renamed and refactored this method to handle multiple operations, depending on the value of the first argument - either 'add' or 'subtract'. I can then call it like this:

```
calculate('subtract', 1, 2, 3, 4, 5)
```

Cleaning up argument lists isn't the only thing that the spread operator is for. You can use it to work with arrays, as well. For instance, say you have one array:

```
var source = [ 3, 4, 5 ];
```

...and you want to inject it into the middle of another array, like this:

```
var target = [ 1, 2, /* Insert here */, 6, 7 ]
```

You can use the spread operator just as you would any other value in the array, except instead of adding one value to the target array, it will expand that source array to add all of its values, like this:

```
var target = [ 1, 2, ...source, 6, 7 ]
```

After we execute this line, the target variable will be an array populated with all the values in the order they were defined:

```
target // 1, 2, 3, 4, 5, 6, 7
```

And, just like we used the spread operator to remove a call to the splice method on the Array prototype, we can replace similar code used to concatenate two arrays together which would have been much more awkward using the ES5 syntax:

```
var list = [ 1, 2, 3 ];  
var toAdd = [ 4, 5, 6 ];  
  
Array.prototype.push.apply(list, toAdd);
```

Where we previously used the spread operator in place of the call to `Array.prototype.splice`, we can replace this call to push with a simple:

```
list.push(...toAdd);
```

And that's the spread operator in action! It may not be a language feature that you use every day, but it can be a great help if you are looking to implement a method with an unlimited number of arguments or clean up code that does a lot of array manipulation.

Computed Properties

In this last section of the ECMAScript 2015 feature chapter, I'm going to show a new language feature that I, quite frankly, have not yet felt the need to use in my applications. However, it is a feature that can come in quite handy if you are implementing an application that is really taking advantage of JavaScript's dynamic nature. This feature is called **computed properties** and it allows you to define a property on an object with a name that is computed dynamically at runtime.

To demonstrate why you might want to use a computed property, consider the following situation:

```
var support = {
  'os_Windows': isSupported('Windows'),
  'os_iOS': isSupported('iOS'),
  'os_Android': isSupported('Android'),
}

function isSupported(os) {
  return Math.random() >= 0.5;
}
```

An object that defines a property for the three operating systems that the application understands - Windows, iOS, and Android - and sets the value of that property to a boolean indicating whether that operating system is supported or not.

(Don't worry about how we're figuring out whether an OS is supported or not - it's not important. Let's just say it's not guaranteed to be the same all the time...)

Notice how each property is prefixed with the string "os_". Let's say that this is an important convention and I want to be sure that is applied the same way to every property name. Of course, we can't just use string concatenation to add the prefix to the beginning of every property... but we can if we use computed properties!

The first thing we need to do is convert the property names into computed properties by first wrapping them in brackets like this:

```
var support = {
  ['os_Windows']: isSupported('Windows'),
  ['os_iOS']: isSupported('iOS'),
  ['os_Android']: isSupported('Android'),
}
```

Now that they are computed properties, we can turn them into expressions, like adding the value of the OS prefix variable to them:

```
const osPrefix = 'os_';

var support = {
  [osPrefix + 'Windows']: isSupported('Windows'),
  [osPrefix + 'iOS']: isSupported('iOS'),
  [osPrefix + 'Android']: isSupported('Android'),
}

function isSupported(os) {
  return Math.random() >= 0.5;
}
```

With that in place, we can run this code and inspect the `support` object to see the defined properties:

```
support // {os_Windows: false, os_iOS: false, os_Android: true}
```

This example may seem a bit contrived because, quite frankly, the cases in which you use computed properties tend to be a bit more complex than is suitable for a simple easy-to-understand tutorial. However, the next time you are dealing with particularly dynamic code and struggling to find the best way to create properties with dynamic names, you may find computed properties to be quite useful in helping you do that.

This concludes the chapter on the new ECMAScript 2015 features that TypeScript allows you to start using now, regardless of whether or not they have been implemented in your target browsers. TypeScript is far more than just a simple ECMAScript 2015 transpiler, however, and the real magic lies in its advanced typing system. Check out the next chapter where I'll show you how to take advantage of TypeScript's powerful static type support.

5. Type Fundamentals

Introducing JavaScript Types

In the introductory chapter I explained that TypeScript brings static typing capabilities to the JavaScript language. In this chapter I'll show you the syntax that TypeScript uses to apply type information to your code, and even how to define the complex data structures that your application depends on.

Before I get into complex data structures, however, let's just start by calling out all the types that JavaScript offers you right out of the box, since these are at the core of everything we do in JavaScript and will be the building blocks that we will use when defining our own complex types later on.

Though it may sound sparse, the ECMAScript 5 specification only defines six data types.

- `boolean`
- `number`
- `string`
- `null` / `undefined`
- `object`

The first three are very popular primitive values: `boolean`, `number`, and `string`.

These types are immutable, meaning once one of these values are defined, it cannot be changed.

Then there are two special types: `null` and `undefined` – that's right, JavaScript has two different ways to refer to nothing! And, finally, there's the type that encompasses all other JavaScript types: `object`.

Objects

You can think of JavaScript objects as simply a dictionary of properties, identified or indexed by string keys. And, unlike the other data types, object properties may be changed after they're defined - you can even add new properties on the fly!

Key	Value
"name"	"Jess Chadwick"
"age"	21
"gender"	"Male"

The fact that ES5 only defines six data types may surprise you since you've probably used a lot of other types such as `Date`, `Regex`, etc., but those are all just different kinds of objects.

But, perhaps the most important object types are **functions** and **arrays**.

Functions

Functions are regular objects in that they are a collection of properties, but the aspect that sets them apart from all other objects is that they contain logic that can be executed.

Key	Value
"version"	1.2.3
<pre>function add(x, y){ return x + y; }</pre>	

Function object

Because of this important ability, functions play an incredibly crucial role in most any JavaScript application so we will be dedicating plenty of time throughout this book to working with functions in a strongly-typed manner.

Arrays

The other special kind of object type is an array. Arrays are special kinds of objects containing a collection of values, with each value represented by an integer key.

Key	Value
"name"	"Jess Chadwick"
0	"Pick up drycleaning"
1	true
2	1.2.3

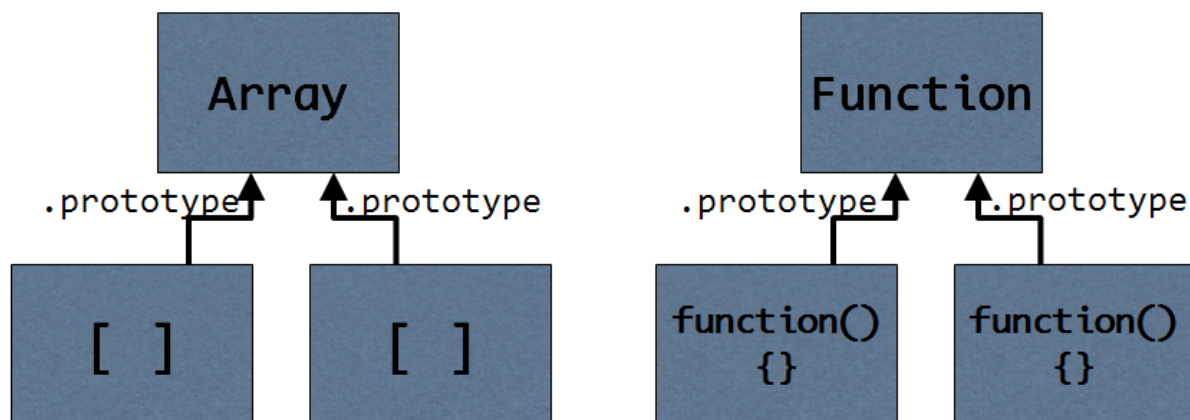
Object Prototype

Both functions and arrays are also enhanced with additional methods that pertain to working with functions and collections, which they receive from their **prototype**.

If this prototype sounds important, it is. As an object-oriented language, JavaScript offers the ability to share properties and behavior between different types through inheritance - a kind of inheritance called **prototypical inheritance**.

Prototypical inheritance simply means that an object is defined that contain the properties and behavior to be shared, and when new instances of that type are created, JavaScript links those properties and behaviors to the new instance.

Prototypical Inheritance



Prototypical inheritance

This approach to creating objects is important - and I will definitely get more into prototypes and prototypical inheritance in later chapters - but not all JavaScript objects must be created from a constructor that has a prototype.

Object Literal

There's an even simpler way to define a JavaScript object called **object literals**.

An object literal is nothing more than a way to define and instantiate an object all at the same time using a very simple and straight-forward syntax.

For example, here is the most basic object literal, representing a completely empty object:

```
{ }
```

Notice how the brackets are used to indicate the beginning and end of the object definition. Then, we can add properties to the object by defining key/value pairs split with a colon in the middle, and those values can be any type, like so:

```
{
    name:  "Fido",
    species: "Dog",
    age:  5
}
```

In this example, I've defined three different properties, and assigned them all values. I've also separated all of the values with both a comma and a line break, though whitespace doesn't matter when defining object literals, which means that I could remove all the spaces and line breaks and end up with this:

```
{name:"Fido",species:"Dog",age:5}
```

Yes, it's true that that this would be exactly the same object... but it's far less readable, so let's add them back before I get a headache!

```
{
    name:  "Fido",
    species: "Dog",
    age:  5
}
```

Now that I can see the object clearly again, let's go the next step and give this object some behavior by defining a function. As I mentioned earlier, functions are just regular values so I can define and assign them in exactly the same way that I assign other values:

```
{
    name:  "Fido",
    species: "Dog",
    age:  5,
    speak: function() { console.log('Woof!'); }
}
```

Again, whitespace doesn't matter: I can put the whole function definition on one line, or spread it across multiple lines - it's up to me.

```
{
  name: "Fido",
  species: "Dog",
  age: 5,
  speak: function() {
    console.log('Woof!');
  }
}
```

Now that I have this object, I can do things with it, like pass it to a function:

```
function makeTheAnimalSpeak(animal) {
  animal.speak();
  animal.speak = 1;
}

var animal = {
  name: "Fido",
  species: "Dog",
  age: 5,
  speak: function() {
    console.log('Woof!');
  }
};

makeTheAnimalSpeak(animal);
```

Notice how the function just calls the `speak()` method on the object that it's given. This is a great example of duck typing in action – the function doesn't check to see if the object is a certain type, or if the method exists before calling it – it just calls the method and assumes that the code that passed the object in knew what they were doing.

In addition to being a great example of duck typing in action, it's also a great example of a place where static typing can really help, and I'll build upon this simple example throughout this book to show you how.

In this section, I gave you an overview of the fundamental types that JavaScript provides, and a quick demonstration of how to create, instantiate, and populate your own types on the fly using object literals. In the next section, I'm going to talk about how TypeScript uses a concept called **type inference** to analyze this code just as it is to start giving you feedback and help you write better code without having to do anything more than what you see here.

Understanding Type Inference

In the previous section, I showed you how to create, instantiate, and populate a new type of object on the fly using the object literal syntax. In this section, I'm going to talk about how TypeScript's powerful static analysis capabilities can give you significant insights into your code using a concept called **type inference**... all while still using standard JavaScript.

Static analysis means that - regardless of whether you explicitly define types or not - TypeScript looks through your code, doing its best to guess (or “infer”) what type any given object could be.

In other words, even though this is just plain old JavaScript, just using the code that is written TypeScript is able to figure out that the type of the `name` and `species` properties are strings, the `age` property is a number, and the `speak` property is a function. Heck, it even knows that the `animal` variable is a type that has 4 properties: `name`, `species`, `age`, and `speak`! It knows all of this because it can inspect the assignment of each of these properties to see what type of value was assigned to them.

Even though I never asked it to, TypeScript will even start to enforce these inferred types, yelling at me if I try to set the `name` property (which is a string) to a number value.

```
var animal = {
  name: "Fido",
  species: "Dog",
  age: 5,
  speak: function() {
    console.log('Woof!');
  }
};

animal.name = 1;
```

The same thing goes for return types, too. If I create a function that returns a value with a type that TypeScript can determine through static analysis - such as a string, a number, or even an object literal - TypeScript can use that information wherever that return value is used.

For example, let's say I add a function to calculate the animal's current age by subtracting the year it was born from the current year:

```
function calculateAge(birthYear) {
  return Date.now() - birthYear;
}
```

When I hover over the method, I can see that TypeScript has figured out that the type of the return value will be a number. What's more, it's figuring that out regardless of the fact that I didn't specify

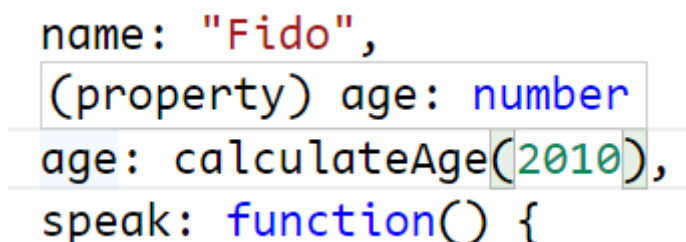
the type of the `birthYear` parameter!

TypeScript is smart enough to know that the result of `Date.now()` is a number, and any time you subtract something from a number, you're going to get a number.

Likewise, I can change the assignment of the `age` property to call this function instead:

```
—— age: 5,  
    age: calculateAge(2010),
```

And, when I do, I can hover over the property to see that TypeScript is assigning the number type from the function's return type.



```
name: "Fido",  
(property) age: number  
age: calculateAge(2010),  
speak: function() {
```

Type inference in action

Now think about the effect this would have if this function were called by many other places throughout the code. Knowing the return type of this code would allow TypeScript to make better inferences about the code that calls this function, which may then trickle into the code that calls those functions, and so on.

Because of this, strategically applying type information to common, low-level functions and API calls is a great way to start getting the most out of TypeScript's type inference in your existing JavaScript codebase without having to write any TypeScript-specific code.

While TypeScript's ability to infer types may be impressive, it does have its limits.

For example, look at this function which takes in two parameters and adds the value of their length properties.

```
function totalLength(x, y) {  
    var total = x.length + y.length;  
    return total;  
}
```

Since every object has a `length` property that is a number value, this function should work with pretty much any inputs and return a number every time. However, even though we can understand it relatively easily, TypeScript has no idea what to do with it.

Whenever this situation occurs – whenever TypeScript doesn't have enough information to definitively figure out what the type of a given object is – it simply gives up and says that it's the most unrestrictive type of all: the `any` type.

The `any` type is the most dynamic and unrestrictive type available - in other words, the default dynamic type behavior of JavaScript - and that means that anything goes: you can call it as a function, set its value to one type, and then go ahead and set that same object to a value of another type.

Working with dynamic types like this can be quite powerful - in fact, it's one of the reasons that so many people love JavaScript. However, whenever you switch into the `any` mode, you lose any and all help that TypeScript provides by enforcing types in your code, even something as common as the `length` property.

That means that TypeScript isn't even able to offer us even the most basic static typing protection to warn me if I spell a property name like `length` wrong:

```
var total = x.length + y.length;  
var total = x.length + y.lengt;
```

Sometimes a truly dynamic object may be what you want, but it's important to keep in mind that TypeScript's type checking exists to help you avoid mistakes in your code so most of the time you're going to want to work with TypeScript to give it the information it needs and avoid the `any` type as much as possible. Reserving the `any` type only for cases when you're actually working with a dynamic object will make your life much easier.

In this section, I showed how to leverage TypeScript's type inference to help you write better JavaScript without introducing any TypeScript syntax, but we ended up in a situation where TypeScript didn't have enough information to safely infer the type of an object and needed a little more help. In the next section, I'll show you how to give TypeScript the information it needs

Specifying JavaScript Types

In the previous few sections I introduced you to the handful of primitive types that JavaScript offers out of the box and how TypeScript is able to figure out which type a variable is just by inspecting how you use it. At the end of the last section, I also showed you that TypeScript is only so smart and can't always figure out variable types without a little help. In this section I'll show you how to clear up the confusion by explicitly defining the type of variables, function parameters, and return values to take most of the guessing out of the way and give TypeScript all the information it needs to figure out the type of every object in your application.

Let's start off where we left in the last section, with this method, complete with the spelling error that I left it with:

```
function totalLength(x, y) {  
    var total = x.length + y.lengt;  
    return total;  
}
```

First I'll tackle that misspelling by specifying the type of the `y` parameter. I'll specify the type by adding a colon after the parameter followed by the name of the type. For this example, I'll say it's a string:

```
function totalLength(x, y: string) {  
    var total = x.length + y.length;  
    return total;  
}
```

As soon as I do that, TypeScript finally has enough information to notice that I have a mistake in my property name and let me know by highlighting it as an error. Now I can fix that and keep applying my type information.

```
var total = x.length + y.length;
```

Since I've already shown you how to define the type of a parameter, I'm going to hold off on giving the `x` parameter a type – plus for now I want to keep the `x` parameter as an any type so that TypeScript still doesn't know anything about it.

I want to leave it as the any type to show you that if I hover over the `total` variable I see that TypeScript is telling me the `total` variable has the type any as well. In other words, even though TypeScript now knows for a fact that the `y.length` property is a number type, because the `x` parameter is the any type, TypeScript has no idea what its properties are and can't validate that

the `length` property even exists, let alone what its type is. So, it just completely gives up and says that the whole equation returns a value of the `any` type.

Here's where I need to step in and give TypeScript enough information that it stops giving up and starts helping me again. Since I happen to know that the `total` variable is a number type, I've got two options here:

1. I can specify type information for the `x` parameter so that TypeScript knows that the `length` property is a number, just like I did with the `y` parameter... or...
2. I can explicitly tell TypeScript that the `total` variable is a number type by applying that same type syntax to the variable declaration itself, like this:

```
function totalLength(x, y: string) {  
  var total = x.length + y.length;  
  var total: number = x.length + y.length;  
  return total;  
}
```

In this case, I'm telling TypeScript: "Don't worry, I know that this equation will return a number, so just go ahead and enforce that from now on."

Something else interesting happens now, too. If I hover over the function signature, I can see that TypeScript has used its type inference to figure out that the function's return type is indeed a number. Now, that happens to be true, but what if I wanted to change that? What if I wanted to explicitly say that this function returned, say, a string instead of a number?

Well, that's easy, too: just apply the same type syntax to the function itself, right after the parentheses holding the function arguments:

```
function totalLength(x, y: string) {  
function totalLength(x, y: string): string {  
  var total: number = x.length + y.length;  
  return total;  
}
```

Of course, as soon as I do that, TypeScript now highlights the return statement, yelling at me that I'm returning a number instead of string – and that's a good thing, because it's wrong! So, let's go ahead and fix that by specifying the return type as a number instead of string, like it's supposed to be:

```
function totalLength(x, y: string): number {  
    var total: number = x.length + y.length;  
    return total;  
}
```

You might be wondering why I am choosing to explicitly declare the function's return type even though TypeScript is smart enough to figure it out. That's a good question, and the answer comes back to one of the primary reasons for using static types at all: the more explicit you are in your intent, the better the type system can help you find mistakes!

Yes, this method happens to return a number now, but if I come back later and change the code to return a string instead of a number, I will still have that check in place so that TypeScript can warn me that the code I just wrote doesn't match how I originally said the function should behave. Maybe I meant to change the type, or maybe I did it on accident - either way, TypeScript has notified me of the mismatch and I can then decide how to resolve it.

And, speaking of being explicit, let's wrap up this example by giving the `x` argument a type, too. This time, though, I'm going to get a little crazy and say that I expect the `x` argument to be an array of objects. Since I'm only interested in the length of the array, I don't really care what values are in the array, as long as it's an array... in other words, an array of any:

```
function totalLength(x: any[], y: string): number {  
    var total: number = x.length + y.length;  
    return total;  
}
```

With that in place, I've now explicitly defined all of the types on my function, including its return value, which should help keep me from adding bugs later on as well as help me to consume this method in the rest of my application.

Now, if you're scratching your head right now wondering why I'd make a method like this at all – a method that takes one array parameter and one string parameter and adds their lengths together – I don't blame you. In the next section, I'll show you how to use union types to allow either of the arguments to accept a string or an array.

Union Types

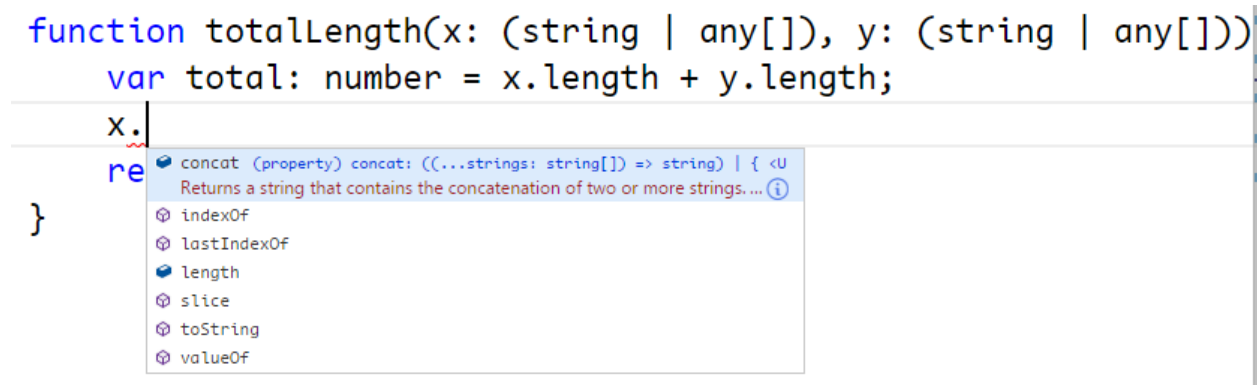
In the previous section, I showed you how to add type information to a function and ended up with the following function that takes two arguments - an array and a string - and adds their lengths together. I also mentioned that it's kind of weird that the first argument *must* be an array and the second *must* be a string. In this section, I'm going to show you how to use a TypeScript feature called **union types** to allow each of the arguments to accept either a string or an array.

To specify a union type, you simply wrap the type definition in parenthesis and use the pipe operator (also sometimes known as the OR operator) to list additional types that are acceptable.

For example, I want the function I've been working on to allow the x and y arguments to accept either a string or an array, and I'll do that with this syntax:

```
function totalLength(x: (string | any[]), y: (string | any[])): number {  
    var total: number = x.length + y.length;  
    return total;  
}
```

Since the string type and the array type both have a `length` property, my code continues to be perfectly valid. In fact, the string type and the array type happen to share a couple of members, and TypeScript gladly offers them as autocomplete options:



Autocomplete options on union types

If, however, I attempt to access a member that only belongs to one type or the other TypeScript will yell at me, telling me that I can't access the member because doesn't exist on both types:

```
function totalLength(x: (string | any[]), y: (string | any[])): number {
    var total: number = x.length + y.length;

    x.slice(0); // Works fine!
    x.push('abc'); // Error: only exists on arrays
    x.substr(1); // Error: only exists on strings

    return total;
}
```

This is fine for our example, but let's say that I really do want to be able to treat each variable differently depending on which type it is. In other words, it's nice that I can add up the total of the length property, or even call slice, but what if I want to be able to push a value onto an array if it's an array type or get the substring of a string if it is the string type.

Luckily, TypeScript is smart enough to handle that too - just use the built-in JavaScript keyword `instanceof` to test the variable's type, like this:

```
if(x instanceof Array) {
    x.push('abc'); // Works, because we know x is an array!
}
```

After doing that, TypeScript knows that in order to make it into the if block, the object had to be an Array so TypeScript lets you treat it like an array. This is known as the **type guard** syntax.

Once I exit the if block and am outside of the scope and safety of the type guard, TypeScript can no longer guarantee the variable's type, so I'm back to only being able to access the members that both union types share.

I can also do the same thing with the string type:

```
if(typeof x === 'string') {
    x.substr(1) // Works, because we know it's a string!
}
```

Notice how in this case I had to use the `typeof` operator and match it against the type name `string`. That's because `string` is a primitive type and this syntax is only reserved for the couple of primitive types. You'll almost always use the `instanceof` operator.

TypeScript's Union Types and Type Guard syntax can be really helpful in creating flexible and forgiving functions and variables that let you take advantage of JavaScript's dynamic nature while not completely giving up the safety of static typing. It may not be a feature you end up using a lot, but when you come across a situation where you want to be able to accept multiple different types in the same variable, it can be a great fit.

Function Overloads

In the previous section, I showed you how to use union types to give the option for consumers of a function to pass in several different types for a given property, but I ended by complaining that in some cases union types aren't restrictive enough.

In other words, I've got two parameters - *x* and *y* - and each of them can accept a string or an array. However - even though this happens to work because they both share a `length` property - it really doesn't make sense to add the length of a string to the length of an array. What I really want to do is expose two different signatures for this method: one that allows you to pass in two strings, and one that allows you to pass two arrays.

If you've got experience with a statically-typed language like C# or Java, you'd probably implement this using multiple different signatures of the function sharing the same name; in other words, you'd create an overloaded function that might look something like this:

```
function totalLength(x: string, y: string): number {  
}  
  
function totalLength(x: any[], y: any[]): number {  
}
```

TypeScript also supports the idea of overloaded functions, but not in the same way that these other languages do. In languages like C# and Java you'd implement two completely different functions and you'd associate them together simply by giving them the same name.

But, this just simply won't work in JavaScript because if you wrote the example I just did, the second definition of the `totalLength` function would simply overwrite the first definition!

So, in JavaScript we can only have one single implementation of a function:

```
function totalLength(x: (string | any[]), y: (string | any[])): number {  
    var total: number = x.length + y.length;  
  
    if(x instanceof Array) {  
        x.push('abc'); // Works, because we know x is an array!  
    }  
  
    if(x instanceof String) {  
        x.substr(1) // Works, because we know it's a string!  
    }  
  
    return total;  
}
```

The way that TypeScript allows us to define overloaded functions is to simply add the alternate signatures that we want to expose right on top of the actual function implementation, like this:

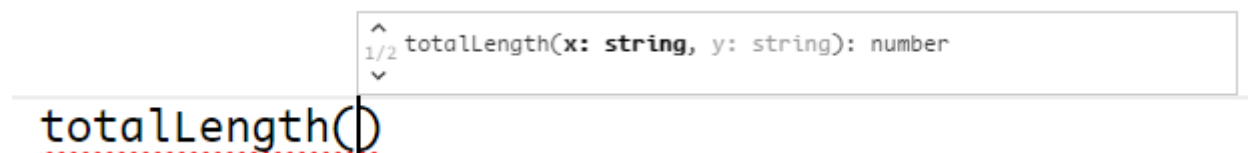
```
function totalLength(x: string, y: string): number
function totalLength(x: any[], y: any[]): number
function totalLength(x: (string | any[]), y: (string | any[])): number {
    var total: number = x.length + y.length;

    if(x instanceof Array) {
        x.push('abc'); // Works, because we know x is an array!
    }

    if(x instanceof String) {
        x.substr(1) // Works, because we know it's a string!
    }

    return total;
}
```

Notice that I haven't changed the underlying implementation at all - it still accepts either a string or an array for either parameter and its logic still works exactly the same. The only difference is that when I attempt to consume this function...



Overloaded function auto-complete

I only see the two overload signatures that allow me to pass two strings or two arrays, but TypeScript does not show me the option to call the underlying function that takes both a string and an array together.

Function overloads, like all other type information, is the first thing to be ripped out of the source code when TypeScript compiles it to JavaScript. So, the end result is that when you look at the generated JavaScript you will only see the underlying function and not have any hint that these overloads were ever defined in the TypeScript version.

Because it doesn't actually change the functionality of the function, you can think about function overloads in the exact same way that you think about specifying variable or parameter types – it's simply metadata for TypeScript to help you write better code.

6. Custom Types

Defining custom types with Interfaces

So far in the book I've showed you how to use TypeScript as a transpiler that allows you to take advantage of ECMAScript 2015 language features and as a way to guard yourself against many common bugs such as misspelling member names or returning the wrong type from a function. That's all very valuable, but the real value in using TypeScript is the fact that you can go above and beyond the types that JavaScript provides out of the box and define custom types that describe the data structures and behavior that your application uses.

TypeScript offers three ways to define a custom type - interfaces, classes, and enums - and in this chapter I'm going to start with the easiest of those three: interfaces.

An interface in TypeScript is the same as an interface in just about any other statically-typed language like C# or Java: it acts as a contract that describes the data and behaviors that the object exposes for others to interact with. In order to define an interface, you use this straightforward syntax:

```
interface Todo {  
}
```

Here I've defined an empty interface called `Todo` that I'll use to describe the `Todo` type that I'll be using to build the sample application through the rest of the book. I'll start by defining two properties: a `name` property to contain the display name of the `Todo`, and a `completed` property to indicate whether the `Todo` has been completed or still remains to be done:

```
interface Todo {  
    name;  
    completed;  
}
```

Now, this is perfectly legitimate, but as I just got done describing in the last chapter, since we have the ability to be very descriptive about our types, we should use it! So, let's specify a type for each of these properties.

Since the `name` property is going to hold a string value containing the name of the `Todo`, I'll specify the `string` type using the same syntax that I used to apply type information to variables and function parameters last chapter: a colon, followed by the type name:

```
interface Todo {  
    name: string;  
    completed;  
}
```

Then I'll do the same thing with the `completed` property, defining it as a boolean:

```
interface Todo {  
    name: string;  
    completed: boolean;  
}
```

Before I begin using this interface, take a look at the code that TypeScript generates to represent this interface at runtime:

Generated JavaScript

It's blank! That's because interfaces are strictly used for compile-time checks only, and have no effect on the code at runtime. Consider interfaces a way to tell TypeScript information about objects to help you catch more errors at build time, but don't ever rely on them being there at runtime. In fact, the first thing that TypeScript does when it compiles the final JavaScript output is to strip out all of the interface definitions.

Let's put this new interface to work by applying it to a new variable:

```
var todo: Todo;
```

And then I'll try to assign that variable with an object that doesn't fit the description that we've laid out in the `Todo` interface. We'll start with just an empty object by creating an object literal with no members defined:

```
var todo: Todo = { };
```

Notice how TypeScript warns us that this object is not compatible with the `Todo` type because it doesn't have the `name` property.

Alternatively, I could go the opposite right and instead of the defining the type of the variable I can specify the type of the object itself using the "casting" syntax, which looks like this:


```
var todo = <Todo>{ };
```

However, I tend to stick with the other way because I find it a little bit cleaner to tell TypeScript what type I want the variable to be and have it make sure I'm assigning the right object, rather than the other way around...

```
var todo: Todo = { };
```

So, now that I have an object that TypeScript is warning me is not really a `Todo`, let's go ahead and add a `name` property... but instead of defining it as a string type like TypeScript is expecting, let's try to trick TypeScript and initialize it to a number value instead:

```
var todo: Todo = {  
  name: 123  
};
```

Nope, TypeScript doesn't like that, either! Now it's telling us that it did find a property called `name`, but that property is not compatible with the type of the `name` property that we defined on the `Todo` interface... In other words, TypeScript is telling us, "You told me this was supposed to be a string, but you're trying to set it to a number and I'm not gonna let you do that!"

If we correct this issue by assigning the property to a string...

```
var todo: Todo = {  
  name: 'Pick up drycleaning'  
};
```

TypeScript gives us the next error: that this object is now missing the `completed` property. At this point we've got two options:

1. Satisfy the interface by adding a `completed` property with a boolean value, or...
2. Decide that maybe not all `Todo` instances really do require a `completed` property and update the interface to make this property optional by adding a question mark right after the name:

```
interface Todo { name: string; completed?: boolean; }
```

What this does is tells TypeScript: "Not every `Todo` object needs to have a `completed` property in order to be considered a `Todo`... *but*, when it does have the `completed` property, that property should always be a boolean value."

It's important to note at this point that interfaces just describe what you intend or expect to happen - they don't actually enforce the fact that values will be of a certain type or have the expected

properties at runtime. To illustrate this point, consider a scenario where these objects are coming from outside of our application, for instance if we were retrieving them from a remote web service through an AJAX call. In this scenario we have no way of controlling what will actually come back from that server, and in fact the server could choose to change its data structure even after our application had been running successfully for some time. When the server changes the data it sends back, our TypeScript interfaces aren't going to do anything to help guard us against that – but what they will do is allow us to update the interface in one place and more easily fix the issue throughout our application.

Data fields aren't the only thing that you can define in an interface: you can also specify the behaviors that an object might have by defining method signatures on the interface. To define a method signature, you use the same exact syntax that you use to define a function, except you just drop the function keyword.

Since I am plan to only use this `Todo` interface to describe the data structure of a `Todo` object with no behavior, I'll create a new interface called `ITodoService` that I will use to do things like create, retrieve, update, and delete `Todos`:

```
interface IToDoService {  
}
```

Then, I'll define some method definitions to give it some behavior.

First I'll define a method that gets me a single `Todo`, called `getById()`:

```
interface IToDoService {  
    getById(todoId: number): Todo;  
}
```

This method takes one argument - the number of the `Todo` to get - and returns a single `Todo` value.

Then, I'll define a method that gets me all of my `Todos` called `getAll()`:

```
interface IToDoService {  
    getAll(): Todo[];  
    getById(todoId: number): Todo;  
}
```

This method takes no arguments and returns an array of `Todo` objects.

Next, I'll add a method to delete a `Todo`:

```
interface IToDoService {  
    delete(todoId: number): void;  
    getAll(): Todo[];  
    getById(todoId: number): Todo;  
}
```

This method takes one argument called `todoId` which is the numeric ID of the `Todo` item to delete and it doesn't return anything which I'll indicate by using `void` as the return type.

Finally, I'll define an `add` method that'll let me add new `Todos`:

```
interface IToDoService {  
    add(todo: Todo): Todo;  
    delete(todoId: number): void;  
    getAll(): Todo[];  
    getById(todoId: number): Todo;  
}
```

This method takes in a `Todo` object and then returns the newly-created `Todo` object (which may or may not be the same `Todo` object that was passed in).

And that's it - I've now shown you how to create two interfaces: one that describes a service that has methods that I can call and another one that simply defines the data structure of an object to make it easier for me to interact with instances of those objects.

Note that I'm not actually going to implement this `IToDoService` just yet - that's not what this chapter is about. Instead, I'll save that for the next chapter where I show you how to create classes that implement interfaces. However, keep reading the rest of the sections in this chapter where I'll show you all the interesting things that you can do with TypeScript interfaces, such as using the concept of structural typing to match objects to interfaces implicitly.

Using interfaces to describe functions

In the previous sections I showed you how to create an interface that describes the data members or methods of an object. This is great when you're dealing with a simple object, but in addition to being a dynamic language, JavaScript is also a functional language, meaning that not only can you assign a function to a variable like this:

```
var $ = function(selector) {  
    // Find DOM element  
}
```

But that functions are also objects that can have their own properties and methods as well! In other words, this is perfectly legitimate JavaScript code:

```
var $ = function(selector) {  
    // Find DOM element  
}
```

```
$.version = 1.12;
```

If this is legitimate JavaScript code, that means that sooner or later we're going to want to describe it with an interface, and that's exactly what I'll show you how to do in this section.

First, let's start off with what we already know how to do and give the interface a `version` property that's a number type:

```
interface JQuery {  
    version: number;  
}
```

Now, in order to define the function itself we simply add a function property without a name:

```
interface JQuery {  
    (selector: string): HTMLElement;  
    version: number;  
}
```

Other than the fact that it doesn't have a name, all of the normal rules apply: I can specify parameters and return values and everything.

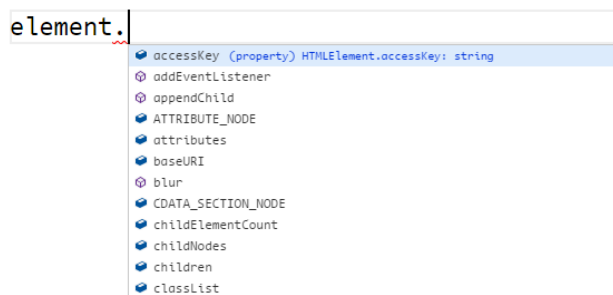
Here I've chosen to return an `HTMLElement`, which is a built-in type that is exactly what it sounds like - a DOM element! That means that if I call this method TypeScript will be able to give me an object with full autocomplete on all the properties and behaviors available to HTML elements.

Once I've defined the interface, I can apply it to the function that I've defined by using the casting syntax to force TypeScript into treating this function an instance of this interface.

```
var $ = <jQuery>function(selector) {  
    // Find DOM element  
}
```

With this interface defined and applied to the \$ variable, I can now call the \$ function and get the full typing of the jQuery interface:

```
var element = $('#container');
```



HTMLElement autocomplete options

In this section, I showed you how to define interfaces that describe functions, and I did using the popular jQuery API as my example. However, for those of you who are familiar with the jQuery API, you know that it offers far more features than just what I've shown here. And, not only that, but it also offers a plugin model that lets anyone come along and dynamically add functionality to the API as well.

In the next section, I'll show you how you can extend this interface to describe all of this functionality without having to actually modify it.

Extending interface definitions

In this chapter I've shown you how to define and apply interfaces that describe the properties and methods of an object, but JavaScript is a dynamic language, which means that it is very easy to extend the structure of an object by adding methods and data properties on the fly.

Perhaps the best example of this is the jQuery library and the way it allows you to extend its behavior through its plugin design pattern.

If you're not familiar with jQuery, it is a library that makes it very easy to work with DOM elements and in fact, I started to describe it with the 'jQuery' interface that I made in the last section, and have updated just a little bit to make it behave a little more like the real jQuery library for this demo.

The biggest difference between this interface and the jQuery library, however, is that instead of HTML elements, the real jQuery function returns a `jQueryElement`, like this:

```
interface jQuery {  
    (selector: (string | any)): jQueryElement ;  
    fn: any;  
    version: number;  
}
```

A `jQueryElement` is a JavaScript object that has helper methods like the `data()` method which allows you to assign data to an HTML data property, or extract a previously-assigned value from a data property.

In other words, the interface for the `jQueryElement` (or at least the part that I'm interested in right now) might look like this:

```
interface jQueryElement {  
    data(name: string): any;  
    data(name: string, data: any): jQueryElement;  
}
```

And I would use jQuery to modify data properties like this:

```
var todo = { name: "Pick up drycleaning" };  
var element = $('#my-element');  
element.data('todo', todo);  
var savedTodo = element.data('todo') // returns todo
```

In the first line I use the jQuery `$` function to get a reference to a DOM element. Then, I use the `data()` method to assign the `todo` object to the DOM element's data property named "todo". Finally,

I use the `data("todo")` method again to extract the value I just set by calling it with only the name of the data property that I want to retrieve.

Now, that's the behavior that comes out of the box in the jQuery library. But, as I mentioned a minute ago, jQuery allows you to extend this API by simply attaching new functions to its `fn` property, like this custom method that assigns an instance of a `Todo` to the data of an HTML element, or retrieves the previously-assigned `Todo` instance:

```
$.fn.todo = function(todo?: Todo) {  
  
    if(todo) {  
        $(this).data('todo', todo)  
    } else {  
        return $(this).data('todo');  
    }  
  
}
```

What's more, the jQuery plugin model says that once I add this method to the `fn` property, I *should* be able to simply call it from any `jQueryElement`, just like I called the `data()` method I just showed. In other words, I should be able to do this:

```
element.todo(todo);
```

But, if I try to do that TypeScript yells at me since my custom `todo` extension method is not listed on the `jQueryElement` interface, even though I know that jQuery will make it available to me so it should be listed as a valid method!

Of course, I just created the `jQueryElement` interface so in this example I could fix this by just updating the interface definition to include the new method, like this:

```
interface jQueryElement {  
    data(name: string): any;  
    data(name: string, data: any): jQueryElement;  
  
    todo(): Todo;  
    todo(todo: Todo): jQueryElement;  
}
```

I *could* do that, but since jQuery is a third-party library that I didn't create, let's assume that this interface was actually created by the jQuery team and not me. In that case, it's a really bad idea to update their interface to add my own extensions.

Luckily, TypeScript will let you extend any interface without actually changing the original definition, and it's actually much simpler than you might expect.

Rather than update the jQuery team's interface, I can create a brand new interface that shares the same exact name as the one that I want to extend:

```
interface JQueryElement {
    data(name: string): any;
    data(name: string, data: any): JQueryElement;

    todo(): Todo;
    todo(todo: Todo): JQueryElement;
}

interface JQueryElement {
}
```

With this in place, I can move my custom methods into this new interface, keeping them completely separate from the original interface that the jQuery team owns:

```
interface JQueryElement {
    data(name: string): any;
    data(name: string, data: any): JQueryElement;
    todo(): Todo;
    todo(todo: Todo): JQueryElement;
}

interface JQueryElement {
    todo(): Todo;
    todo(todo: Todo): void;
}
```

Now if I look at the autocomplete options for the element I see both the data and the todo methods listed, even though my custom methods were not on the original interface.

Even though it seems like a second definition of the same interface would just conflict with or overwrite the first definition, it doesn't. Anything added to this second definition (or a third, or any number of additional definitions) will simply be tacked on to the original interface just as if it were directly added directly to it.

Generally speaking, it's a pretty good idea to only apply this approach to code that you don't own, as I've done in this example. In other words, if I wanted to extend one of the interfaces in my own application - such as the `Todo` or `ITodoService` interfaces that I made earlier - I'd be better off just

going and adding these new methods directly to the original interface. If, however, I extend the behavior of a third-party library, using this approach is a pretty good way to extend the library's interface without actually changing the original code.

Defining constant values with enums

In the beginning of this chapter I mentioned that TypeScript offers three different ways to define custom types: interfaces, classes, and enums. I showed you how to define interfaces in the previous two sections, and I'll spend the whole next chapter showing you how to use classes. So that leaves enums, which is what I'll cover in this section.

If you're familiar with the concept of enums from other languages such as C# or Java, enums in TypeScript act in pretty much the same way: they are a way to define a set of meaningful - and constant - values that you can use to replace the "magic" strings and numbers that you would otherwise use.

What's a "magic" value? The easiest way to explain is to look at an example.

Up to now, I've been using a boolean value - either true or false - as the type of the completed property that describes whether a todo is completed or not:

```
var todo = {  
  name: "Pick up drycleaning",  
  completed: true  
}
```

But, let's say instead of that a todo can be in one of several different states: New, Active, Complete, and Deleted.

In other words, instead of that completed property, I want to have a state property, like this:

```
var todo = {  
  name: "Pick up drycleaning",  
  state: 1  
}
```

Here I've used a number to indicate which state the todo is in, but I've got a couple of different options.

I might assign each state a number, like this:

```
/*  
New = 1  
Active = 2  
Complete = 3  
Deleted = 4  
*/
```

Or, I could assign them all a string or a character, like this:

```
/*  
New = 'N'  
Active = 'A'  
Complete = 'C'  
Deleted = 'D'  
*/
```

Whatever values I decide to use, I'll end up hard-coding those values all throughout my code.

For example, let's say a todo must be in the "Complete" state in order to delete it, so I'd end up writing this logic:

```
function deleteTodo(todo: Todo) {  
  
    if(todo.state !== 3) {  
        throw "Can't delete incomplete task!"  
    }  
  
}
```

This makes a lot of sense right now because I just told you that the value 3 means the "Complete" state. But, what happens when we have to read this in 3 months and we've completely forgotten what all of those values are, or a new developer joins the team who just doesn't know? All of the sudden, this same code is completely unreadable because we have no idea what 3 refers to.

In this case 3 is the "magic number" - a value that is crucial to making the application work, but is completely meaningless without some kind of context. Enums provide that context.

The syntax to define enums in TypeScript is very similar to defining them in other languages such as C# and Java. It looks like this:

```
enum TodoState {  
    New = 1,  
    Active = 2,  
    Complete = 3,  
    Deleted = 4  
}
```

Take a look at what JavaScript code TypeScript generates:

```
(function (TodoState) {
    TodoState[TodoState["New"] = 1] = "New";
    TodoState[TodoState["Active"] = 2] = "Active";
    TodoState[TodoState["Complete"] = 3] = "Complete";
    TodoState[TodoState["Deleted"] = 4] = "Deleted";
})(TodoState || (TodoState = {}));
```

Ok, maybe it's not obvious what this code does.

Instead, copy this generated code, open the developer console of your browser and just paste it in.

Then, inspect the `TodoState` object to see what it looks like:

```
TodoState // {1: "New", 2: "Active", 3: "Complete", 4: "Deleted", New: 1, Active: 2, Complete: 3, Deleted: 4}
```

I can see here that the code that TypeScript generated creates two properties on the `TodoState` object for each enum value I defined: a numeric value with a string key, and a string value with a numeric key.

In other words, `TodoState.New` translates to the number 1. And, if I have the number 1 and I want to get its name, I can request the name by requesting it from the enum object like this:

```
TodoState[1] // "New"
```

Note that, even though this looks like the syntax I'd used to access a value in an array, it is really the syntax you use in order to dynamically access a property on an object. In this case, I passed the value 1 directly, but most of the time when you use this syntax with an enum, you're going to be passing in a variable.

For example, I might have a variable named `todoState` that I assign to an enum value somewhere:

```
var state = TodoState.New
```

And then later on in my code I want to display the name of that enum value:

```
TodoState[state]
```

Remember, the actual value of the `state` variable is actually a number:

```
state // 1
```

Now that I have my new `TodoState` enum defined, I'll revisit and refactor that example from earlier, and instead of hard-coding the number 1 to mean the "New" todo state, I'll use the enum:

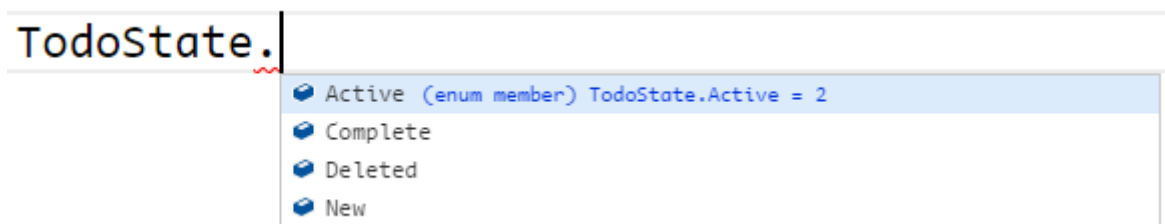
```
function deleteTodo(todo: Todo) {  
  
    if(todo.state !== 1) {  
    if(todo.state !== TodoState.New) {  
        throw "Can't delete incomplete task!"  
    }  
  
}
```

At runtime, this code works pretty much exactly the same as before – in other words, when this expression evaluates it really says “if(todo.state is not equal to 1)”.

In fact, with this enum in place, I can actually update the Todo interface, updating the type of the state property to be TodoState to enforce the fact that consumers may only assign valid values that I’ve defined in the TodoState enum:

```
interface Todo {  
    name: string;  
    state: TodoState;  
}
```

Also, in addition to all of those benefits, TypeScript is also able to help me out by giving me a great autocomplete list of TodoStates that I can choose from as I type:



TypeScript enum autocomplete

Even though all of this code is still functioning pretty much exactly the same as it did before I introduced enums, using TypeScript’s enum feature offers a simple - yet incredibly powerful - way to reduce those “magic strings” that we find ourselves relying on so often and give meaning to those important values in our code, which ultimately make the code much easier to understand. I think that once you try using enums, you’ll wonder how you ever worked in JavaScript without them.

Anonymous Types

Now that I've showed you how to define custom types in TypeScript using interfaces, let me add a little twist: you can actually declare interfaces right in-line anywhere that accepts a type. This approach is called an **anonymous type**.

For example, let's say that I have a variable that I want to accept an object that has a `name` property. Rather than define a whole interface with just one property, I could simply define the type right inline, like this:

```
var todo: { name: string };
```

With this anonymous type definition in place, TypeScript will begin enforcing it. For instance, it'll give me an error if I try to assign an object to this variable that doesn't have the `name` property:

```
todo = { age: 42 };
```

This can actually be very useful in certain scenarios. Remember in the last chapter when I introduced you to union types with this example to allow consumers to pass in either strings or array values?

```
function totalLength(x: (string | any[]), y: (string | any[])): number {  
    var total: number = x.length + y.length;  
    return total;  
}
```

If all I really care about is the `length` property of these two arguments, why should I restrict them to just string and array types? Why not embrace JavaScript's dynamic nature and just let in ANY object that has a `length` property (as long as it's a number, of course)?

Well, if I replace those union type with anonymous types I can achieve exactly that:

```
function totalLength(x: { length: number }, y: { length: number }) {  
    var total: number = x.length + y.length;  
    return total;  
}
```

With this simple change, I've now opened the door to any object with a `length`, making this code dramatically more reusable and, personally speaking, I think it's also much cleaner and easier to understand.

Keep in mind, there is one issue that still remains and that is there is nothing forcing consumers of this method to pass in two values of the same exact type, therefore allowing them to continue to be able to add the length of an array to the length of a string for example. But, we'll take care of that issue in the generics chapters. For now, let's keep moving on to the next section and check out the other ways that TypeScript allows us to define custom types.

```
var todo: { name: string };
```

```
todo = { age: 41 }
```

```
function totalLength(x: { length: number }, y: { length: number }): number {  
    var total: number = x.length + y.length;  
    return total;  
}
```

Using interfaces to dynamically access object properties

When I first heard about TypeScript and that it was bringing static typing to my beloved dynamically-typed JavaScript language, I wrote it off immediately. After all, for all of the issues JavaScript has, its dynamic nature can be incredibly powerful when you use it correctly! As far as I was concerned, anything that took that dynamic power away was bad in my book.

Of course, as I've mentioned several times already, TypeScript doesn't *take away* anything from JavaScript. And, whenever you come across a scenario when you want to ditch static typing and really leverage JavaScript's dynamic nature, you can always feel free to use the any type to tell TypeScript not to enforce strong typing rules on a given object.

For example, one pattern that I use all the time in my JavaScript code is to leverage a regular object as a collection, in which each of the object's properties acting as the key for each of the items in the collection, like so:

```
let todoComments = {  
  123: [ "Still need to clean the Bat computer" ],  
  456: [ "Waiting to hear back from Alfred." ]  
}
```

In this example, I've create an object named `todoComments` to store an array of comments for each of my `Todo` objects, using the `Todo` items' `Id` value as the property name. Using this approach, I can access `Todo` comments like this:

```
let comments = todoComments[todo.id];
```

Or, I can add new comments on the fly, like this:

```
todoComments[todo.id] = [ "Wow, what a great example!" ];
```

Of course, TypeScript supports this approach just fine right out of the box. The problem comes when we consider the fact that you and I can look at the code and see what it does - that whenever I pass in a `Todo` item's `Id`, I get a collection of strings back - however, TypeScript has no idea about that... unless we tell it.

Luckily, there's a simple syntax to do this:


```
interface IToDoCommentsLookup {  
    [todoId: number]: string[];  
}
```

What this interface says is that the object that implements it offers an indexer that accepts a number as the key and returns a collection of strings (`string[]`).

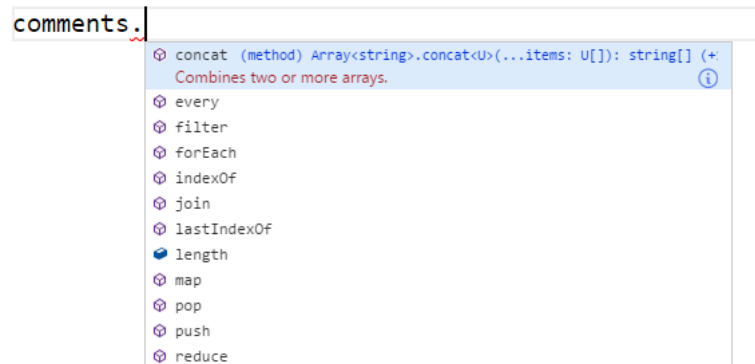
Of course, you apply this interface like any other interface:

```
let todoComments: IToDoCommentsLookup = {
```

Then, from there on out, whenever you use the accessor with a number, TypeScript will know that the object returned is an array of strings and will give you the proper static typing behavior:

This technique may not be something that you use every day, but it is one that can be really powerful when it is applicable. So, keep this syntax in mind the next time you want strong typing support on an object that you've retrieved using an object's indexer.

```
let comments = todoComments[todo.id];
```



Full type support for indexed properties

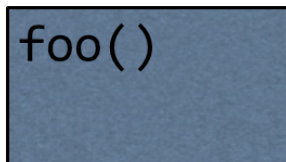
7. Classes

Understanding Prototypical Inheritance

In the previous chapter I mentioned that TypeScript offers 3 ways to define custom types: interfaces, enums, and classes, then I showed you all about how to use interfaces and enums. Now it's time to show you how to implement custom types using classes and use those classes to implement all kinds of object-oriented functionality such as inheritance, abstraction, and encapsulation. Before I get into the syntax of creating a class, however, I want to define what JavaScript a class is - and is not.

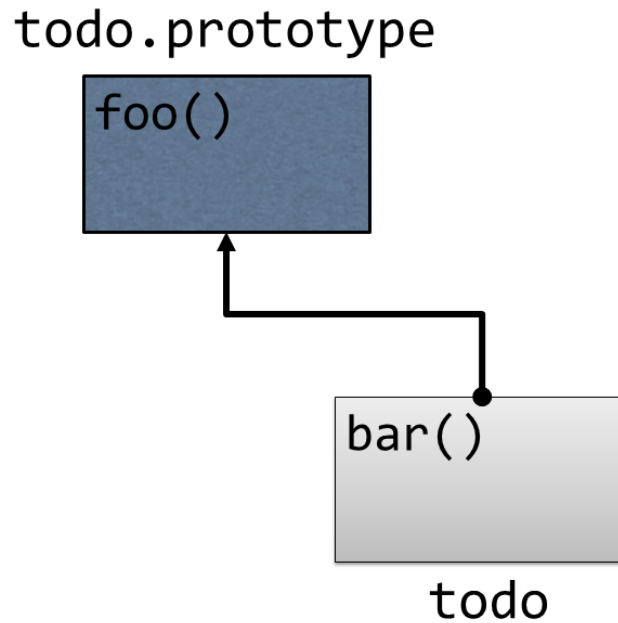
First off, I want to be clear that TypeScript doesn't introduce the concept of a class - ECMAScript 2015 does! However, it's crucial to understand that - even though JavaScript does now have a `class` keyword and syntax, all of this is syntactic sugar and it doesn't change the fact that JavaScript is still based on objects and prototypical inheritance, otherwise known as prototype-based programming.

Prototype-based programming all starts with a special object called the **prototype**.



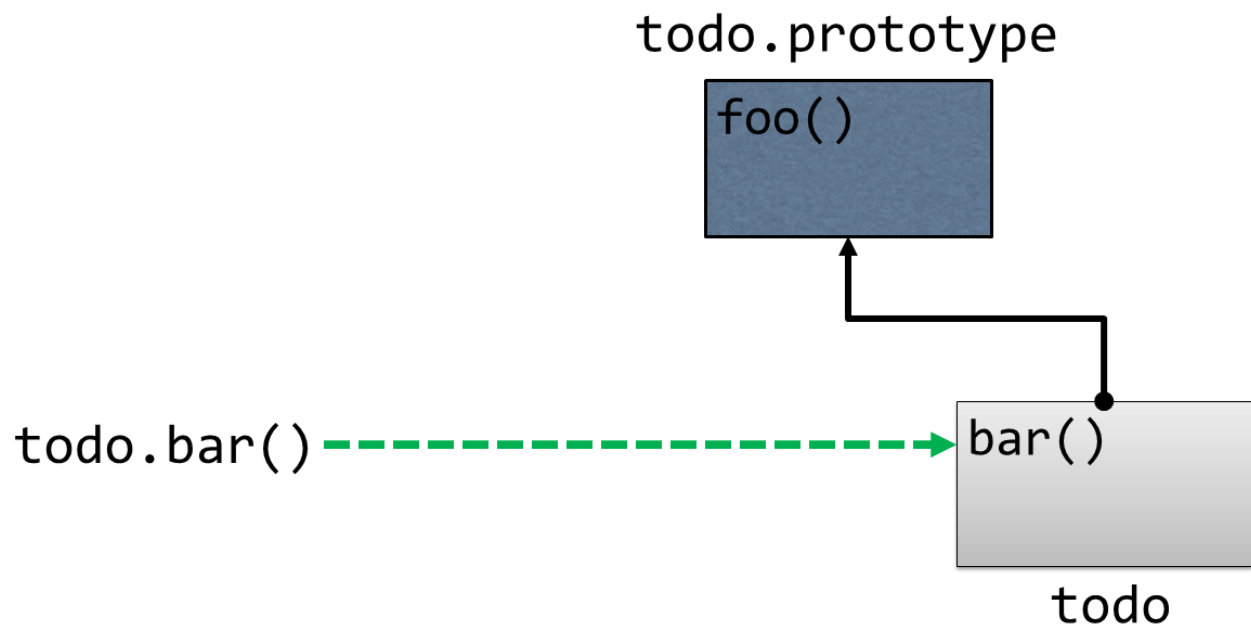
The Prototype

If you want to share behavior between object instances, you define that behavior on the prototype object, and then link other object instances to that object.



Prototypical Linking

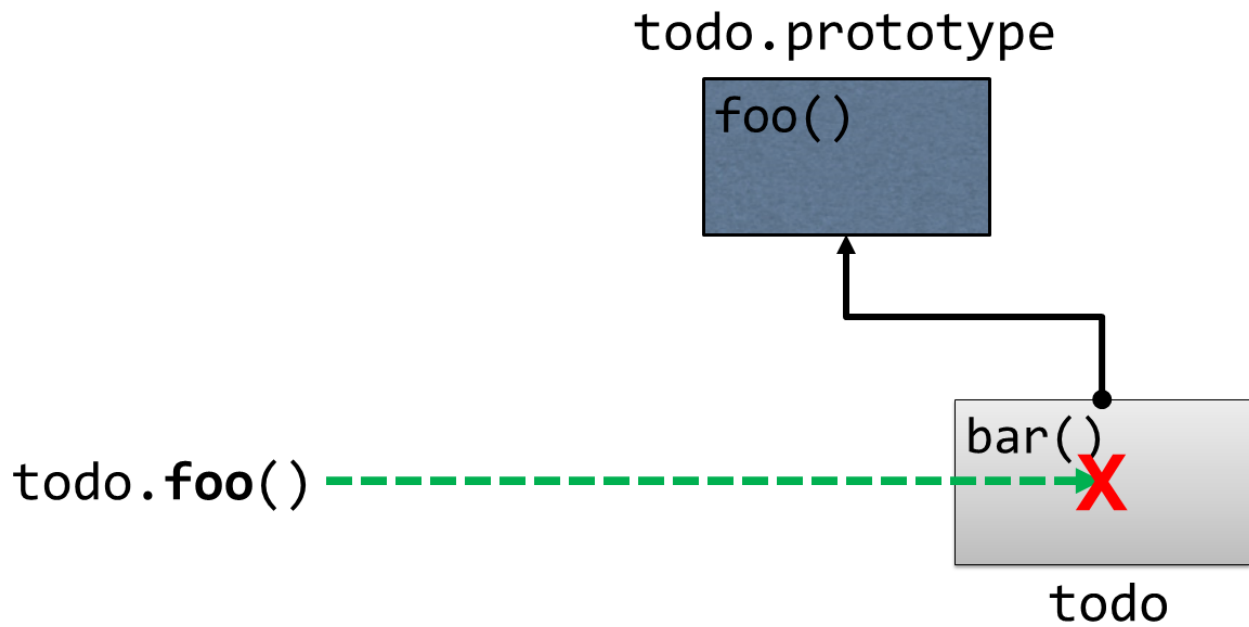
Keep in mind that in JavaScript, objects are just bags of dynamic properties, which means that accessing a member of an object is not as simple as testing whether or not that member exists. Instead, whenever you attempt to access any member of an object - regardless of whether it's a method or a value field - JavaScript tries as hard as it can to find the member that you're looking for.



Prototypical Linking: locating a member

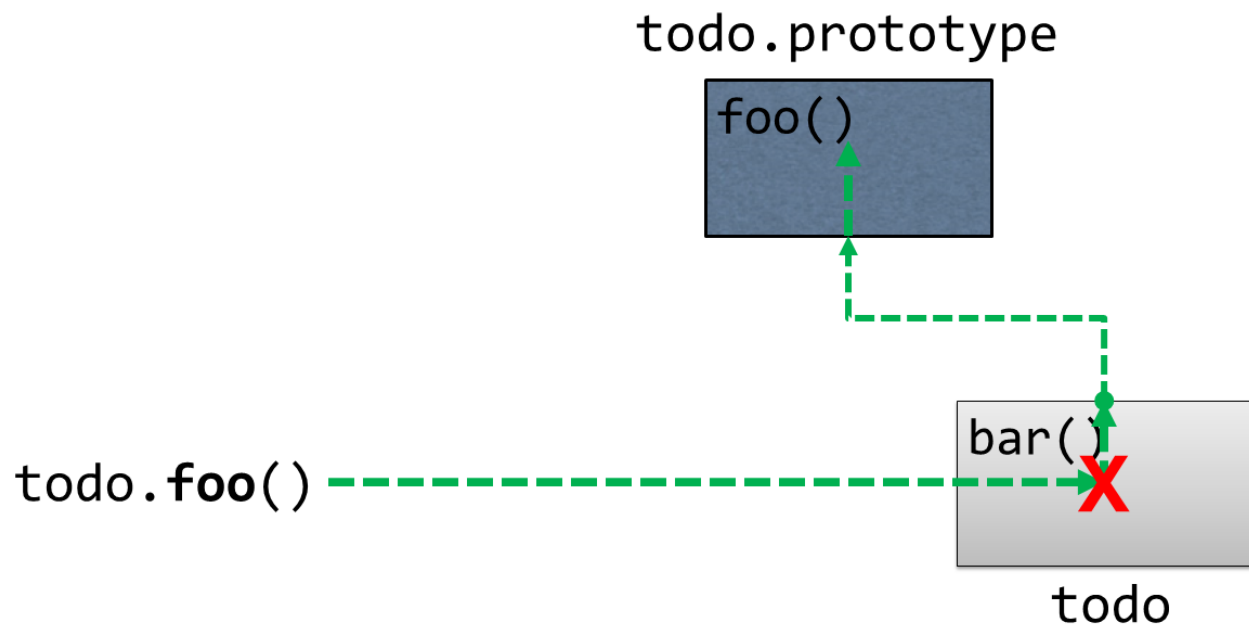
First, JavaScript looks for the member on the object itself, just like you asked it to. If it finds a member with that name on the object you referenced, then great! It accesses that member and it's done!

If, however, JavaScript does not find a matching member on that object, it doesn't give up there.



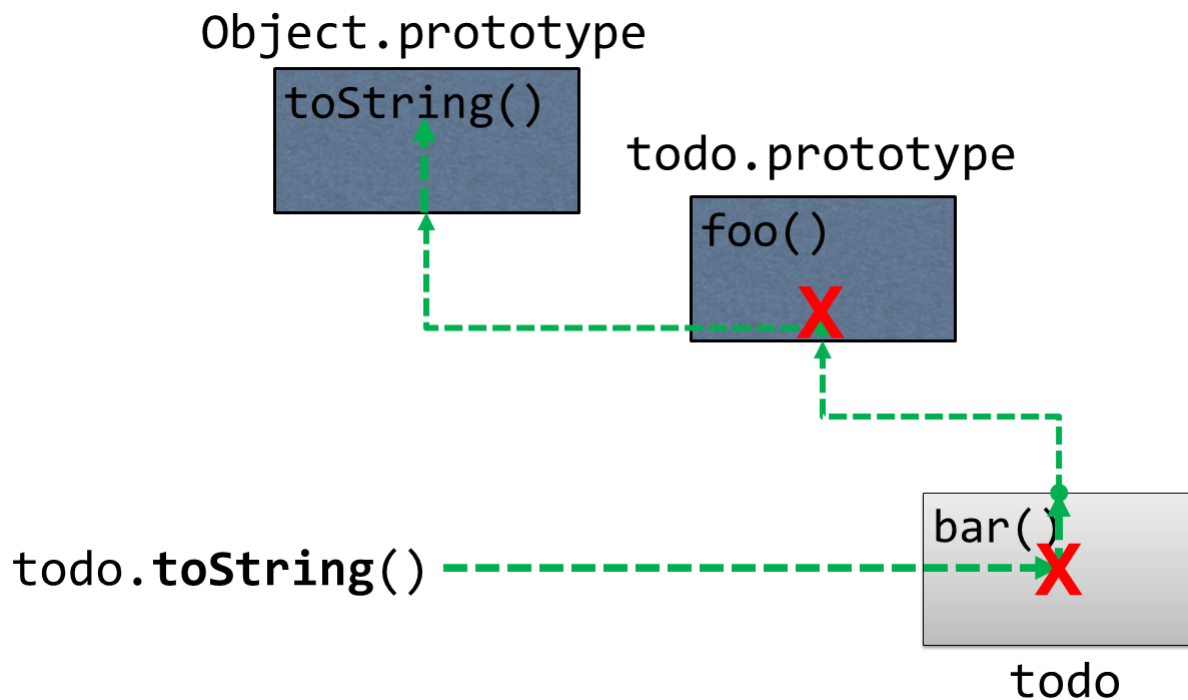
Prototypical Linking: locating a missing member

Instead, it looks at the object's prototype object to see if the member exists on that object. If a member of that name exists on the prototype, then JavaScript refers to that member and it's done! Otherwise, it accesses the prototype's prototype, and the cycle continues... all the way up the chain until it reaches the root of everything: the Object prototype.



Prototypical Linking: locating a missing member (second try)

And I mean that quite literally: `Object.prototype` is actually an object that you can see and interact with. That means that if you type `Object.prototype` into the console...



Prototypical Linking: `Object.prototype`

everything that you see on this object will be available to all other JavaScript objects.

The next question is: where does this special prototype object come from and how (and when) is it assigned to my objects? The simple answer is that, most of the time, JavaScript does it for you. The more complex answer is that there are actually quite a few ways that prototype objects get assigned.

Let's look at a couple of those ways, starting with a simple object literal:

```
var todo = { };
```

When JavaScript creates any object - even one as simple as this - it automatically links its prototype object to the `Object.prototype`.

The same is true of the other two “special” types of objects that I mentioned in the last chapter - functions and arrays - only, when JavaScript creates objects with these two types, it links them to the `Function.prototype` and `Array.prototype` objects respectively, allowing all functions and arrays to share common behavior such as the `.slice` method on an `Array` or the `.bind()` method on a function.

Perhaps the most common way that JavaScript assigns a prototype to an object is with a **constructor**, which is really just a function that is called with the `new` keyword. When you initialize an object with the `new` keyword, JavaScript does three things:

1. Creates a new object

2. Sets the new object's prototype to the constructor function's prototype
3. Executes the function that you called with the `new` keyword (referring to the new object as `this` within that method)

To see this in action, I'll create a new function that will act as the constructor function:

```
function TodoService() {  
    this.todos = [];  
}
```

This simple function just assigns the `todos` property on the `this` object to a new, empty array. Remember, this `this` object refers to the object that JavaScript just created for us in the first step.

Then, I can define members that I want to be available to all instances of `TodoService` to its prototype, like this:

```
1 function TodoService() {  
2     this.todos = [];  
3 }  
4  
5 TodoService.prototype.getAll = function() {  
6     return this.todos;  
7 }
```

Notice how I'm referring to the array of `todos` through the `this` keyword in this method. Just as you'd expect, the `this` keyword refers to the instance of the object, so `this.todos` on line 6 will refer to the same `this.todos` on line 2 at runtime.

With this in place, I can create a new instance of the `TodoService`

```
Var todoService = new TodoService();
```

And then I can call the methods that are defined on the prototype like they were defined on the instance itself:

```
todoService.getAll();
```

And that is a very quick overview of prototypical inheritance in action. I gave this recap because it's crucial to understand prototypes and prototypical inheritance when dealing with any JavaScript code, but it's especially important to keep this behavior in mind as you use the class syntax that I will show you in the next section to define the behavior in your application. Just keep in mind that, even though you may be using a new syntax, you're still using JavaScript's prototype-based behavior.

Defining a class

In the last section, I explained how JavaScript's prototypical inheritance works and how you can share behavior across objects by defining behavior on a prototype object. If all of that seemed a little overwhelming or unwieldy to have to type, I don't really blame you. That's why ECMAScript 2015 introduced the class syntax that allows you to define this same behavior in a much simpler and more straight-forward way. What's more, you can use all of the TypeScript features I've shown so far to extend the ECMAScript 2015 class syntax with additional static type information.

At its core, the ECMAScript 2015 class syntax is pretty simple. To demonstrate, I'll convert the prototype-based example from the previous section to use the class syntax instead:

```
function TodoService() {  
    this.todos = [];  
}  
  
TodoService.prototype.getAll = function() {  
    return this.todos;  
}
```

I'll start by defining the class itself, which is just the class keyword, followed by the class name - in this case, `TodoService` - followed by brackets that contain the class's members:

```
class TodoService {  
}
```

Then, I can begin adding members to the class. The syntax you use to define members of a class is almost exactly the same as the syntax you used to define members of an interface.

I'll start with the class's constructor function, which is the function that JavaScript calls to initialize the new object. In the prototype-based syntax, the constructor function is simply the function that you attach all of the prototype behavior to - in our example, it's the function named `TodoService`.

In order to define a constructor function using the class syntax, you simply define a method with the name `constructor`, like this:

```
class TodoService {  
    constructor() {  
    }  
}
```

Likewise, I can add methods to the class as well, using the same syntax that I used to define them on an interface: basically just the standard way of defining a JavaScript function, except without the `function` keyword.

For example, I can implement the `getAll` method, like this:

```
class TodoService {  
    constructor() {  
    }  
  
    getAll() {  
    }  
}
```

Now that I've got a class with a constructor and a method in place, let's add the array of todos that I referenced in the prototypical inheritance example.

In that example I defined the property by simply assigning a value to the `this` object - in other words: `this.todos = []`. Let's try to copy that call into the new constructor and see what happens.

```
constructor() {  
    this.todos = [];  
}
```

When I try to assign a property on the `this` object in a class constructor, TypeScript will yell at me, saying "Hey! You're trying to assign a value to a property that I don't know about!" That's a good thing - TypeScript is trying to make me define a property before I access it... but, how do I do that?

Well, there are a couple of ways...

The first and simplest way is to define the property just like I did with on the interface: by defining the member outside of the constructor like this:

```
class TodoService {  
  
    todos;
```

Up to this point, everything I've shown you is just ECMAScript 2015 class syntax, but here's where TypeScript comes into play by allowing me to specify type information on this property:

```
class TodoService {  
  
    todos: Todo[];
```

This syntax tells TypeScript that the `TodoService` has a property named `todos` that can contain an array of `Todo` objects, but it doesn't actually create the property - that only happens in the constructor when I call `this.todos = []`;

However, I can combine these two operations by defining a property and assigning and initializing it with a value all in one expression like this:

```
todos: Todo[] = [];
```

This approach is a great way to define an initial variable that's the same every time a new object is created. But, you may often find that you would like to accept these initial values as parameters to the constructor function as opposed to the hard-coded values I've shown here.

In other words, you'll end up doing this:

```
todos: Todo[] = [];  
  
constructor(todos: Todo[]) {  
    this.todos = todos;  
}
```

Since this is a pretty standard practice, TypeScript actually gives you one more bit of syntactic sugar, and that is the ability to define a constructor parameter and a class property all in one expression, simply by putting an access modifier such as `private` in front of it.

In this example, that means that all five of these lines can be condensed into one expression:

```
constructor(private todos: Todo[]) {  
}
```

Finally, now that I've defined and assigned the `todos` property, I can update the code in the `getAll` method to retrieve them by referencing them as `this.todos`, just like in the prototype-based approach:

```
getAll() {  
    return this.todos;  
}
```

With this in place, take a look at the code that TypeScript generates.

TodoService.js (generated code)

```
var TodoService = (function () {  
    function TodoService(todos) {  
        this.todos = todos;  
    }  
    TodoService.prototype.getAll = function () {  
        return this.todos;  
    };  
    return TodoService;  
})();
```

Does that look familiar? That's right, after all of this syntactic sugar is stripped away and compiled down, TypeScript is just creating a prototype-based object underneath it all. That's why the bulk of the code that you'll write inside of your methods will look pretty much exactly the same as it did when you user using the prototype-based syntax... because it is still prototype-based!

Regardless of that fact, I've found the class syntax to be an effective way to both think about and implement object behavior. And this syntax becomes even more powerful as you implement the rest of the features in this chapter to build out nice, clean, and maintainable objects. So, move on to the next sections where I'll show you all of the ways that you can extend and leverage TypeScript classes.

Applying static properties

There are some situations in which you need to be able to maintain one single value that is used across many components - or many instances of one type of component.

For example, let's say that every time I create a new Todo item, I want to assign it a unique ID value so that I can retrieve and refer to it later on. One of the ways I might do that is to keep a single counter in my application and increment it with each new Todo item that is added: the first Todo is #1, the second is #2, etc. But, in order to implement that approach, I'd need to ensure that my application has one single variable to keep track of that counter.

In other languages such as C#, Java, or even C/C++, you'd refer to these kinds of variables as **static members**.

For years, the easiest and most acceptable way to implement static members in JavaScript was to simply create a variable in the global namespace, like this:

```
var lastId = 0;
```

However, global variables are now generally considered bad practice and to be avoided at all costs. So, instead of putting the variable in the global scope, I want to be able to nest it inside some other object.

These days, the more generally accepted practice of defining a static variable is to attach it to an object - especially the function that is going to use it the most. In order example of incrementing a unique ID to apply to new Todo items, that function would be the `TodoService` I created in the previous section.

Prior to the ECMAScript 2015 class syntax, when I'd create a constructor function and then attach behavior to it via its prototype, I'd define that `lastId` variable directly onto the `TodoService` constructor function itself, like this:

```
function TodoService() {  
}
```

```
TodoService.lastId = 0;
```

Since there is only ever one instance of the `TodoService` constructor function in our application, there will only ever be one instance of any variables I attach to that function.

In order to access this static variable - even in methods that are part of the `TodoService` - I'd have to do so by referring to the variable through the constructor function, exactly as I'd assigned it in the first place.

In other words, I'd have to access it this way:

```
TodoService.prototype.add = function(todo) {  
    var newId = TodoService.lastId += 0;  
}
```

Likewise, I can use the same approach to define a static method as well:

```
TodoService.getNextId = function() {  
    return TodoService.lastId += 0;  
}
```

And, I'd access the static method in the same way, too:

```
TodoService.prototype.add = function(todo) {  
    var newId = TodoService.getNextId();  
}
```

Now, that's the way I'd do it in the ES5 syntax, but with the ECMAScript 2015 class syntax, I simply add the property to my class just like I would add a regular property, and then put the keyword `static` in front of it:

```
class TodoService {  
  
    static lastId;  
  
    constructor() {  
    }  
}
```

I can even apply type information and assign it an initial value, just like any other class property.

```
class TodoService {  
  
    static lastId: number = 0;  
  
    constructor() {  
    }  
}
```

And, staying true to its ES5-compatible nature, TypeScript diligently compiles this syntax down into the same exact approach that I demonstrated earlier: by attaching a variable directly to the `TodoService` constructor function.

```
TodoService.lastId = 0;
```

Likewise, I can also apply the `static` keyword to methods as well:

```
static getNextId() {  
    return TodoService.lastId += 0;  
}
```

Because it is all just the same code underneath, it's important to note that I consume the static property exactly as I did when I assigned it to the constructor function.

TodoService.ts

```
1 class TodoService {  
2  
3     static lastId = 0;  
4  
5     constructor() {  
6     }  
7  
8     add(todo: Todo) {  
9         var newId = TodoService.getNextId();
```

```
10     }
11
12     getAll() {
13     }
14
15     static getNextId() {
16         return TodoService.lastId += 0;
17     }
18 }
```

Notice how I'm accessing the static variable `lastId` on line 16 and the `getNextId` static method on line 9.

While static methods can be helpful to centralize small bits of common logic used by many components, static properties - even when they are attached to a class rather than defined in the global namespace - are still something that you should avoid at all costs since they tend to make your code a little more brittle, so they really should be your last resort. However, in scenarios such as this - when you need to guarantee a unique value, for instance - ECMAScript 2015 static members can be a very helpful feature.

Making properties smarter with accessors

In the previous chapter, I showed you how to use interfaces and object literals to define and instantiate simple objects with properties and even behaviors. Sometimes, however, you may need a property on an object to do a little bit more than just save a value and give that value back to you when you ask for it later.

For example, in the last chapter I used an interface to define a `Todo` object that looked like this:

```
interface Todo {
    name: string;
    state: TodoState;
}
```

Now, what if I wanted to enforce a specific workflow for `Todo` states. Something like: you can't change the state of a `Todo` to "Complete" unless its state is currently "Active" or "Deleted".

In order to accomplish this, I'd need something a little smarter than the basic property: I'd need to use a property with **getter and setter methods**.

Creating a property with getter and setter methods is pretty much exactly like it sounds. Take this `Todo` object literal from the previous chapter:

```
var todo = {  
  name: "Pick up drycleaning",  
  state: TodoState.Complete  
}
```

In order to convert the state property with getter and setter methods, I'll start by just introducing the getter method, which I'll implement by simply replacing the state property with a method called `state()` that accepts no parameters, and put the `get` keyword in front of that method, like this:

```
var todo = {  
  name: "Pick up drycleaning",  
  get state() {}  
}
```

That's the basic syntax, but in order for the getter to work it has to return something, so let's start by just returning the hard-coded value `TodoState.Complete`:

```
var todo = {  
  name: "Pick up drycleaning",  
  get state() {  
    return TodoState.Complete;  
  }  
}
```

At this point I want to point out that, even though I've been calling them "getter and setter properties" like they have to go together, this code is actually perfectly legitimate – you don't need to define a getter AND a setter for a property - you can implement either one or both of them, depending on what you're trying to accomplish.

In this scenario, however, the magic I'm trying to accomplish is in the setter method, which I define in the same way that I defined the getter method, except that the setter method must take in a parameter, which is the variable to be assigned... and, of course, it must be preceded by the "set" keyword rather than the "get" keyword that the getter uses.

For example:


```
var todo = {  
  name: "Pick up drycleaning",  
  get state() {  
    return TodoState.Complete;  
  },  
  set state(newState) {}  
}
```

That's the basic syntax, and it's actually perfectly legitimate to have a setter that doesn't have any logic, effectively ignoring any values that are assigned to it... however, most of the time that you're defining a getter and setter, the very least you're going to do is assign a value to an internal variable with the setter, and retrieve that value from the getter.

For instance:

```
var todo = {  
  name: "Pick up drycleaning",  
  get state() {  
    return this._state;  
  },  
  set state(newState) {  
    this._state = newState;  
  }  
}
```

Here I've dynamically created the property `_state` in order to have a variable that both the getter and setter have access to. Note that there's nothing special about this variable name - neither the underscore, nor the fact that I called it `state` - I could have called it `foo` if I wanted to. In fact, the same thing goes for the setter parameter, too: I happened to call it `newState`, but its name can be anything at all.

Once I have this in place, I access the `state` variable exactly as I did before. In other words, as a consumer I have no idea that it's implemented as a getter/setter property - I only see a regular old property.

That means I can set the value like this:

```
todo.state = TodoState.Complete;
```

And I can get the value like this:

```
todo.state;
```

Ok, so, this is a fully-working example of the getter/setter syntax in action: I've got a setter method that saves the value of a variable and a getter that gets that value back. But so far this really doesn't buy me anything - it's no more useful than just a regular old property.

So, let's revisit the `Todo` scenario I described earlier, in which I want to restrict the ability to set the state of the `Todo` to complete only when it is currently either `Active` or `Deleted`. In order to implement that, I might expand the setter logic to include something like this:

```
var todo = {
  name: "Pick up drycleaning",
  get state() {
    return this._state;
  },
  set state(newState) {

    if( newState == TodoState.Complete ) {

      var canBeCompleted =
        this._state == TodoState.Active
        || this._state == TodoState.Deleted;

      if(!canBeCompleted) {
        throw 'Todo must be Active or Deleted in order to be marked as C\
ompleted'
      }
    }

    this._state = newState;
  }
}
```

With this logic in place, every time the state property is assigned a value, the logic in the setter will execute and validate whether or not the new state can be assigned: if the `Todo` is in the `Active` or `Deleted` state, everything is fine and the `_state` property is updated. If not, an exception is thrown and the `_state` property remains in its current state.

Now that I've shown you how to take advantage of the getter and setter syntax, I have to admit something: you normally wouldn't use this approach on an object literal like I've been showing. It just doesn't make much sense. But, I've been defining these getter and setter properties on an object literal for two reasons:

1. Well, because that's what the `Todo` object has been in the previous examples and I didn't want to change it, and

2. To prove that the getter/setter syntax can be applied to object literals if you really want to.

The reason you generally don't want to apply getter/setter syntax to an object literal is because usually when you have this kind of logic you really want to put it in a class and create new instances of that class. And, when you do that, the getter/setter syntax starts to look a whole lot more applicable.

To demonstrate, I'll create a new class to house this behavior called `SmartTodo`, and I'll give it the one "normal" property that the `Todo` needs: `name`...

```
class SmartTodo {
  name: string;

  constructor(name: string) {
    this.name = name;
  }
}
```

Then, I'll copy the same getter/setter logic that I defined in the object literal:

```
class SmartTodo {
  name: string;

  constructor(name: string) {
    this.name = name;
  }

  get state() {
    return this._state;
  }

  set state(newState) {

    if( newState == TodoState.Complete ) {

      var canBeCompleted =
        this._state == TodoState.Active
        || this._state == TodoState.Deleted;

      if(!canBeCompleted) {
        throw 'Todo must be Active or Deleted in order to be marked as C\
ompleted'
      }
    }
  }
}
```

```
    }  
  
    this._state = newState;  
  }  
}
```

As soon as I do that, TypeScript yells at me for reasons that should be expected at this point in the course: I'm referring to a property named `this._state`, but that property doesn't actually exist on this class! Of course, that's easy enough to fix – just define it:

```
class SmartTodo {  
  _state: TodoState;
```

With that, the errors go away, and I can now begin using this class:

```
var todo = new SmartTodo("Pick up drycleaning");
```

```
todo.state = TodoState.New;
```

```
todo.state = TodoState.Complete // BOOM!
```

I have to admit that I don't use getters and setters much, since my personal preference is to stick with mostly simple objects and put this kind of logic into a service (such as the `TodoService` that I defined earlier). However, there are plenty of folks who prefer to have this kind of logic embedded directly in their class, as well as plenty of scenarios where this is a perfectly understandable approach to take. And, when you find yourself in need of properties that are more intelligent than just the basic “assign and retrieve” logic that regular properties gives you, the getter/setter method syntax is a great choice.

Inheriting behavior from a base class

In the last section I mentioned that I don't use getter/setter properties much because I prefer to have my logic in a separate class or service. That's because many times logic that may seem simple on the surface is not so simple when you start implementing it, and the scenario I gave in the previous section about only being able to change the `Todo`'s state when a certain condition is met is the perfect example.

It's the perfect example because I only mentioned the logic that gets enforced when attempting to change the `Todo` to a `Complete` state... but, wouldn't it make sense that if there were logic for the `Complete` state that there'd probably be logic for the other state changes as well? In these cases, I

might want to implement a design pattern called a “state machine” to represent the logic for each of these state changes as their own separate classes.

Now, I’m not going to explain the details of the “state machine” design pattern here - that’s a different book entirely. But, what I will say is that most of my state change classes are going to look a lot like this `TodoStateChanger` class:

```
class TodoStateChanger {  
  
    constructor(private newState: TodoState) {  
    }  
  
    canChangeState(todo: Todo): boolean {  
        return !!todo;  
    }  
  
    changeState(todo: Todo): Todo {  
        if(this.canChangeState(todo)) {  
            todo.state = this.newState;  
        }  
  
        return todo;  
    }  
}
```

The `TodoStateChanger` class accepts the desired end state as a constructor parameter on line 3, as well as two methods: `canChangeState` and `changeState` that provide the basic workflow for changing a state. The `canChangeState` contains basic logic to validate whether or not the `Todo` may be changed to the new desired state. Likewise, the `changeState` method first calls the `canChangeState` method to make sure it can proceed, and if it is safe, it updates the state on the `todo`.

In a class-based language like C# or Java, I would use the `TodoStateChanger` class as my base class - creating new classes that derived from it so they could share most of its logic, but also have the ability to override behavior as they see fit.

Lucky for us, not only does the ECMAScript 2015 syntax introduces the `class` keyword, it also introduces the ability for classes to inherit or “extend” from one another. So, let’s use that syntax to create a version of the `TodoStateChanger` that implements the `Complete` state change that I previously implemented in the getter/setter example.

The first thing to do is to create the new class that is going to extend from the `TodoStateChanger` class. Since I’m using it to execute the `Complete` state change, I’ll call it `CompleteTodoStateChanger` and start by defining it as a regular old class:

```
class CompleteTodoStateChanger {  
}
```

Then, inheriting from the `TodoStateChanger` class is as simple as adding the keyword `extends` after the class name, followed by the name of the class that I'd like to extend. In order case, this class is called `TodoStateChanger`.

```
class CompleteTodoStateChanger extends TodoStateChanger {  
}
```

Technically speaking, that's all you have to do to inherit from a class - now we have a brand new class named `CompleteTodoStateChanger` that contains all of the behavior defined on the `TodoStateChanger` base class. But, usually the reason for inheriting from a base class is to extend and/or override its behavior, which is exactly what we want to do here.

I'll start with the constructor function. In some other statically-typed languages like C#, whenever you derive from a class that has a constructor with a parameter, you must explicitly define your own constructor in the derived class. In other words, if we were trying to implement this same example in C#, I'd have to give `CompleteTodoStateChanger` a constructor.

Luckily, that's actually not true in JavaScript. If you choose to derive from a class that has a constructor and then choose to not create a constructor of your own, you simply inherit the constructor from the base class.

In other words, this would be perfectly valid given the code we're looking at right now:

```
var changer = new CompleteTodoStateChanger(TodoState.Complete);
```

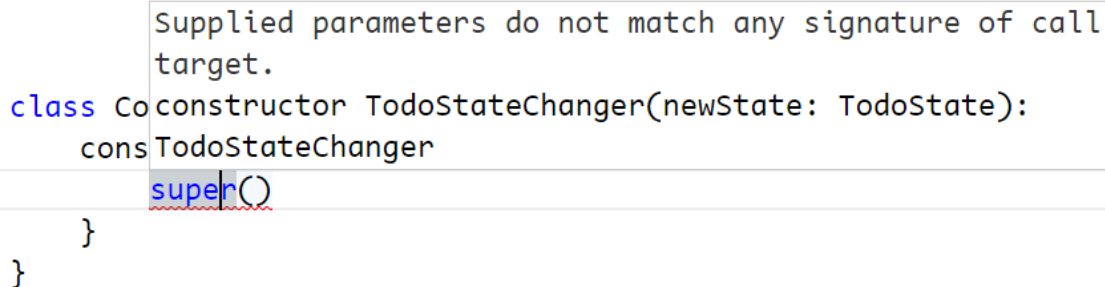
That's great for a lot of scenarios, but not for this scenario. In this scenario it doesn't make sense to set the new todo state with anything but `Complete`, so I'll give the `CompleteTodoStateChanger` its own constructor that does not allow the ability to pass in a todo state:

```
constructor() {  
}
```

However, there are two problems with this implementation:

1. You don't have to declare a constructor on a derived class, but every time you do, you have to call the constructor on the base class, and
2. In this particular case, we want to call the constructor on the base class anyway, to pass in the `Complete TodoState`

If I hover over the constructor function to see the error TypeScript is giving, I see that it's actually telling us exactly what I just said, but also a tip:



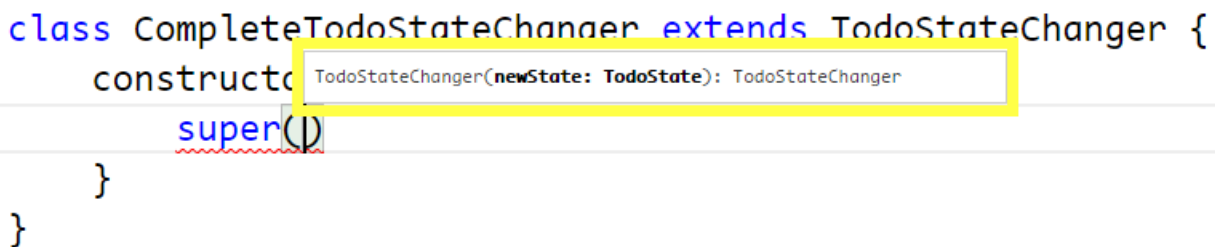
```

class Coconstructor TodoStateChanger(newState: TodoState):
  constructor() {
    super()
  }

```

super() error

It's saying that the way to call the constructor on the base class is the super function. So, I'll go ahead and add a call to the super function like it says.



```

class CompleteTodoStateChanger extends TodoStateChanger {
  constructor(newState: TodoState) {
    super(newState)
  }
}

```

super() IntelliSense

Notice how TypeScript gives me full IntelliSense on the super() function because it knows that I'm attempting to call the constructor on the TodoStateChanger base class.

```

constructor() {
  super(TodoState.Complete);
}

```

Ok, so now that I've called the constructor on the base class and told it that I always want to change the state to Complete, let's override the behavior of the base class to enforce the fact that Todo's must be in the Active or Deleted state in order to be eligible to change state to Complete. In other words, I want to override the logic in the canChangeState method on the TodoStateChanger base class, and I do that by simply creating a method with the same signature.

```

canChangeState(todo: Todo): boolean {
}

```

Notice how I don't have to use any special keywords like override or new tell TypeScript that I'm overriding the method like I would have to do with C# or other languages - I just go ahead and write the new method.

Now that I have this method in place, let me start by just reproducing the base class behavior – in other words, I don't want to override all of the logic in the base method, I just want to extend it a little bit. I can accomplish this by using the `super` keyword again, only this time `super` is the base class object as opposed to the base class constructor, so I can use it like this:

```
canChangeState(todo: Todo): boolean {  
    return super.canChangeState(todo);  
}
```

The `super` object is a reference that allows me to continue to access methods defined in the base class even though I'm overriding them.

With all this in place, I can go ahead and add the logic that's specific to the `Complete` state transition, specifically making sure that the `Todo` is either `Active` or `Deleted`:

```
canChangeState(todo: Todo): boolean {  
    return super.canChangeState(todo) && (  
        todo.state == TodoState.Active  
        || todo.state == TodoState.Deleted  
    );  
}
```

And that is how you inherit from a base class in TypeScript! Actually, once again, everything in this example other than the static type information is really just ECMAScript 2015 syntax - even the `extends` keyword. However, one of the things that JavaScript and ECMAScript 2015 does not support is the ability to define and inherit from abstract base classes – or base classes that are only defined in order to act as base classes and never actually meant to be instantiated by themselves. Luckily, TypeScript does support abstract classes, and if you check out the next section, I'll show you how to implement them!

Implementing an abstract class

In the previous section, I showed you how to use the ECMAScript 2015 class syntax to create an inherit from a base class in order to share behavior between classes. And, I left off in that section by pointing out that even though ECMAScript 2015 doesn't support the idea of abstract base classes, TypeScript does! In this section, I'll show you how to use TypeScript to implement the power concept of abstract base classes in JavaScript.

Let's do a quick recap: in the last section, I defined two classes: `TodoStateChanger`, and `CompleteTodoStateChanger`, which derived from `TodoStateChanger`. Since both of these were defined as classes, that means that I could create an instance of `TodoStateChanger` directly if I wanted to – there's

nothing stopping me from doing that right now. However, let's say that my intent was that this class was only ever intended to serve as a base class and never instantiated directly.

Luckily, TypeScript offers the `abstract` keyword to help me express that intent. By simply placing the `abstract` keyword in front of the class definition like this...

```
abstract class TodoStateChanger {
```

I can still derive from the class, but TypeScript will consider it an error any time I try to instantiate that class anywhere in my code.

```
new TodoStateChanger();
```

What's more, I can express my intent even more by describing any member of the class as `abstract` simply by applying the `abstract` keyword in front of it as well.

For example, let's say that while I do want the `TodoStateChanger` to define the shared behavior in the `changeState` method, I want to force every derived class to implement their own `canChangeState` method from scratch, and not even implement the basic logic that I currently have. In order to accomplish that, I simply add the `abstract` keyword to the beginning of the method:

```
abstract canChangeState(todo: Todo): boolean {
```

When I do that, of course, the implementation is no longer needed, so I can simply remove it:

```
abstract canChangeState(todo: Todo): boolean;
```

Now TypeScript will enforce my intent that every derived class must implement its own `canChangeState` method. Of course, the derived class I've already created - `CompleteTodoStateChanger` already does implement this method, so we're good there... but, the current `canChangeState` implementation is referencing the base implementation... which doesn't exist anymore!

Accordingly, TypeScript is yelling at us for making a call to a method that doesn't exist, and we can fix this by simply removing the reference:

```
canChangeState(todo: Todo): boolean {  
    return !!todo && (  
        todo.state == TodoState.Active  
        || todo.state == TodoState.Deleted  
    );  
}
```

Once I do so, everything is fine again and I have now enforced the fact that my abstract base class may only be used as a base class. Well... throughout my TypeScript code at least. Since this is all getting compiled down to JavaScript and sent to the browser, there's nothing stopping some rouge JavaScript code from creating a new instance of the class, but like everything TypeScript is designed to guard me against, there's ultimately nothing I can really do about that. All I can do is focus on the protection that it gives at development time, and in that regard abstract base classes are an excellent tool.

Controlling visibility with access modifiers

A few sections ago, I showed you how to apply the `private` keyword to a constructor parameter to automatically assign or “upgrade” that parameter to a property on the class, as I show here:

```
constructor(private todos: Todo[]) {  
}
```

But, the `private` keyword isn't just a way to create properties - if you have ever used a class-based language such as Java or C# then you might know that the `private` keyword is an access modifier, used to hide members of a class from being accessed outside of that class.

And, just like those other languages, you can apply access modifiers to any member of a TypeScript class.

You can apply it to constructor parameters as a shortcut to automatically turn it in to a member, like I've already shown...

You can apply it to methods:

```
private getAll() {
```

And, you can apply it to properties (even static ones!):

```
private static lastId = 0;
```

You can also apply them to getters and setters:

```
private get state() {  
private set state(newState)
```

However, keep in mind that both the getter and setter must have the same access modifier applied. In other words, TypeScript won't let you make the setter private and the getter public (which, unfortunately, is exactly what I often want to do!).

Now, I've been applying the `private` modifier in all of my examples so far, but TypeScript actually offers three access modifiers for you to describe how much you want to protect the members of your classes: `private`, `protected`, and `public`.

The `private` modifier is the most restrictive modifier of the three. Placing the `private` modifier on a member means that only methods defined directly on the same class definition may access that member. TypeScript will complain if any other type - including types that inherit or extend from this class - attempt to access the member.

Next, there's the `protected` modifier, which is similar to the `private` modifier in that only methods defined on the same class may access the member, but expands this definition to any classes that inherit or extend from this class. In other words, it only applies to the base class examples that I showed in the previous two sections. And, actually, it would have been really useful for the scenario that I have been describing because right now the constructor on the `TodoStateChanger` base class sets the `newState` property, but since it's `private` none of the derived classes are able to access it!

```
abstract class TodoStateChanger {  
  
    constructor(private newState: TodoState) {  
    }  
}
```

However, if I update this to `protected`...

```
abstract class TodoStateChanger {  
  
    constructor(protected newState: TodoState) {  
    }  
}
```

the auto-assigned `newState` property will be accessible to all derived classes, which is exactly what I'd expect to happen.

Finally, there's the `public` modifier, which is the least restrictive of all. A member with public access may be accessed from any other type. In other words, the `public` modifier just describes the default JavaScript behavior, which also means that it's the default access modifier so you probably won't see it much.

The one case you might see the `public` modifier more often is actually the same scenario that began this whole conversation: applying it to a constructor parameter to automatically create a property from that parameter.

For example, in the getter/setter section I created a new class named `SmartTodo` that looked like this:

```
class SmartTodo {  
    name: string;  
  
    constructor(name: string) {  
        this.name = name;  
    }  
}
```

You may have wondered at that time why I manually assigned the `name` parameter in the constructor to the `name` property on the `SmartTodo` class and didn't use the private constructor parameter feature to define and assign the property all at once. Well, I would have, except that at the time I'd only shown you how to use the `private` keyword to accomplish that, and if I'd have used the `private` keyword then the automatically-created `name` property would not have been accessible outside of the class! But now that I've introduced the concept of access modifiers, I can revise this example to use any of the three modifiers - `private`, `protected`, or `public` - to automatically create a property with the same name and level of access.

In other words, I can rewrite the `SmartTodo` constructor like this:

```
class SmartTodo {  
    constructor(public name: string) {  
    }  
}
```

Now that I've explained what the three modifiers mean and how to use them, I have to point out that there's a problem with all of this, and that is that JavaScript doesn't actually support private members. Remember, JavaScript objects are just bags of dynamic properties, and once you add a property to the bag, that property is accessible to everyone who has access to that object, and that's all there is to it.

TypeScript is a superset of JavaScript which means that it doesn't change how JavaScript works, so how can TypeScript support private when JavaScript does not? Well, the short answer is, it doesn't. Much like interfaces and specifying a type at the end of a variable or method doesn't change how the JavaScript gets generated, the two access modifiers that TypeScript supports - `private` and `protected` - are strictly for your protection during development and have absolutely no effect at runtime in the browser.

But, that doesn't mean they're not worth using. Keep in mind that one of the most important reasons for static typing is to express the intent of your code. Since JavaScript has never supported the idea of private members, developers have come up with their own ways to express their intent for a property to only be used by the component itself and not by anyone else.

Perhaps the most widely-used convention is to put an underscore in front of these variables, like this:

```
private static _lastId = 0;
```

Then, if you find yourself calling a member on another class whose name starts with an underscore, then you're probably doing something wrong.

And, hey, if you like that convention, feel free to keep on using it! The only thing that TypeScript changes in this equation is that it won't even list private members in its autocomplete options. And, will consider it an error if you attempt to access another object's private members. Other than that, if you really, really want to access another object's private members, go ahead and write the code to do so - TypeScript won't stand in your way. However, much like the rest of the static typing capabilities that TypeScript provides, it's a great idea to use access modifiers to express your intent when you don't want other components interacting directly with certain members of a type. Even if TypeScript can't enforce this behavior at runtime, I've found that enforcing it during development is good enough for all but the most exceptional scenarios.

Implementing interfaces

In the last chapter I showed you how to define interfaces and use them to make sure that the objects you reference have the members that you expect them to. But, the primary reason that interfaces exist in a statically-typed language is to attach them to classes. And, now that I've shown you how to create classes, it's time to see just how to apply interfaces to them to make sure that they behave like we expect them to.

Before I show you how to apply an interface to a class, I just want to point out something that I think is pretty great: because TypeScript understands and implements JavaScript's native duck-typing behavior, you don't actually have to do anything in order to make use of an interface. By simply applying the interface to the variables and return values in which you expect that interface to be present, TypeScript will automatically inspect every object that you attempt to assign to that variable or attempt to return as the return value. When the object's structure matches the interface you've defined, then it really doesn't matter what type the object is - everything is fine! However, if the object's structure does not match the interface - it is missing properties or methods that the interface expects, or the types of the properties don't match - only then will TypeScript yell at you that you've attempted to use an object that doesn't match.

All that said, keep in mind the overwhelming theme of this book: even though you don't have to apply static type information in order for TypeScript to work, the more explicit you are, the better! In other words, even though TypeScript can figure out that that a class satisfies the needs of an interface, if you intend for that class to implement that interface then you should just go ahead and say it!

Throughout this book, I've created several interfaces and classes; in particular, the `ITodoService` interface and the `TodoService` class. As you may have guessed by their names, I intend for the `TodoService` to implement the `ITodoService` interface, but at this point the two are not explicitly connected... so let's go ahead and do that now. Luckily, that's pretty easy to do - just use the `implements` keyword after the class name, and then the interface that the class must implement:

```
class TodoService implements IToDoService {
```

As soon as I do this, TypeScript starts to enforce the interface and gives me an error saying that the `TodoService` does not implement the `IToDoService` interface. This is because the `IToDoService` interface defines 4 methods - `add`, `delete`, `getAll`, and `getById` - but the `TodoService` class that I showed earlier only implements the `add` and `getAll()` methods. This is easy enough to fix by just implementing those other missing methods.

First, I have to add an `id` property to the `Todo` interface so that I can reference `Todo` items by their ID. That part's easy.

```
interface Todo {  
  id: number;  
  name: string;  
  state?: TodoState;  
}
```

Then, I'll update the `add` method to set that ID property using the `nextId` getter method that I introduced in the getter/setter section.

```
add(todo: Todo) {  
  todo.id = this.nextId;  
  this.todos.push(todo);  
  return todo;  
}
```

After it gives the `Todo` a unique ID, it adds it to the list of `Todos` and then returns the `Todo` as the return value.

Next there's the `getAll()` method.:

```
getAll(): Todo[] {  
  return JSON.parse(JSON.stringify(this.todos));  
}
```

In previous examples I just let this method return the array of `todos` directly, but this is actually a bad practice since consumers of this method could choose to manipulate the `Todo` items themselves and I want to control any changes that happen to them in this class. So, instead I'll make a copy of the `Todo` items by serializing them to JSON and then parsing them back out again. I know that this seems like it might be expensive, but it's actually a pretty performant way to clone objects!

Then there's the `getById` method, which takes a number indicating the ID of the `Todo` item to find in the list of `Todos`.

```
getById(todoId: number): Todo {  
    var filtered = this.todos.filter(x => x.id == todoId);  
  
    if(filtered.length) {  
        return filtered[0];  
    }  
  
    return null;  
}
```

In this method I use JavaScript's built-in Array method `filter` to filter the list down to only the `Todo` objects whose ID property matches the `todoId` value that was passed in. Notice how I'm passing an arrow function to the `filter` method rather than typing out a full-fledged function the long way as I would have had to do in the past. If the filter method call finds at least one item, I return the first item. Otherwise, I'll return null.

Finally, there's the `delete` method, which accepts a number indicating the ID of the `Todo` item to delete.

```
delete(todoId: number): void {  
    var toDelete = this.getById(todoId);  
  
    var toDeleteIndex = this.todos.indexOf(toDelete);  
    this.todos.splice(toDeleteIndex, 1);  
}
```

In this method I start by calling the `getById` method to find the actual `Todo` object, then I use the JavaScript Array method `indexOf` to figure out where the object is in the array. Finally, I pass this index to the JavaScript Array method `splice` to remove the `Todo` item from the array.

And that's it! Now that I've implemented those methods, TypeScript is happy that I've satisfied all of the members of the `ITodoService` interface and stops giving me an error. Now that I have this interface in place, if I ever change any of these methods so that it doesn't accept the right parameters or returns the wrong type, TypeScript will be sure to let me know.

Also, if you're wondering: yes, it's possible for a class to implement multiple interfaces at once. When you want to specify multiple interfaces, you can add them all after the `implements` keyword, separating each one with a comma.

For example, if I have an interface named `IIDGenerator` that describes a type that has a `nextId` property, like this...

```
interface IIdGenerator {  
    nextId: number;  
}
```

I can add this to my list of interfaces on the `TodoService` class like this:

```
class TodoService implements ITodoService, IIdGenerator {
```

Then, of course, I have to implement this interface, but since the `TodoService` already has a `nextId` property it already implements the interface so there is nothing to be done and TypeScript is happy.

And that wraps up the chapter on classes! At this point in the book, I've shown you the majority of the features and syntax that you're likely to consistently leverage while using TypeScript. The remainder of this book is going to dive deeper into some of the more advanced features that TypeScript provides, but I think you'll actually be quite surprised with how much just these few features can bring meaning and structure to your application code so it's important that you understand everything that I've described up to this point if you're going to start working with TypeScript on a daily basis. That's not to say that the advanced features I'm going to show you in the rest of the book are unnecessary or irrelevant – quite the contrary! Consider these features to be the icing on the cake - they may not be required, but including them makes everything so much better!

8. Generics

Creating generic functions

So far in this book I've showed you how to use TypeScript to implement a number of object-oriented principles in JavaScript – principles that are generally reserved for statically-typed languages such as C# and Java. But, not only does TypeScript allow you to implement concepts like encapsulation and inheritance of an abstract base class, there is one more powerful feature that both Java and C# offer that TypeScript offers as well, and that is generics.

Generics are a way to create functions and classes that define a behavior that can be reused across many different types while retaining the full information about that type.

For example, take this function, which clones an object using the JSON API to serialize the object to a JSON string and then deserialize it back into a new instance of the same object:

```
function clone(value) {  
    let serialized = JSON.stringify(value);  
    return JSON.parse(serialized);  
}
```

Now, looking at this we can see that if it works as I just described then the return value of this function should return the exact same type of object that passed in. In other words, the return value should be the same type as the “value” object.

However, since the JSON.parse API accepts a string that can represent any type of object, there is absolutely no way for it - or TypeScript - to even guess at what type of object is going to come out when that string is serialized. It could literally be any type at all.

By applying generics to this function we can solve two problems at once: not only can we give TypeScript the type information that it needs in order to determine the return type, we can also reuse this same exact code for any and all types in our application!

The syntax for defining a generic function is pretty straight-forward: you simply add “greater-than” and “less-than” signs right after the function name, but before the parentheses containing the parameter list, and give your generic type parameter a name, like this:

```
function clone<T>(value) {
```

What this syntax does is tell TypeScript that you will be referring to a generic type in this function and you'll be referring to the type by the name "T". Note that I'm using the name "T" here simply because that's the name that many people use by convention, but you can feel free to give it any valid variable name that you like, for instance: "X", or "Tinput".

With this generic type defined, I can use it throughout this method any place I'd use a regular type name: like on the "value" parameter, for instance:

```
function clone<T>(value: T) {
```

Now when I hover over the "value" parameter to see its type information, I can see that TypeScript thinks it's type is "T". And, the whole point of using it in this example is to tell TypeScript that the return type will always be the same as the input type, so I'll specify "T" as the return type as well:

```
function clone<T>(value: T): T {
```

What is "T"? Well, it's the type of the parameter that you pass as the "value", of course! In other words, "T" is not any specific type at all, but is determined by TypeScript each and every time you call this function, and solely depends on how you use the function...

For example, when I pass in a string as the value...

```
clone("Hello!")
```

"T" is the string type.

Or, when I pass in a number...

```
clone(41)
```

"T" is the number type.

And, of course, it works with custom types as well:

```
let todo: Todo = {  
  id: 1,  
  name: 'Pick up drycleaning',  
  state: TodoState.Active  
};  
  
clone(todo)
```

Here's where it gets interesting though... What's the return value when I pass in an object literal that has a "name" property?

```
clone({ name: 'Jess' })
```

Well, TypeScript is smart enough to tell me that it's an unnamed type that has a name property! Pretty cool, huh?

Even though this example may seem simple, don't let it fool you: generics are an incredibly powerful and useful tool in the right situations. So, whenever you see a place in your application that you seem to be copying the same code over and over again, and all you're doing differently in each version is simply changing which type name you're using, then that might be a great opportunity to reduce that duplicated code into a single generic function.

Creating generic classes

In the previous section, I introduced you to the concept of generics and showed you how to use them to define the input and output types of a function. But, functions aren't the only place where generic types can come in handy - you can apply them to classes, too!

In fact, you've already seen a generic class in action, you just didn't realize it at the time. Among other classes, TypeScript treats the built-in JavaScript Array type as a generic class. In other words, in previous chapters I used this syntax to indicate an array of number values:

```
var array: number[] = [1,2];
```

But you can also use this syntax as another way of writing the same thing:

```
var array: Array<Number> = [1,2];
```

Note that in this example these two syntaxes are actually representing the exact same thing so they are completely functionally equivalent. It just so happens I like to use the first syntax, but as this example proves, you can choose whichever syntax you prefer!

Similar to the generic Array type, one of the best scenarios I've seen generics come in handy is the typed Key/Value pair class which contains two properties - a key and a value property, and looks like this:

```
class KeyValuePair<TKey, TValue> {  
    constructor(  
        public key: TKey,  
        public value: TValue  
    ) {  
    }  
}
```

Notice how in this class accepts two generic type parameters - TKey, and TValue - each separated by a string. In fact, technically speaking, you can define as many generic types as you need, however you generally don't tend to go over two.

With this class in place, I can create instances of the key/value pair object using any two types of values I want:

```
let pair1 = new KeyValuePair(1, 'First');  
let pair2 = new KeyValuePair('Second', Date.now());  
let pair3 = new KeyValuePair(3, 'Third');
```

If I hover over each of these instances, I can see that TypeScript is inferring the generic type parameters based on the values that I passed in. For instance, it knows pair1 and pair3 are instances of KeyValuePair<number, string> because I passed in a number and a string value as the first and second parameter for each. Likewise, it knows that pair2 is an instance of KeyValuePair<string, Date> because I passed in a string value and a Date as the first and second parameters.

If I wanted to, I could explicitly tell TypeScript which types I wanted by applying the same generic type syntax as I did to the class, only instead of the generic type parameter names "Tkey" and "Tvalue" I'll use the names of the actual types that I want, like this:

```
let pair1 = new KeyValuePair<number, string>(1, 'First');  
let pair2 = new KeyValuePair<string, Date>('Second', Date.now());  
let pair3 = new KeyValuePair<number, string>(3, 'Third');
```

And, look at that - by explicitly defining the types that I intended I actually caught a bug in my code when I defined pair2. Oddly enough, the JavaScript Date.now() function doesn't actually return a Date object - it returns the number of milliseconds elapsed since 1 January 1970! So, I have to take that number and turn it into a Date and everything is fine again:

```
let pair2 = new KeyValuePair<string, Date>('Second', new Date(Date.now()));
```

Now that I have my variables I can begin using them. And, the cool thing is that since the types are determined on the fly and TypeScript will remember what those generic type parameters were through the rest of my code:

```
pair2.key. [AUTOCOMPLETE]
```

So when I attempt to access them later I continue to get full type support, like I can see here that I am getting this list of string members on the key value that I had passed as a string.

This kind of approach is one way that you can use generic classes, but it gets even more interesting. Now that I've defined a generic Key/Value pair type, I can also define a class that can interact with any instances of that type.

For example, I can create a generic class that iterates through a collection of any type of KeyValuePair objects and prints them out to the console:

```
class KeyValuePairPrinter<T, U> {

    constructor(private pairs: KeyValuePair<T, U>[]) {

    }

    print() {

        for( let p of this.pairs ) {
            console.log(`${p.key}: ${p.value}`);
        }

    }

}
```

I start by passing in two generic type parameters to the class (just like I did with the KeyValuePair class itself) on line XX, which I call “T” and “U”.

Then, I define a constructor that accepts an array of KeyValuePair's that all share the same two types “T” and “U”.

And in that method I iterate through each KeyValuePair and print out their “key” and “value” properties on line XX.

Even though printing the values to the output may not be too compelling of a scenario, the interesting thing about this example is that I have statically-typed access to the objects “key” and “value” properties - regardless of what types “T” and “U” are - because I know that they are all “KeyValuePair” objects!

Now let's see it in action:

```
let printer = new KeyValuePairPrinter([ pair1, pair3 ]);
printer.print();
```

In this example, I create a new instance of the printer and use it to print out two `KeyValuePair` objects. Notice that I'm only passing in the `pair1` and `pair3` variables. This is because they share the same key and value types - in other words, they both have a number for a key and a string for a value. And, because they share those same values, those are the types that TypeScript uses as the types for the "T" and "U" generic type parameters.

In other words, TypeScript now knows that this printer object is an instance of `KeyValuePairPrinter<number, string>` [hover over printer variable] and therefore only accepts `KeyValuePairs` with the generic parameters of number and string.

On the other hand, if I tried to pass in "pair2", TypeScript would give me an error, warning me that `pair2` does not share the same type as `pair1` and `pair3`:

```
let printer = new KeyValuePairPrinter([ pair1, pair2, pair3 ]);
```

This is because `pair2` doesn't share the same generic type parameters - its key is a string and its value is a `Date`. And, since it has different generic type parameters it is, for all intents and purposes, a completely different type of object altogether.

If generic classes seem a little overwhelming at first, I don't blame you - I was a little overwhelmed with them, too. But, they are a great way to group generic methods that all operate on the same types of objects.

And, now that you've seen generic classes in action, check out the next section where I'll show you how to place constraints on your generic types to limit the type parameters that consumers of your generic functions or classes can apply.

Creating generic constraints

In the previous few sections I introduced you to generics and showed you how to apply them to both functions and classes. In this section, I'm going to show you some of the more advanced things that you can do with generic type parameters to control how consumers of your generic functions and classes can use them.

First, let me jump back to an example that I showed a few chapters ago: the "totalLength" function that adds the "length" property of two objects together:

```
function totalLength(x: { length: number }, y: { length: number }) {  
    var total: number = x.length + y.length;  
    return total;  
}
```

As I mentioned when I last showed this example, the way this method is currently implemented there's nothing stopping me from trying to add the length of a string value to the length of an array value... which makes absolutely no sense:

```
var length = totalLength('Jess', [1, 2, 3])
```

However, over the past few sections I showed you how to use generic type parameters to define a type in a generic way and then apply that type throughout your function or class. In other words, I can force consumers of this method to pass in two values of the same type but replacing the anonymous types with generic types:

```
function totalLength<T>(x: T, y: T) {  
    var total: number = x.length + y.length;  
    return total;  
}
```

By doing that, I've solved one problem, but I've created another: I've removed the type information saying that the type T must have a length property on it! Luckily, TypeScript offers the concept of generic constraints that I can apply to my generic type parameters to restrict the kinds of values that satisfy those parameters.

The generic constraint syntax is simple. In fact, it involves a keyword you've already seen. In order to constraint the types that satisfy a generic type parameter, I simply follow the parameter with the "extends" keyword, followed by the type describing the constraint that I want to put in place.

For example, before I just added the generic type parameter to the "totalLength" method, it only accepted objects that had a "length" field. To apply that same constraint to the generic type parameter, I simply use the anonymous type again:

```
function totalLength<T extends { length: number }>(x: T, y: T) {  
    var total: number = x.length + y.length;  
    return total;  
}
```

With this constraint in place, I finally have everything I want: a method with code that supports a myriad scenarios, but also enforcing the fact that you can't add the lengths of two completely unrelated types!

And, of course, I don't have to use anonymous types as my generic type constraints - any type will do, including interfaces, classes, or even primitive types. For instance, I can move this anonymous type into a formal interface if I wanted to:

```
interface IHaveALength {
    length: number
}

function totalLength<T extends IHaveALength>(x: T, y: T) {
    var total: number = x.length + y.length;
    return total;
}
```

Using an interface instead of the anonymous type may be a different syntax, but it works out to be the same exact constraint.

That's the basic overview of generic type constraints, but there are a few caveats to keep in mind...

First, I have to clarify something: for the past few minutes I've been saying that both parameters in this example must be of the same type - type "T". But that's not strictly true - they can be any type that is compatible with type T, including those types that inherit from it.

For example, let's say that I use the method to add two arrays together:

```
var length = totalLength([1, 2], [1, 2, 3])
```

That's fine - nothing new there. But, how about if I create my own custom class that extends the base Array class, like this:

```
class CustomArray<T> extends Array<T> {
    toJson(): string {
        return JSON.stringify(this);
    }
}
```

Well, because it inherits from the Array class, it is an instance of the Array class, so it matches the same generic type parameter "T":

```
var length = totalLength([1, 2], new CustomArray<number>())
```

There's a second caveat with generic type constraints and unfortunately this one's not as nice a surprise as the fact that I can use inherited types to satisfy the same generic type. In fact, it's a little bit of the opposite...

Remember in the last section when I created the "KeyValuePairPrinter"?


```
class KeyValuePairPrinter<T, U> {  
  
    constructor(private pairs: KeyValuePair<T, U>[]) {  
    }  
  
    print() {  
  
        for ( let p of this.pairs ) {  
            console.log(`${p.key}: ${p.value}`)  
        }  
    }  
}
```

When I showed you that class, did you find it a little awkward that I defined the generic types T and U on the class, and then the only place I used them was to pass them along to define the generic types on the KeyValuePair object? I found it awkward.

Frankly, I would have rather included the KeyValuePair type in the class's type constraint somehow... Something like this, maybe:

```
class KeyValuePairPrinter<V extends KeyValuePair<T, U>>
```

But, if I try to do that, TypeScript tells me that it doesn't know about the types U and V because they weren't declared. Ok, that's fair. So how about this?

```
class KeyValuePairPrinter<T, U, V extends KeyValuePair<T, U>>
```

Sadly, that doesn't work, either, because - as TypeScript reminds me in its warning - you're not allowed to refer to generic parameters that you define in the same type list... Ah well, I guess the original way was really the best I could've done...

However, don't let these kinds of limitations discourage you from using generics or even generic constraints. Used in the right way on the right problems, generic functions and classes can elevate your code from good to elegant.

And, speaking of elegant code, why don't you head on to the next chapter where I'll introduce you to another powerful concept: modules.

9. Modules

JavaScript has been around a long time - over 20 years, as of the writing of this book! And, only in recent years has the industry really started to take JavaScript seriously and started to apply patterns, practices, and development standards when working in the browser. A few sections ago, I pointed out the problem that is still chief among the issues that lingers today - putting all of your code in the global namespace.

Putting all your code in the global namespace encourages you to create all sorts of dependencies between components simply because you can. You need to share a value between two classes? Put it into a global variable! Why not - both components have access to it, right?

This kind of coding practice produces something that is affectionately nick-named “spaghetti code” because your entire application just becomes one giant ball of intertwined threads without any way of telling where one component ends and another begins.

To avoid spaghetti code, many first-class languages have some kind of mechanism to modularize code - to keep components separate from one another, and distinguish between two components that may share the same name but are otherwise completely different.

Given the absence of any JavaScript syntax prior to ECMAScript 2015 to support the modularization of code, the development community had to get creative and come up with a variety of design patterns to both encapsulate the internal workings of components and help organize growing codebases.

Among the many great resources that define these patterns is the book “Learning JavaScript Design Patterns” by Addy Osmani. The physical version of this book is available for purchase, however you can find the entire manuscript for free online - just check the course notes for the link.

In this book, Addy discusses several concepts that are particularly relevant when it comes to modularizing and organizing JavaScript codebases. The first is two different design patterns - the module pattern and the revealing module pattern. The next is the relatively simple and straightforward concept of namespaces. And, the third is the slightly more complex concept of modules and module loaders.

In the rest of this chapter, I will explain all of these concepts to you, as well as show you how ECMAScript 2015 and TypeScript enable you to implement these best practices in your current codebases.

So, if you’re interested in learning about how to organize your code, check out the next chapter where I will introduce you to the simplest modularization concept: **namespaces**.

Organizing Your Code with Namespaces

In the previous section, I mentioned that TypeScript offers a few ways to better encapsulate and organize your code. In this section I'm going to show you the simplest of the techniques called "namespaces".

Many modern programming language have a concept of namespaces - C, C++, Java, and C#, to name just a few - and the reason that namespaces are so popular is because they are an excellent way to avoid naming collisions and refer to a group of types as one organizational unit.

And, if you've used C, C++, and C# the TypeScript namespace syntax should feel incredibly familiar to you. It's simply the keyword `namespace`, followed by the name of the namespace - any valid JavaScript variable name will do. For example:

```
namespace Model {  
}
```

Or, if you like you can add some periods to add a hierarchy of namespaces all at once:

```
namespace TodoApp.Model {  
}
```

Now, let's take a look at the code that is generated when you do this:

```
'Model.ts'
```

Well, actually, it's nothing... Since there is no code in the namespace it has no functional effect so TypeScript simply ignores it when it comes to compiling the code to JavaScript.

So, I'll add some code. Let's start by moving the `Todo` interface in there:

```
namespace TodoApp.Model {  
  interface Todo {  
    id: number;  
    name: string;  
    state: TodoState  
  }  
}
```

If I look again, there's still no code generated since interfaces also have no functional effect and get ignored.

‘Model.ts’

So, now I’ll get serious and add an enum which does require code to interact with at runtime:

```
namespace TodoApp.Model {  
  
    interface Todo {  
        id: number;  
        name: string;  
        state: TodoState;  
    }  
  
    enum TodoState {  
        New = 1,  
        Active,  
        Complete,  
        Deleted  
    }  
  
}
```

Which generates the following:

‘Model.js’

```
var TodoApp;  
(function (TodoApp) {  
    var Model;  
    (function (Model) {  
        var TodoState;  
        (function (TodoState) {  
            TodoState[TodoState["New"] = 1] = "New";  
            TodoState[TodoState["Active"] = 2] = "Active";  
            TodoState[TodoState["Complete"] = 3] = "Complete";  
            TodoState[TodoState["Deleted"] = 4] = "Deleted";  
        })(TodoState || (TodoState = {}));  
    })(Model = TodoApp.Model || (TodoApp.Model = {}));  
})(TodoApp || (TodoApp = {}));
```

Note that you are free to declare the same namespace multiple times, either in the same file or throughout multiple files, allowing you to organize your code however you like.

For example, I can split the `TodoApp.Model` namespace definition up into two parts:

```
namespace TodoApp.Model {  
  
    interface Todo {  
        id: number;  
        name: string;  
        state: TodoState;  
    }  
  
}
```

```
namespace TodoApp.Model {  
  
    enum TodoState {  
        New = 1,  
        Active,  
        Complete,  
        Deleted  
    }  
  
}
```

Or, I can even define a whole new namespace in the same file:

```
namespace TodoApp.DataAccess {  
  
    interface IToDoService {  
        add(todo: TodoApp.Todo): Todo;  
        delete(todoId: number): void;  
        getAll(): Todo[];  
        getById(todoId: number): Todo;  
    }  
  
}
```

Notice how - as soon as I put these types into their own namespace declarations - TypeScript begins to warn me that it doesn't know about the `Todo` type in the `IToDoService` interface definition. In the next section I'll get into the technical explanation as to why that happens even when two types share the same namespace, but the short answer is because in order to use types outside of the scope that you define them in, you need to expose them using the `export` keyword, like this:

```
export interface Todo {

export enum TodoState {

export interface IToDoService {
```

Now, that helps clean up reference between `Todo` and `TodoState` because they are in the same namespace, but I still have an issue when I try to refer to types across namespaces even when I export them, as is the case where the `IToDoService` in the `ToDoApp.DataAccess` namespace is trying to refer to the `Todo` type in the `ToDoApp.Model` namespace.

This is because I'm still referring to the global type `Todo`, which no longer exists. It no longer exists because I took it out of the global scope when I moved it into its own namespace – which is the whole point of applying a namespace. Going forward, I now have to refer to it by its fully-qualified name, including the namespace.

In our example that means changing all references of `Todo` to `ToDoApp.Model.Todo`:

```
export interface IToDoService {
  add(todo: ToDoApp.Model.Todo): ToDoApp.Model.Todo;
  delete(todoId: number): void;
  getAll(): ToDoApp.Model.Todo[];
  sgetById(todoId: number): ToDoApp.Model.Todo;
}
```

Alternatively, I can create an alias for the namespace or the type by using the `import` statement in my namespace declaration to import a type from another namespace:

```
namespace DataAccess {

  import Todo = ToDoApp.Model.Todo;

  export interface IToDoService {
    add(todo: Todo): Todo;
    delete(todoId: number): void;
    getAll(): Todo[];
    getById(todoId: number): Todo;
  }

}
```

Either approach works just fine, and it's up to you which approach you want to you for any given scenario. The general rule of thumb that I follow is to import a type or namespace if the namespace

is very long or if you are going to refer to it more than a once or twice, as is the case in this example. Likewise, whether or not you choose to import the whole namespace or just one type at a time is completely up to you.

In this section, I showed you how to apply TypeScript's namespace syntax to better organize your types and get them out of the global namespace. However, I started off the section by mentioning that organization is only one of the benefits of applying namespaces. In the next section I'll show you how to take advantage of the other benefit that TypeScript namespaces provide and that is encapsulation and creating truly private variables and types.

Using namespaces to encapsulate private members

In the previous section I showed you how to leverage TypeScript's namespace syntax to better organize your application code and get it out of the global scope. In this section, I'll dive into how, exactly, namespaces work and how you can leverage them to define truly private variables and types.

I'll start by checking out the code that gets generated when you define a TypeScript namespace.

```
// This...
namespace TodoApp.Model {
    export enum TodoState {
        New = 1
    }
}

// Gets generated as this:
var TodoApp;
(function (TodoApp) {
    var Model;
    (function (Model) {
        var TodoState;
        (function (TodoState) {
            TodoState[TodoState["New"] = 1] = "New";
        })(TodoState || (TodoState = {}));
        Model.TodoState = TodoState;
    })(Model = TodoApp.Model || (TodoApp.Model = {}));
})(TodoApp || (TodoApp = {}));
```

Take a good look, and keep this code in mind as I introduce you to a new JavaScript design pattern called an “Immediately-Invoked Function Expression”, (or “IIFE” for short). An IIFE is a popular pattern to encapsulate code while it executes and then choose which parts of the code will be exposed to the rest of the world by specifying them as the return value. Its syntax is pretty straight-forward:

First, create a normal function:

```
function defineType() {  
}
```

Then, immediately follow the function with a set of parentheses, which immediately executes the function:

```
function defineType() {  
}()
```

And, most implementations wrap the function in a set of parentheses, too, just to be safe:

```
(function defineType() {  
})()
```

You can take the IIFE approach one step further by passing an object into the function that the function may refer to. The function then interact with that object either by referring to variables or functions on that object or building up the object by attaching additional members to it.

For example, I could rewrite my earlier jQuery plugin example like this:

```
var jQuery = {  
    version: 1.19,  
    fn: {}  
};  
  
(function createType($){  
  
    if($.version < 1.15)  
        throw 'Plugin requires jQuery version 1.15+'  
  
    $.fn.myPlugin = function() {  
        // plugin  
    }  
  
})(jQuery)
```

This example starts off by defining a variable named `jQuery` and giving it two properties: a version number and an `fn` object which I can attach custom functions to.

Notice how I did something a little tricky here and changed the name of the variable as it got passed in. Outside of the IIFE, the variable is named `jQuery` and that's the object I pass in to the function when it gets immediately invoked on line 14. However, within my function I wanted to refer to

the object using only the dollar sign, so I used the dollar sign as the name of the variable instead of calling it `jQuery`. The name really doesn't matter, though – even though it's called `$` within the function and `jQuery` outside of the function, it still refers to the same exact object.

So, what do IIFEs have to do with TypeScript namespaces? Well, let's go back to the code that TypeScript generated for the namespace declaration that I wrote:

```
var TodoApp;
(function (TodoApp) {
    var Model;
    (function (Model) {
        var TodoState;
        (function (TodoState) {
            TodoState[TodoState["New"] = 1] = "New";
        })(TodoState || (TodoState = {}));
        Model.TodoState = TodoState;
    })(Model = TodoApp.Model || (TodoApp.Model = {}));
})(TodoApp || (TodoApp = {}));
```

Even though it's a whole lot more code, what you're looking at here is actually just a bunch of IIFEs! That's right - a TypeScript namespace is really nothing more than syntactic sugar to create an IIFE! That means that everything you write within the namespace declaration is automatically kept within the same scope and not allowed to leak out of that scope unless you explicitly expose it yourself.

This is why, in the previous section, I had to apply the `export` keyword to allow the `Todo` interface to reference the `TodoState` enum: even though they belonged to the same namespace, they were declared in two different namespace declarations and were private by default. The way the `export` keyword works is that it tells TypeScript to attach the type to the namespace object so that it can be accessed anywhere in the application - even in other declarations of the same namespace.

To really understand what's happening, let's walk through the generated code, line by line:

1. The first thing to notice is that TypeScript generates the global object to hold the top-level namespace. In our example, this is the `TodoApp` variable on line 1.
2. The declaration of the global variable on line 1 is then followed by the beginning of the IIFE on line 2. I'll get into the guts of the IIFE next, but for now I'll jump all the way to line 13, where the IIFE ends and is immediately invoked. Look at the expression used to define the parameter being passed into the IIFE on this line: it basically says "if the global variable `TodoApp` already has a value, use that, otherwise, assign that global variable to an empty object and pass that in instead." Regardless of whether the object existed before the declaration or not, by the time it gets passed as a parameter to the IIFE function on line 2, it has been defined.
3. Since the `TodoApp` namespace is empty, there is no logic in it other than the declaration of the nested namespace `Model`, which follows the same exact approach to create the `TodoApp.Model` object as I just described for the `TodoApp` object.

4. Finally, we get to the good stuff inside of the `TodoApp.Model` declaration. Here TypeScript generates the same code that we saw before to define the `TodoState` enum. And now that we know about the IIFE design pattern, we can see that enums and classes are declared using this pattern as well! The most interesting part of the `Model` namespace declaration is where the `TodoState` enum is exposed to the public by assigning it as a property of the `Model` namespace object. This line is the direct result of the `export` keyword, which I can prove by removing the `export` keyword and watching this line disappear: `Model.TodoState = TodoState; .` And, when I add the `export` keyword back, the line reappears.

When it comes to encapsulation, there are two very important points about what I just told you:

1. Things declared within a namespace remain private unless you explicitly expose them, and
2. When it comes down to it, namespaces are really just functions; special, powerful functions, sure... but just functions... and that means that you can define more than just types within them. You can also use them to create private variables and functions.

With that in mind, let's see how I can take my `TodoService` and refactor it so that the private static variable and method that I'm currently using to generate unique IDs for my `Todo` items is truly inaccessible from other components in my application as opposed to simply telling TypeScript to yell at me whenever I try to access it.

I'll start by moving the whole `DataAccess` namespace into its own file and copying the `TodoService` class into it:

DataAccess.ts

```
namespace DataAccess {  
  
    import Model = TodoApp.Model;  
    import Todo = Model.Todo;  
  
    export interface ITodoService {  
        add(todo: Todo): Todo;  
        delete(todoId: number): void;  
        getAll(): Todo[];  
        getById(todoId: number): Todo;  
    }  
  
    class TodoService implements ITodoService {  
  
        private static _lastId: number = 0;  
  
        get nextId() {
```

```
        return TodoService._lastId += 1;
    }

    constructor(private todos: Todo[]) {
    }

    add(todo: Todo): Todo {
        todo.id = this.nextId;

        this.todos.push(todo);

        return todo;
    }

    delete(todoId: number): void {
        var toDelete = this.getById(todoId);

        var deletedIndex = this.todos.indexOf(toDelete);

        this.todos.splice(deletedIndex, 1);
    }

    getAll(): Todo[] {
        var clone = JSON.stringify(this.todos);
        return JSON.parse(clone);
    }

    getById(todoId: number): Todo {
        var filtered =
            this.todos.filter(x => x.id == todoId);

        if( filtered.length ) {
            return filtered[0];
        }

        return null;
    }
}
```

Then, I simply cut the `lastId` variable from inside the class and move it outside...

DataSource.ts

```
private static _lastId: number = 0;

class TodoService implements ITodoService {

    private static _lastId: number = 0;

    get nextId() {
        return TodoService._lastId += 1;
    }
}
```

Remove the private and static keywords, which are no longer relevant, and replace them with a simple let statement:

DataSource.ts

```
private static _lastId: number = 0;
let _lastId: number = 0;

class TodoService implements ITodoService {

    get nextId() {
        return TodoService._lastId += 1;
    }
}
```

Finally, I'll clean up any references to the static variable that are now broken.

Because the lastId variable is defined in the same scope that my class is defined in, my class can simply reference the variable by name:

DataSource.ts

```
let _lastId: number = 0;

class TodoService implements ITodoService {

    get nextId() {
        return TodoService._lastId += 1;
        return _lastId += 1;
    }
}
```

I can even go one step further and replace the getter property method the same way: cut it, paste it outside of the class, and convert it into a normal function.

While I'm at it, I'll rename it to generateTodoId():

DataSource.ts

```
let _lastId: number = 0;

function generateTodoId() {
    return _lastId += 1;
}

class TodoService implements ITodoService {

    get nextId() {
        return _lastId += 1;
    }
}
```

Then I'll update the reference to the getter method with a call to this private function instead...

DataSource.ts

```
add(todo: Todo): Todo {
    todo.id = this.nextId;
    todo.id = generateTodoId();
}
```

The end result of all of this is that I've got two private members that live outside of my class, completely encapsulated from the rest of the world, but fully accessible to any classes or functions that are defined within the same namespace declaration. Assuming that I really, really wanted to hide the implementation details of how I am generating new Todo IDs, this would be a great way to do it.

While incredibly effective, the namespace approach - also referred to as the **internal module** approach - is only one way that TypeScript enables you to encapsulate your components. The other way, referred to as the "external module" approach is equally effective and actually far more common in browser-based lazy-loading solutions such as require.js and in the NodeJS development world. To see what I'm talking about, check out the next section to see how implement the external module approach in TypeScript.

Understanding the difference between Internal and External Modules

In the previous section, I showed you how to leverage TypeScript namespaces, otherwise known as "internal modules".

But, even though the internal module approach can be very effective, it's actually fallen out of grace

in recent years in favor of **external modules**, and in this section I'll explain what external modules are and how you can start using them in your TypeScript applications.

Namespaces and the internal module approach allows you to group components together and get them out of the global scope by bringing them together under one umbrella object. Using TypeScript, you can control which components belong to which object by wrapping them with namespaces using the syntax that I demonstrated in the last section:

```
namespace TodoApp.Model {  
    export interface Todo {  
        id: number;  
        name: string;  
        state: TodoState  
    }  
}
```

And in that last section I also showed how you can use that same syntax to define multiple namespaces in the same file.

Put simply, the external module approach has most of the same goals of encapsulation and organization as the internal module approach, but its implementation is a bit different. Rather than using a keyword and brackets to indicate the scope of the module, the external module approach uses the file itself as the module scope.

Even under this approach, many of the same restrictions apply and perhaps the most important one of all is that all components defined within the module are only accessible to that file by default and must be exported in some way in order to be used.

The inverse is also true: any components defined outside of the module must be imported in order for the module to access them. This is in stark contrast to the traditional JavaScript development methodology of defining everything in the global scope, and even though I showed you in the last section how to use the `import` keyword to make an alias to another type or namespace, it's really more of a shorthand than anything else – you don't have to import another internal module in order to use it - you just... use it.

In the next few sections, I'm going to show you two different syntaxes that TypeScript provides for you to import modules: the “require” syntax (similar to the syntax that NodeJS uses) and ES2015 (the syntax that's now part of the ECMAScript standard).

You may be scratching your head wondering why TypeScript offers two different syntaxes to import modules, and there's a pretty simple answer for that: ECMAScript 6 (more commonly referred to as ECMAScript2015 when referring to the module syntax) defined its syntax after TypeScript had already implemented the `require` syntax. Since it couldn't just drop support for a syntax that it's had for a long time, now TypeScript supports both syntaxes.

As I'm showing you these two syntaxes, the thing to remember as you learn each of them is that they are completely functionally equivalent, by which I mean that TypeScript ends up generating the same exact code for them at compile time.

Ultimately, you can choose either syntax and be quite fine! However, as with most of the features that are in the ECMAScript specification, I would recommend using the ECMAScript syntax as this is the syntax that will eventually have native browser support.

Regardless of which syntax you choose, TypeScript supports two ways to import external modules so I'm going to show you how to use both of them. Either way, there is a little bit of configuration and preparation that you'll need to do first, so head on to the next section where I'll show you what you need to take care of in order to start using the external module approach.

Switching from internal to external modules

In the previous section I explained the concept of external modules and how they compare to the internal modules that I already demonstrated.

And, as I mentioned in that section, perhaps the biggest difference between the two module approaches is that external modules use the file that they're defined in as their module border as opposed to an IIFE within a file. This important difference means that it is kind of redundant to apply internal namespaces within external modules, because the file already is the module.

For example, in the namespace section I introduced a namespace called `DataAccess` inside a file named `DataAccess.ts`. Once we switch to external modules in the next two sections, I'll need to reference any types defined in this module by going through the external module first, then through the internal module, then finally get to the type I was looking for. In other words, I'll end up writing this:

```
DataAccess.DataAccess.TODOService
```

And that's just silly.

So, before I get into the next sections, I'm going to prepare by removing the internal modules that I added previously, but leave the `export` keyword on their public members because that syntax is the same syntax that you continue to use when exposing members in the external module approach. I'll also remove those import statements that I was using as aliases because I'll be replacing them in just a bit.

.DataAccess.ts

```
namespace DataAccess {  
  
    import Model = TodoApp.Model;  
    import Todo = Model.Todo;  
  
    // ... Rest of file  
  
}
```

As I make these changes, I see that TypeScript is getting upset with me about the fact that I'm still using the `export` keyword even though I'm not defining any modules.

That is because in order to leverage the external module approach, I have to tell TypeScript that that's what I'm doing.

I can tell TypeScript this and make these errors go away by adding another configuration parameter to my TypeScript configuration located in my `tsconfig.json` file:

The option I'm looking to add is the `module` compiler setting:

tsconfig.json

```
{  
    "compilerOptions": {  
        "target": "es5",  
        "module": "system",  
    }  
}
```

As I add this setting, Visual Studio Code offers several different autocomplete options for me to choose from: CommonJS, AMD, System, UMD, etc.

Keep in mind that this setting is really just telling TypeScript that I intend to use external modules by indicated what the compiled output should be.

As I mentioned in the previous section, you are free to use either syntax you like when actually defining the import, so it really doesn't matter which option you choose here.

I've chosen the `system` setting because I'll be implementing the `System.js` module loader in the next chapter, but you can feel free to choose whichever one you like.

Switching back to the `DataAccess` module I can see that the errors about the `export` keyword have gone away, but now it has no idea what the `Todo` type is.

That's because I need to import that type from the `Model` namespace, so head on to the next section where I'll show you how to use the `require` syntax to do exactly that.

Importing modules using Require syntax

In the previous few sections, I've been introducing the concept of external modules and explained that TypeScript supports two syntaxes for importing these modules: the require syntax and ECMAScript 2015 syntax and, in this section, I'll be explaining the require syntax. The require syntax is pretty simple, so I'll just go ahead and use it to show you what it looks like.

When I left off in the last section, TypeScript was giving me errors in the `DataAccess.ts` file because it didn't know about the `Todo` type and I needed to import that type in order to make it happy again.

In order to do that, I simply write the following statement at the beginning of my file in order to reference the `Model` module...

```
import Model = require('./Model');
```

This brings everything exported in the `Model` module into the scope of this file, all contained under the `Model` object, very similar to the internal module approach I showed earlier.

Notice that, while I did specify the path to the file I was looking for, I did not specify the extension of the file – in other words, I didn't say `Model.js` or `Model.ts`. This is deliberate, and I'll explain why in more detail in the next chapter when I discuss how modules are actually loaded, but for now this is an important part of the syntax to keep in mind.

And, like the internal module approach, I have two ways to reference components in this module: through the module object itself or by establishing a shortcut to them by giving them an alias.

For instance, if I want to refer to the `Todo` type, I can change every individual reference, like this:

```
import Model = require('./Model');

export interface IToDoService {
    add(todo: Model.Todo): Model.Todo;
    delete(todoId: number): void;
    getAll(): Model.Todo[];
    getById(todoId: number): Model.Todo;
}
```

Or... I can establish one alias to the type, like this:

```
import Model = require('./Model');
import Todo = Model.Todo;

export interface IToDoService {
  add(todo: Todo): Todo;
  delete(todoId: number): void;
  getAll(): Todo[];
  getById(todoId: number): Todo;
}
```

You might think it's a little annoying to have to do this in two lines – I sure do. In other words, I'd love to be able to write this:

```
import Todo = require('./Model').Todo;
```

Sadly, TypeScript doesn't support this usage, so you'll have to keep the two lines in place in order to create an alias like this.

```
import Model = require('./Model');
import Todo = Model.Todo;
```

However, the ECMAScript 2015 syntax *does* allow you to import individual types and create aliases for them all in one line. In fact, it's one of the reasons that I prefer that syntax over this one.

To see what I'm talking about, check out the next section where I show you everything you need to know in order to use the ECMAScript 2015 module syntax.

Importing modules using ECMAScript 2015 syntax

A few sections ago, I mentioned that TypeScript supports two different syntaxes for importing external modules - the `require` syntax and the ECMAScript 2015 syntax - and in the last section, I showed you how to use the `require` syntax. In this section, I'll show you how to use the ECMAScript syntax to import components from external modules. And, in fact, the ECMAScript 2015 syntax is actually pretty close to the “`require`” syntax.

As I showed in the previous section, you import a module with the “`require`” syntax like this:

```
import Model = require('./Model');
```

But, with the ECMAScript 2015 syntax, you would write the same thing like this:

```
import * as Model from './Model';
```

That's really all the difference! Once I've got the module referenced, the rest of the code works exactly the same way.

In other words, in order to refer to the `Todo` type in the `Model` module, I need to refer to it with its full name, like this:

```
let todo: Model.Todo;
```

But, remember how, in the previous section, I used the `import` statement to give the `Todo` type an alias, like this?

```
import Todo = Model.Todo;  
let todo: Todo;
```

Well, that same syntax will continue to work and I could use it here, but it's not actually part of the ECMAScript 2015 spec... and that's for a good reason: because the ECMAScript spec allows you to define aliases in the `import` statement itself!

To demonstrate, I'll first remove the alias:

```
import * as Model from './Model';  
import Todo = Model.Todo;
```

Then, I'll set my focus on the original `import` statement, in particular the `* as Model` part.

What this part of the syntax is saying is, "import everything that is being exported from this module and attach it to a temporary object named `Model` so that I can refer to them in this file.

However, importing *everything* in the module like this is rarely what you want to do.

More often, you only need a few exports from a module, and in those cases you'll want to import them individually, which you do by replacing this part of the statement with a set of brackets, like this:

```
import { } from './Model';
```

Then, inside of these brackets, you define every individual export that you want to import by name, separated by commas. In my example I just want to import the `Todo` type, which I can do like this:

```
import { Todo } from './Model';
```

As soon as I do that I can see it's working because the TypeScript warnings about not knowing what the `Todo` type is go away.

Likewise, if I wanted to import more exports that this module provides, I could add them to the list, like this:

```
import { Todo, TodoState } from './Model';
```

Or, if you prefer to give the type an alias for when you're working with it in this file, you can use that as syntax that I showed in the original import statement.

For example, if I didn't like the name of the type `Todo` and I wanted to call it `TodoTask` instead, I could do this:

```
import { Todo as TodoTask, TodoState } from './Model';
```

Notice that as soon as I do this, TypeScript begins warning me again that it doesn't know about a type named `Todo` – that's because I just renamed it... well, in this file at least. In order to get rid of that error I have to use the new alias that I gave it, `TodoTask`:

```
let todo: TodoTask;
```

And, finally, it's important to point out that you are not required to import anything at all, in which case you simply say "import module name", like this:

```
import './Model';
```

Really the only scenario you'd want to do this is when you have a script that modifies the environment in some way that you are dependent on. In those cases, the import statement is still relevant because you do depend on that module getting loaded... you're just not relying on any particular exports that the module provides.

And that's how you use the ECMAScript module syntax! As I've mentioned several times now, even though this is one of the two syntaxes that TypeScript supports, I strongly recommend using the ECMAScript syntax if no other reason than it is a standard that will eventually have native browser support.

Now that I've demonstrated both syntaxes, head on to the next section where we'll revisit that module configuration setting and see how it affects the code that TypeScript generates.

Loading External Modules

In the previous few sections I introduced you to the concept of external modules and two different syntaxes that are available to you to import one module from another. I also mentioned that - regardless of which of the two syntaxes you used to import modules - the code that gets generated is not only exactly the same, but also driven by a setting in the TypeScript configuration file `tsconfig.json`. In this section, I'll not only talk about what each of those configuration options means, but I'll also explain how you can actually use the external module approach in your application.

I'll start by explaining what I meant by that last part about "actually using the external module approach". You see, as of this writing, even though ECMAScript 2015 defines the syntax for importing one module into another, there are two problems:

1. The syntax is not widely supported in browsers just yet, and
2. There is no standard implementation for actually loading the modules when they're referenced. As of this writing there is an ECMAScript loader specification that is in the process of gaining approval, but that means that native support for module loading is still very far off.

The net result of those two problems is that, until both of them are fixed, you're probably going to want to introduce a new tool into your toolbox and that is a module loader.

Choosing a module loader is easier said than done because there are plenty of options available and all of them have different ways of doing things. This is why TypeScript provides so many options for the `module` configuration setting: because there is one for every major module loader that's available! So, the short answer to the question of which module configuration setting you should choose is: which module loader do you want to use?

Obviously, teaching you how to use every module loader out there is clearly outside the scope of this book (not to mention a whole lot of work). So, through the rest of the book I'll be using the `System.js` module loader in order to load module files and manage module imports at runtime. The reason I'm choosing `System.js` over the others should come as no surprise – I'm choosing it because it is the loader that attempts to implement the proposed ECMAScript specification. In other words, my hope is that I can use it as a temporary solution until the browsers actually implement the ECMAScript proposal, at which time I should be able to simply drop the loader and use the native implementation. At least, I hope... in reality, I'll probably end up needing to use `System.js` for longer than I expect, but that's ok because at least it's letting me write ECMAScript 2015 code now.

Now that I've chosen my module loader, I've got to update my `index.html` to change the way I'm loading my application.

Before using a module loader, I had to include every single JavaScript file explicitly, creating a `<script>` tag for each one. But, with a module loader, the only script you need to specify in your HTML is the module loader itself, then use the module loader's API to call into your application module.

In other words, I can remove all of these script tags and replace them with a single reference to `system.js`.

`index.html`

```
// All other script tags removed...  
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/systemjs/\0.19.22/system.js"></script>
```

Notice that I've taken a shortcut here and linked directly to the `system.js` file hosted on the Cloudflare CDN. This should work for you, but if it doesn't you'll need to download it and put it into your project locally, or use the npm command line tool to install the "systemjs" package:

```
npm install systemjs
```

Regardless of how you get System.js loaded, the next thing you'll do is use it to load your module with this simple line (placed in a `<script>` block, of course):

```
<script type="text/javascript">
  System.import('app')
</script>
```

Note that at the time of this writing, I also had to add this configuration statement before the import call in order for it to work properly:

```
<script>
  System.defaultJSExtensions = true;
  System.import('app')
</script>
```

Hopefully by the time you're reading this the TypeScript compiler or System.js will have been updated to remove the need for this line.

Once I have all this in place, I should be able to run the site and see it in action. Unfortunately, right now it doesn't do anything, so let me write a little bit of code that loads the `TodoService` from the `DataAccess` module, initializes it with a `Todo`, and then prints out all the `Todos` onto the command line:

```
import { Todo, TodoState } from './Model';
import { TodoService } from './DataAccess';

let service = new TodoService([]);

service.add({
  id: 1,
  name: 'Pick up drycleaning',
  state: TodoState.New
})

let todos = service.getAll();

todos.forEach(todo =>
  console.log(`${todo.name} [${TodoState[todo.state]}]`)
)
```

Now when I run the site and open up the console I should be able to see my `Todo` printed out indicating that everything has worked!

And with that, we've got a working external module implementation! Now, if you found this chapter perhaps a little overwhelming or even complicated, well... I don't blame you. It actually took me a while to get comfortable with modules, too. It's not really a concept that we JavaScript developers have really had to know until now! But, that's ok for two reasons:

1. If you take the time to learn how to use external modules properly, your code will become far more organized and easy to manage, and
2. If you really don't like the external module approach then you don't have to use it! You can still use namespaces to implement an internal module approach that doesn't require adding any more complexity or tools to your existing environment and yet still get the benefits of getting your code out of the global namespace!

It's up to you what you decide to do, but I know for me, spending the time learning how to embrace external modules has really made my codebases far more enjoyable to work with, and I think it will be worth your time to learn, too.

10. Real-World Application Development

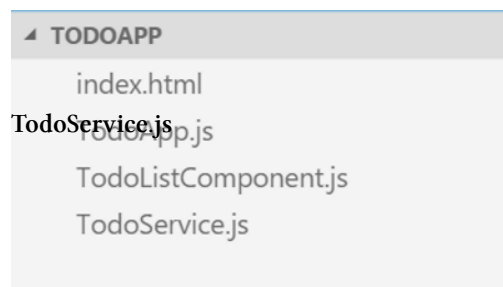
So far in this book I've shown you a variety of features that TypeScript offers which allow you to introduce powerful static typing capabilities to JavaScript as well as write ECMAScript 2015 code that compiles into code that runs in pretty much any modern browser.

Even though I haven't shown you every nuance that TypeScript has to offer, I've shown you the 95% of it that you'll use in your day-to-day development.

But, at this point I'd like to shift the focus from the syntax that TypeScript provides and move toward the mindset that you will need when using TypeScript to create a full-blown JavaScript application - regardless of whether it's converting an existing application or starting a new one from scratch.

Throughout several chapters in the book, I've referred to the fact that I am going to build a sample Todo application to demonstrate how to bring all of TypeScript's features together into a real, working application, and up to this point I've only been showing bits and pieces.

Well, in this chapter, I'm finally going to show you that complete, working application... except I'm going to start with the application written in ECMAScript 5 syntax and then apply each of the TypeScript features I've shown you in this book in order to convert it into the TypeScript version of the same application.



ECMAScript 2015 Todo Application

So, let's walk through this sample application. We'll start with the `index.html` page of the Todo application.

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1">
  <title>TypeScript Todo List</title>
  <link rel="stylesheet" href="//maxcdn.bootstrapcdn.com/bootstrap/3.3.7/css/bootstrap.min.css">
  <link rel="stylesheet" href="styles.css">
</head>

<body>
  <div class="container">
```



```

<div class="header clearfix">
  <h1>TypeScript Todo List</h1>
</div>
<div class="row">
  <div class="panel panel-default clearfix">

    <div class="panel-body">

      <div class="row">
        <div class="todo-list col-md-12">
          <!-- Todo Items rendered here -->
        </div>
      </div>

      <br>

      <div class="row">
        <div class="col-md-10">

          <!-- Add Todo -->
          <form class="add-todo">
            <div class="form-group">
              <div class="input-group">
                <span class="input-group-addon">Add</span>
                <input type="text" class="form-control input-lg" placeholder="What do you need to get done?" />
              </div>
            </div>
          </form>
          <!-- /Add Todo -->

        </div>

        <div class="col-md-2">
          <!-- Clear completed button -->
          <button class="clear-completed btn btn-default" type="button">Clear Completed</button>
          <!-- /Clear completed button -->
        </div>
      </div>
    </div>
  </div>
</div>

```

```

        </div>
    </div>
</div>
</div>

<script type="text/javascript" src="//code.jquery.com/jq
</script>
<script type="text/javascript" src="TodoListComponent.js
<script type="text/javascript" src="TodoService.js"></sc
<script type="text/javascript" src="TodoApp.js"></script>
<script type="text/javascript">
    new TodoApp(document, [
        'Pick up drycleaning',
        'Clean Batcave',
        'Save Gotham'
    ]);
</script>
</body>

</html>

```

It's really nothing fancy - just some standard HTML and CSS (using the Bootstrap CSS library that I introduced in the beginning of the book), and a couple of script tags with references to ECMAScript 5 JavaScript files.

For example, here's the ECMAScript 5 version of the `TodoService` that we've been looking at during this whole book - a great example of building a "class" using the prototype object before ECMAScript 2015 introduced the concept of classes.

`TodoService.js`

```

var TodoService = (function () {

    // Private, static variable
    var _lastId = 0;

    // Private, static method
    function generateTodoId() {
        return _lastId += 1;
    }

```

```
}

// Private, static method
function clone(src) {
  var clone = JSON.stringify(src);
  return JSON.parse(clone);
};

// Constructor method
function TodoService(todos) {
  var _this = this;

  this.todos = [];

  if(todos) {
    todos.forEach(function(todo) {
      _this.add(todo);
    })
  }
}

// input parameter can be either string or object ({ name })
TodoService.prototype.add = function (input) {

  var todo = {
    id: generateTodoId(),
    name: null,
    state: 1
  };

  if(typeof input === 'string') {
    todo.name = input;
  }
  else if(typeof input.name === 'string') {
    todo.name = input.name;
  }
  else {
    throw 'Invalid Todo name!';
  }

  this.todos.push(todo);

  return todo;
}
```

```
};

TodoService.prototype.clearCompleted = function () {

  this.todos = this.todos.filter(function(x) {
    return x.state == 1;
  });
}

TodoService.prototype.getAll = function () {
  // Don't return the actual array - clone it instead
  return clone(this.todos);
};

TodoService.prototype.getById = function (todoId) {
  var todo = this._find(todoId);
  // Don't return the actual object - clone it instead
  return clone(todo);
};

TodoService.prototype.toggle = function (todoId) {

  var todo = this._find(todoId);

  if(!todo) return;

  switch(todo.state) {
    case 1:
      todo.state = 2;
      break;
    case 2:
      todo.state = 1;
      break;
  }
}

// "Private" method (just because it starts with an _)
TodoService.prototype._find = function (todoId) {
  var filtered = this.todos.filter(function (x) {
    return x.id == todoId;
  });
};
```

```

    if (filtered.length) {
        return filtered[0];
    }

    return null;
}

// Return the class from the IIFE
return TodoService;
})();

```

Next, I've got the `TodoListComponent` class - a class that takes an HTML element and a list of `Todo` objects and uses the `jQuery` library to render them as HTML.

`TodoListComponent.js`

```

var TodoListComponent = (function () {

    function TodoListComponent(el) {
        this.$el = $(el);
    }

    TodoListComponent.prototype.render = function (todos) {

        this.$el.html('');

        if (!todos.length) {
            this.$el.html(
                "<div class='list-group-item text-center text-giant'" +
                "  <strong>You've completed everything you needed to do!</strong>" +
                "</div>"
            );

            return;
        }

        for(var index in todos) {
            var todo = todos[index];
            renderTodo(todo).appendTo(this.$el);
        }
    };

    function renderTodo(todo) {

```

```

    return $(
      "<div class='todo-item list-group-item '" + (todo.state == 2 ? 'completed' : \
      '') + "'>" +
      "    <div class='row'" +
      "      <div class='col-md-2 text-center'" +
      "        <i class='incomplete glyphicon glyphicon-unchecked text-muted text-\
      -giant'" +
      "          <i class='completed-indicator completed glyphicon glyphicon-ok tex\
      t-giant'" +
      "        </div>" +
      "      <div class='col-md-10'" +
      "        <span class='incomplete text-giant'" + todo.name + "</span>" +
      "        <span class='completed text-strikethrough text-muted text-giant'" + \
      + todo.name + "</span>" +
      "      </div>" +
      "    </div>" +
      "  <div class='clearfix'" +
      "  </div>"
    ).on('click', function() {
      var event = document.createEvent('CustomEvent');
      event.initCustomEvent('todo-toggle', true, true, { todoId: todo.id });
      this.dispatchEvent(event);
    });
  }

  return TodoListComponent;
})();

```

This class also triggers a custom DOM event called `todo-toggle` on line 44 whenever the user clicks on the Todo item.

In other words, this class is simply a UI component that handles the rendering of Todo items and lets other components know about the user's actions by raising a DOM event.

If you've used jQuery before, you probably would have implemented this differently and created a jQuery plugin. And, quite honestly, I probably would have, too. But, I wrote it this way so that this class is the only class in the application that uses jQuery, which will be important when I convert everything to the external module approach.

And, finally, the `TodoApp` class, which orchestrates the DOM, the `TodoService` and the `TodoListComponent` together to create a working application.

TodoApp.js

```
function TodoApp(el, todos) {
  this.todoService = new TodoService(todos);
  this.initialize(el);
}

TodoApp.prototype.addTodo = function (todoName) {
  this.todoService.add(todoName);
  this.renderTodos();
}

TodoApp.prototype.clearCompleted = function () {
  this.todoService.clearCompleted();
  this.renderTodos();
}

TodoApp.prototype.toggleTodoState = function (todoId) {
  this.todoService.toggle(todoId);
  this.renderTodos();
}

TodoApp.prototype.renderTodos = function () {
  var todos = this.todoService.getAll();
  this.todoList.render(todos);
}

TodoApp.prototype.initialize = function (el) {
  var _this = this;

  var addTodoFormEl = el.getElementsByClassName('add-todo')[0],
      addTodoNameEl = addTodoFormEl.getElementsByTagName('input')[0],
      todoListEl = el.getElementsByClassName('todo-list')[0],
      clearCompletedEl = el.getElementsByClassName('clear-completed')[0];

  addTodoFormEl.addEventListener('submit', function (evnt) {
    _this.addTodo(addTodoNameEl.value)
    addTodoNameEl.value = '';
    evnt.preventDefault();
  });

  todoListEl.addEventListener('todo-toggle', function (evnt) {
    var todoId = evnt.detail.todoId;
```

```
    _this.todoService.toggle(todoId);
    _this.renderTodos();
  });

  clearCompletedEl.addEventListener('click', function () {
    _this.clearCompleted();
  });

  this.todoList = new TodoListComponent(todoListEl);

  this.renderTodos();
}
```

Now that you've got the lay of the land, follow me through the next few sections as I apply all of the TypeScript features I've shown in this book to bring this application from ECMAScript 5 to TypeScript. Because I've already shown you all of these features, I'll be moving through the changes pretty quickly, but I think that watching a "classic" JavaScript application evolve into a TypeScript application is a very helpful way to understand the true power of TypeScript... and I've still got a couple of cool things that I haven't shown you yet. So, head on to the next section where I show you how to convert an existing JavaScript file into a TypeScript file.

Converting existing JavaScript code to TypeScript

Perhaps my favorite aspect of TypeScript is that it is a superset of JavaScript which means that "migrating" existing JavaScript applications to TypeScript is pretty much as simple as adding a `tsconfig.json` file, renaming files from `.js` to `.ts`, then updating them to use the new ECMAScript 2015 syntax. And, in this section, that's exactly what I'm going to do to the sample Todo application I just walked you through.

The first step in creating or converting any TypeScript application is to define the `tsconfig.json` file, and luckily this really is as simple as I showed in the earlier sections:

`tsconfig.json`

```
{
  "compilerOptions": {
    "target" : "es5"
  }
}
```

Next, I'll jump to the command line and start up the TypeScript compiler in watch mode and leave it running while I make the rest of my changes:


```
tsc -w
```

Then I can start renaming and upgrading my JavaScript files.

Let's start the `TodoService.js` file.

I've chosen this file because it is at the heart of my application – it's the thing keeping track of all of the Todo items. That means, that if I convert the `TodoService` class first then when other classes interact with it, TypeScript will be able to use type inference to help push static typing throughout my application, even if I forget to add the type information in the consuming components.

To convert it to TypeScript, start by renaming the file from `TodoService.js` to `TodoService.ts`.

Once I've done that, I'll convert the prototype-based approach into a class:

`TodoService.ts`

```
class TodoService {

    private static _lastId = 0;

    private static generateTodoId(): number {
        return TodoService._lastId += 1;
    }

    private static clone(src) {
        var clone = JSON.stringify(src);
        return JSON.parse(clone);
    };

    private todos = [];

    constructor(todos) {
        var _this = this;

        if(todos) {
            todos.forEach(function(todo) { _this.add(todo) });
        }
    }

    add(input) {

        var todo = {
            id: TodoService.generateTodoId(),
            name: null,
            state: 1
```

```
};

if(typeof input === 'string') {
  todo.name = input;
}
else if(typeof input.name === 'string') {
  todo.name = input.name;
} else {
  throw 'Invalid Todo name!';
}

this.todos.push(todo);

return todo;
};

clearCompleted() {

  this.todos = this.todos.filter(
    function(x) { x.state == 1; }
  );
}

getAll() {
  return TodoService.clone(this.todos);
};

getById(todoId) {
  var todo = this._find(todoId);
  return TodoService.clone(todo);
};

toggle(todoId): void {

  var todo = this._find(todoId);

  if(!todo) return;

  switch(todo.state) {
```

```
        case 1:
            todo.state = 2;
            break;

        case 2:
            todo.state = 1;
            break;
    }
}

private _find(todoId) {
    var filtered = this.todos.filter(
        function(x) { x.id == todoId; }
    );

    if (filtered.length) {
        return filtered[0];
    }

    return null;
}
}
```

Notice that I'm implementing the `lastId` counter as a static variable because I want to keep it out of the global scope for now. When I switch to the external module approach in a little while, I'll come back and move these into the module scope, exactly like I described in the external module chapter.

Next, I'll introduce some interfaces to define a few custom types. For now, I'll just define them right here in the file, and then I'll move them when I'm done.

TodoService.ts

```
1 interface Todo {
2     id: number;
3     name: string;
4     state: TodoState;
5 }
6
7 enum TodoState {
8     Active = 1,
9     Complete = 2
10 }
```

Hey, those types look familiar, don't they?

And, now that I have those custom types, I'll apply them to the variables, parameters, and return values throughout my class.

TodoService.ts

```
interface Todo {
  id: number;
  name: string;
  state: TodoState;
}

enum TodoState {
  Active = 1,
  Complete = 2
}

class TodoService {

  private static _lastId = 0;

  private static generateTodoId(): number {
    return TodoService._lastId += 1;
  }

  private static clone(src) {
    var clone = JSON.stringify(src);
    return JSON.parse(clone);
  };

  private todos = [];

  constructor(todos: string[]) {
    var _this = this;

    if(todos) {
      todos.forEach(function(todo) { _this.add(todo) });
    }
  }

  add(input) {

    var todo = {
```

```
        id: TodoService.generateTodoId(),
        name: null,
        state: 1
    };

    if(typeof input === 'string') {
        todo.name = input;
    }
    else if(typeof input.name === 'string') {
        todo.name = input.name;
    } else {
        throw 'Invalid Todo name!';
    }

    this.todos.push(todo);

    return todo;
};

clearCompleted() {

    this.todos = this.todos.filter(
        function(x) { x.state == 1; }
    );
}

getAll() {
    return TodoService.clone(this.todos);
};

getById(todoId) {
    var todo = this._find(todoId);
    return TodoService.clone(todo);
};

toggle(todoId): void {

    var todo = this._find(todoId);
```

```
        if(!todo) return;

        switch(todo.state) {
            case 1:
                todo.state = 2;
                break;

            case 2:
                todo.state = 1;
                break;
        }
    }

    private _find(todoId) {
        var filtered = this.todos.filter(
            function(x) { x.id == todoId; }
        );

        if (filtered.length) {
            return filtered[0];
        }

        return null;
    }
}
```

I'll even replace the magic number that I'm using to indicate the `TodoState` with the custom `TodoState` enum that I just defined.

TodoService.ts

```
82 switch(todo.state) {
83     case 1:
84         todo.state = 2;
85     case TodoState.Active:
86         todo.state = TodoState.Complete;
87         break;
88
89     case 2:
90         todo.state = 1;
91     case TodoState.Complete:
92         todo.state = TodoState.Active;
```

```
93     break;
94 }
```

Maybe I'll add a few overloads to the add method...

TodoService.ts

```
35 add(todo: Todo): Todo
36 add(todo: string): Todo
37 add(input): Todo {
```

Then, I'll go back and clean up those anonymous functions by turning them into arrow functions... which also lets me get rid of those annoying temporary “_this” variables.

TodoService.ts

```
27 constructor(todos: string[]) {
28     var _this = this;
29
30     if(todos) {
31         todos.forEach(function(todo) { _this.add(todo) });
32         todos.forEach( todo => this.add(todo) );
33     }
34 }
```

TodoService.ts

```
60     this.todos = this.todos.filter(
61         function(x) { x.state == 1; }
62         x => x.state == 1
63     );
```

TodoService.ts

```
93 private _find(todoId) {
94     var filtered = this.todos.filter(
95         function(x) { x.id == todoId; }
96         x => x.id == todoId
97     );
```

And, while I'm at it, I'll turn the clone method into a generic function:

TodoService.ts

```
20 private static clone(src) {  
21 private static clone<T>(src: T): T {  
22     var clone = JSON.stringify(src);  
23     return JSON.parse(clone);  
24 };
```

And with those changes in place, all of the TypeScript errors have gone away which means I've successfully converted this JavaScript file into a TypeScript file.

Now let's head on to the `TodoApp` and do the same thing, taking advantage of the fact that I've already converted the `TodoService` class.

I'll start by renaming the file from `TodoApp.js` to `TodoApp.ts`.

```
mv TodoApp.js TodoApp.ts
```

Then, convert it to a class...

TodoApp.ts

```
1 class TodoApp {  
2  
3     private todoService: TodoService;  
4     private todoList: TodoListComponent;  
5  
6     constructor(el, todos) {  
7  
8         this.todoService = new TodoService(todos);  
9         this.initialize(el);  
10    }  
11  
12    addTodo(todoName) {  
13        this.todoService.add(todoName);  
14        this.renderTodos();  
15    }  
16  
17    clearCompleted() {  
18        this.todoService.clearCompleted();  
19        this.renderTodos();  
20    }  
21
```



```
22 toggleTodoState(todoId) {
23     this.todoService.toggle(todoId);
24     this.renderTodos();
25 }
26
27 renderTodos() {
28     var todos = this.todoService.getAll();
29     this.todoList.render(todos);
30 }
31
32 initialize(el) {
33
34     var _this = this;
35
36     var addTodoFormEl = el.getElementsByClassName('add-todo')[0],
37         addTodoNameEl = addTodoFormEl.getElementsByTagName('input')[0],
38         todoListEl = el.getElementsByClassName('todo-list')[0],
39         clearCompletedEl = el.getElementsByClassName('clear-completed')[0];
40
41     addTodoFormEl.addEventListener('submit', function(evt) {
42         _this.addTodo(addTodoNameEl.value)
43         addTodoNameEl.value = '';
44         evt.preventDefault();
45     });
46
47     todoListEl.addEventListener('todo-toggle', function(evt) {
48         var todoId = evt.detail.todoId;
49         _this.todoService.toggle(todoId);
50         _this.renderTodos();
51     });
52
53     clearCompletedEl.addEventListener('click', function() {
54         _this.clearCompleted();
55     });
56
57     this.todoList = new TodoListComponent(todoListEl);
58
59     this.renderTodos();
60 }
61
62 }
```

Nothing special here... Except that, when I define properties on my classes, I like to specify their types with interfaces, rather than depend on implementations directly. So, let's hop back to the `TodoService`, create and implement the `ITodoService` interface for it...

TodoService.ts

```
12 interface ITodoService {
13     add(todo: Todo): Todo;
14     add(todo: string): Todo;
15     clearCompleted(): void;
16     getAll(): Todo[];
17     getById(todoId: number): Todo;
18     toggle(todoId: number): void;
19 }
20
21 class TodoService {
22 class TodoService implements ITodoService {
```

Then jump back and use the `ITodoService` interface for the property type as opposed to the `TodoService` class itself.

TodoApp.ts

```
4     private todoService: ITodoService;
```

After that the only error I have left is that TypeScript doesn't know what the `TodoListComponent` class is, so let's fix that by converting `TodoListComponent.js` into TypeScript, too.

```
mv TodoListComponent.js TodoListComponent.ts
```

TodoListComponent.ts

```
1 class TodoListComponent {
2
3     private $el: JQueryStatic;
4
5     constructor(el: HTMLElement) {
6         this.$el = $(el);
7     }
8
9     render(todos: Todo[]) {
```

```

11     this.$el.html('');
12
13     if (!todos.length) {
14         this.$el.html(
15             "<div class='list-group-item text-center text-giant'" +
16             "    <strong>You've completed everything you needed to do!</strong>" +
17             "</div>"
18         );
19
20         return;
21     }
22
23     for(let todo of todos) {
24         this.renderTodo(todo).appendTo(this.$el);
25     }
26 };
27
28 renderTodo(todo: Todo) {
29     return $(
30         "<div class='todo-item list-group-item '" + (todo.state == 2 ? 'completed' : \
31         '' ) + "'>" +
32         "    <div class='row'" +
33         "        <div class='col-md-2 text-center'" +
34         "            <i class='incomplete glyphicon glyphicon-unchecked text-muted text-\
35         -giant'" +
36         "                <i class='completed-indicator completed glyphicon glyphicon-ok tex\
37         t-giant'" +
38         "            </div>" +
39         "        <div class='col-md-10'" +
40         "            <span class='incomplete text-giant'" +
41         "            <span class='completed text-strikethrough text-muted text-giant'" +
42         + todo.name + "</span>" +
43         "        </div>" +
44         "    </div>" +
45         "    <div class='clearfix'" +
46         "    </div>"
47     ).on('click', function() {
48         let event = document.createEvent('CustomEvent');
49         event.initCustomEvent('todo-toggle', true, true, { todoId: todo.id });
50         this.dispatchEvent(event);
51     });
52 }

```

```
53  
54 }
```

Now that I've converted the `TodoListComponent`, everything works great and there are only two TypeScript errors left, and they are both about the JQuery APIs that I'm referencing:

```
TodoListComponent.ts(3,16): error TS2304: Cannot find name 'JQueryStatic'.  
TodoListComponent.ts(6,16): error TS2304: Cannot find name '$'.  
TodoListComponent.ts(29,12): error TS2304: Cannot find name '$'.
```

Regardless, TypeScript generates the JavaScript code for me anyway, which means that I can now run my fully converted TypeScript application.

```
lite-server
```

That's nice that TypeScript compiles my application for me even though I have these two errors, but wouldn't it be nicer if we could actually get rid of those errors, too, by telling TypeScript about the `jQuery` library?

Generating Declaration Files

As I demonstrated in the previous section, one of the first major issues that you're going to face when using TypeScript in a real-world application is that not all libraries are written in TypeScript. Think about it: people have been creating JavaScript libraries for over 15 years now and TypeScript is only a few years old, so of course that's the case! So, in this section, I'm going to show you what you can do when you want to consume a JavaScript library that doesn't have TypeScript type information.

When I left off in the previous section, I had converted all of the JavaScript files in my sample application to TypeScript and ended up with only a couple of errors: the places that I was referring to `jQuery` and the `jQuery $` function to perform DOM manipulation.

The easiest way for me to make these errors go away is to simply tell TypeScript that the `$` function is a third-party library and not to worry - I know what I'm doing when I use it, so just consider it the "any" type.

Your first instinct might be to do something like this:

```
var $: any;
```

And, I guess that might work, but it will also be generated out as JavaScript code and could possibly overwrite the real `jQuery $` variable. No, what we need is a way to tell TypeScript about a variable without actually creating a reference to that variable, and in order to do that, we can use the `declare` keyword, like this:

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

If you open up the generated code you can see that this line of code is not included - it only exists for TypeScript's benefit at design time.

But that approach only makes the errors go away - it doesn't actually give us strong typing for the jQuery library. Fortunately, there's another solution, and that is the combination of TypeScript declaration files and TypeScript Typings library.

A TypeScript declaration file is simply a file that describes a library that's not written in TypeScript, or perhaps a library that was originally written in TypeScript but then got compiled into JavaScript for distribution. In these cases, the declaration file describes the type information about the JavaScript code that it goes along with. If you've ever used the C or C++ languages, declaration files are a lot like those – they don't actually define an implementation themselves, they just describe an implementation that lives in another file.

TypeScript declaration files are made up of their own interesting syntax that you could learn, however you don't have to. If you ask it to, TypeScript will generate declaration files for you!

To tell TypeScript that you want declaration files, simply go to the `tsconfig.json` file and set the declaration compiler option to true:

`tsdconfig.json`

```
{
  "compilerOptions": {
    "target": "es5",
    "declaration": true
  }
}
```

Then, save and run the TypeScript compiler.

When it's all done compiling you'll see that not only has TypeScript produced the compiled JavaScript file, but also the corresponding declaration file – a file with the same name as the TypeScript file, only with the `.d.ts` file extension.

When I open one of these files I see that TypeScript includes the same interfaces that I defined, pretty much untouched:

'TodoService.d.ts'

▲ TODOAPP

index.html
 TodoApp.js
 TodoApp.ts
 TodoApp.d.ts
 TodoListComponent.js
 TodoListComponent.ts
 TodoListComponent.d.ts
 TodoService.js
 TodoService.ts
 TodoService.d.ts
 tsconfig.json

Generated type definitions

```
interface Todo {
  id: number;
  name: string;
  state: TodoState;
}

interface IToDoService {
  add(todo: Todo): Todo;
  add(todo: string): Todo;
  clearCompleted(): void;
  getAll(): Todo[];
  getById(todoId: number): Todo;
  toggle(todoId: number): void;
}
```

The only difference is that, for any type that represents actual code – things like enums, variables and class definitions - TypeScript attaches the `declare` keyword to the front to indicate that the type definition just describes an implementation that lives elsewhere.

'TodoService.d.ts'

```
declare enum TodoState {
  Active = 1,
  Complete = 2,
}
```

These kinds of declarations that don't define an implementation are called **ambient declarations**.

Why would you generate a declaration file for your own TypeScript files? Well, in cases like this where you're just using TypeScript to generate JavaScript for a single application, you probably wouldn't. But, if you were creating a utility library or some other bit of code that you wanted to share with another team or application, you could generate the JavaScript - probably bundling and minifying it in the process - and then deliver that minified JavaScript along with the TypeScript definition to the other team or application so they have the

JavaScript to run and the TypeScript to give them the type information. In other words, you've got the best of both worlds.

You may also be wondering how the ability to generate a TypeScript declaration files would help with our jQuery \$ function errors... well, it doesn't. But, if we can get ahold of a declaration file that describes the jQuery library for us that somebody else wrote, well that would really help! Check out the next section where I'll introduce you to a tool that allows you to download TypeScript declarations for pretty much any popular open source library you might need.

Referencing third-party libraries

As I demonstrated in the previous few sections, one of the first major issues that you're going to face when using TypeScript in a real-world application is that not all libraries are written in TypeScript. In the last section, I showed you how TypeScript provides the ability to use declaration files to describe any JavaScript code and, in fact, many JavaScript libraries are beginning to include these declaration files in their distribution packages. However you're bound to come across a library that doesn't include its type information, so in this section I'm going to show you a tool named "TSD" that helps you download TypeScript declarations for pretty much any popular open source library you might want.

Unfortunately, TSD is not installed along with the TypeScript compiler, so you'll need to install it separately. You'll use NPM again, just like for the TypeScript compiler, only this time you're installing the "tsd" package using this command:

```
npm install -g tsd
```

Notice the message during the installation mentioning that the TSD tool is deprecated in favor of a new tool named Typings. As of the time of this writing - March, 2016 - this tool is very new and personally I don't think it's mature enough yet so I'm going to stick with TSD for the time being. But, keep an eye on it as it promises that it will allow you to install all of the type definitions that the TSD tool does, plus more!

Once you've got TSD installed, you can use its "query" command to search for packages. For instance, if I want to find type information for jquery, I can say:

```
tsd query jquery
```

I can see here that TSD found a result for jquery. Since it found something, I know I can safely install that package using the install command and passing it the package name, like this:

```
tsd install jquery
```

This will download all of the type information and add it to my project in a new folder named "typings".

If you open the `jquery.d.ts` file, you can see the ambient declaration of the full jQuery library, complete with comments! Now, if you go back to `TodoListComponent.ts` - and remove that silly `declare` statement that I made before - I can get full autocomplete and IntelliSense information for the jQuery library - all provided by that new type declaration!

And, of course, since Visual Studio Code is not showing me any errors, that means the `tsc` compiler is now happy, too!

Note that using `tsd` in this way just downloads the typings once.

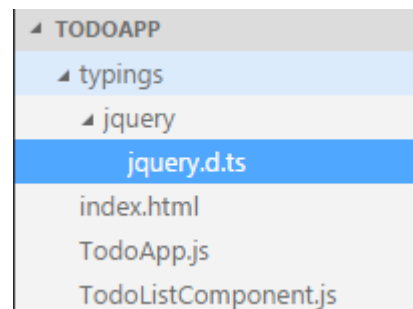
Generally speaking, once you've found the type definition that your project depends on, you're going to want to save a reference to that definition in your source control as opposed to the definition itself. Luckily, `tsd` supports that: just run the `tsd install` command but this time add the `--save` flag to the end.

```
tsd install jquery --save
```

Just like last time, `tsd` says that it installed 1 `.d.ts` file, however this time it also created a special file named `tsd.json` that contains a reference to the type definition file that it installed so it's easier to get later.

tsd.json

```
{
  "version": "v4",
  "repo": "borisyankov/DefinitelyTyped",
  "ref": "master",
  "path": "typings",
  "bundle": "typings/tsd.d.ts",
  "installed": {
    "jquery/jquery.d.ts": {
      "commit": "dade4414712ce84e3c63393f1aae407e9e7e6af7"
    }
  }
}
```



Typings folder

With this file in place, I can exclude the `typings` folder from source control and every time I get the project I can run the `tsd install` command with no parameters to restore all of the type definitions that I've saved into the `tsd.config` file:


```
rm -rf typings
tsd install
```

Where are all of these typings coming from you might be wondering? There is a community-supported hosted on GitHub named DefinitelyTyped ([github.com/DefinitelyTyped](https://github.com/DefinitelyTyped/DefinitelyTyped)¹).

If you navigate into the repository, you can see that it has hundreds of type definitions – so many in fact that GitHub has to truncate the list after 1,000 entries! Pretty impressive.

So, the next time you use your favorite open source library and it doesn't include TypeScript type information, use TSD or check out the GitHub repository to see if the library is listed and download it from there. I have yet to find a library that isn't in here... and, hey, if it doesn't exist, maybe you could contribute your own definition!

Converting to External Modules

In this chapter, I've walked you step-by-step through converting an existing JavaScript application to TypeScript, and at this point I'm pretty much done – I could stop here. I could stop, but instead let's go the extra step to get all of our code out of the global namespace and into external modules. I'm going to move quickly through this section because I already covered most of what I'm about to show you in the modules chapter. However, there are a few little tips and tricks that I still wanted to share, so let's get on with it.

Before I get started, I just want to point out that I moved the `Todo` and `TodoState` enum types into their own `Model.ts` file in between the last section and this one, just like I said I would. This shouldn't make any difference, I just didn't want to catch you by surprise.

First, I'll start by setting the `module` compiler option in `tsconfig.json` to "system" because I'm going to use the `system.js` module loader again.

tsconfig.json

```
{
  "compilerOptions": {
    "module": "system"
  }
}
```

Then, I'll go through each file, exporting its types and importing types from the other modules it depends on.

I'll start with the `TodoService`, importing the `Todo` and `TodoState` from the `Model` module:

¹<https://github.com/DefinitelyTyped/DefinitelyTyped>

```
import { Todo, TodoState } from './Model';
```

Then, I'll export the `ITodoService` interface and `TodoService` class...

```
export ITodoService {
```

```
// ...
```

```
export TodoService implements ITodoService {
```

And, since I'm in here, I'll move those static variables out of the class and into the module scope so that they can be truly private.

```
import { Todo, TodoState } from './Model';
```

```
export interface ITodoService {
  add(todo: Todo): Todo;
  add(todo: string): Todo;
  clearCompleted(): void;
  getAll(): Todo[];
  getById(todoId: number): Todo;
  toggle(todoId: number): void;
}
```

```
let _lastId = 0;
```

```
function generateTodoId(): number {
  return _lastId += 1;
}
```

```
function clone<T>(src: T): T {
  var clone = JSON.stringify(src);
  return JSON.parse(clone);
};
```

```
export default class TodoService implements ITodoService {
```

```
  private static _lastId = 0;
```

```
  private static generateTodoId(): number {  

    return TodoService._lastId += 1;  

}
```

```

private static clone<T>(src: T): T {
  var clone = JSON.stringify(src);
  return JSON.parse(clone);
};

private todos = [];

```

Of course, when I do this, I need to fix all the references...

```

var todo = {
  id: TodoService.generateTodoId(),
  id: generateTodoId(),
  name: null,
  state: 1
};

// ...

getAll() {
  return TodoService.clone(this.todos);
  return clone(this.todos);
};

// ...

getById(todoId) {
  var todo = this._find(todoId);
  return TodoService.clone(todo);
  return clone(todo);
};

```

Next, I'll move on to the `TodoListComponent`.

First, I'll import the `Todo` interface from the `Model`...

```
import { Todo } from './Model';
```

Then I'll export the class...

```
export class TodoListComponent
```

Now, this module is interesting because it depends on the jQuery library which right now I'm loading from a CDN!

But, the module loader doesn't care if I put a full URL in there, so I'll just copy this URL and directly reference it as an import.

```
import '//code.jquery.com/jquery-1.12.1.min.js';
```

And, since I'm just loading it just so that it is available to this module, I'm not actually importing any types from it. I'm just adding this import statement to let the module loader know that this module expects the library to be loaded and available before the module itself gets loaded.

Also, It's important to keep in mind that TypeScript knows about the jQuery types from the type declaration I installed earlier, not because I'm importing it as a module import!

And then there's the TodoApp component... Just like the others, I'll import my dependencies and export the TodoApp class.

TodoApp.ts

```
import { Todo, TodoState } from './Model';  
import { TodoService, ITodoService } from './TodoService';  
import { TodoListComponent } from './TodoListComponent';
```

Notice that I'm not importing the jQuery library – that is not my dependency! jQuery will get loaded at runtime because the TodoApp module is loaded and the TodoApp module depends on the TodoListComponent, which depends on the jQuery library. Right after encapsulation and keeping things out of the global scope, it's this separation of concerns is the true power of the external module approach.

Also, here's another option that you have when exporting members from a module - you can choose one member to be the “default” member of the module, which changes both the export and the import syntax ever so slightly...

I'll start by exporting the TodoService class as the default export of this module by simply adding the keyword “default” after the export keyword:

TodoService.ts

```
export default class TodoService implements ITodoService {
```

Note, even though I can only have one default export, I can still export other members normally like I'm doing with the ITodoService interface.

Then, I can switch back to the TodoApp and import the new default export by simply removing the brackets around it, like this:

TodoService.ts

```
import TodoService from './TodoService';
```

When I do this, TypeScript now warns me that it doesn't know about `ITodoService` anymore, so I have to add it back.

Luckily, you can import both default exports and regular exports all in one line by adding a comma between the default import and the import brackets like this:

```
import TodoService, { ITodoService } from './TodoService';
```

Once that's done, I'll head back to the `index.html` file and clean that up by removing all of the script tags that refer to my individual JavaScript files and replace them with a single script reference to `System.js`.

index.html

```
<script type="text/javascript" src="//cdnjs.cloudflare.com/ajax/libs/systemjs/0.\n19.22/system.js"></script>
```

And, like last time, I'll add that `defaultJSExtensions = true` setting:

index.html

```
<script type="text/javascript">\n    System.defaultJSExtensions = true;\n</script>
```

Notice that this demo is a little bit different than the one from the modules chapter in that I'm not actually initializing the application in the `TodoApp` module - that module just provides the application class and I need to wire it up. Luckily, the `System.import()` function returns the module to me in a promise, so I can dig into that promise to pull out my class and create a new instance of it, like this:

index.html

```
System.import('TodoApp').then(function(module) {  
    new module.TodoApp(document, [  
        'Pick up drycleaning',  
        'Clean Batcave',  
        'Save Gotham'  
    ]);  
})
```

And that's it! Now I can switch back to the command line, run the TypeScript compiler...

tsc

And then launch the site to prove that it all works!

lite-server

And there you go - a fully-working TypeScript application, starting with an existing JavaScript application and converting to TypeScript. Even though most of this section was rehashing features that I've already demonstrated in previous sections, hopefully seeing it in action with slightly more real-world examples helps bring it home a little bit more.

Now that I've shown you how to convert an existing JavaScript application, head on to the next section where I'll show you how to debug it!

Debugging TypeScript with source maps

In this chapter I took all of the TypeScript features that I've shown throughout this book and used them to convert an existing JavaScript application into a TypeScript application. But, what I didn't show it's the realest of the real-world scenarios and that is using a debugger to find bugs! In this section, I'll introduce you to the concept of source maps and show you how they enable you to debug your TypeScript code right inside the browser!

I'll start by strategically introducing a bug into my code... because of course the only way bugs can get into your code is when you deliberately put them in there... right?

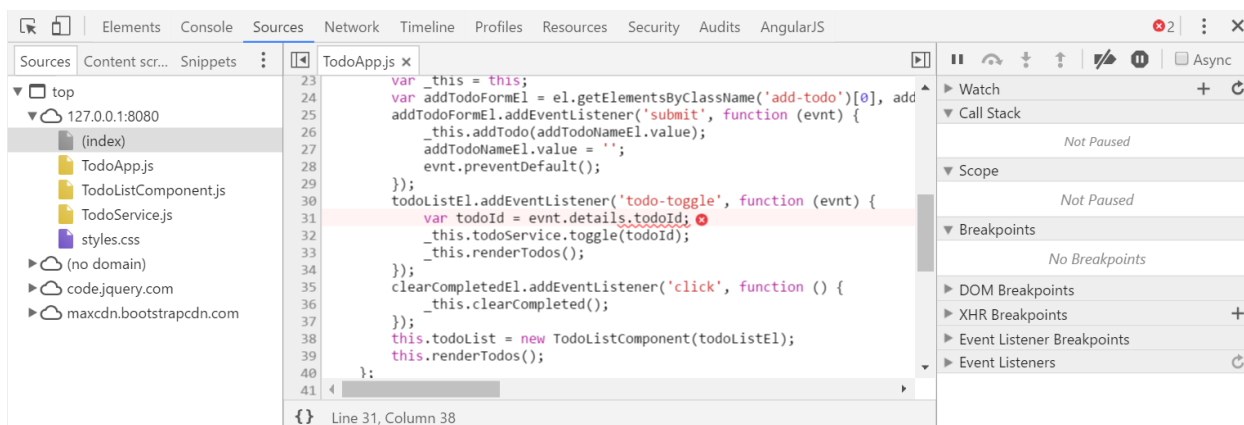
TodoApp.ts

```
51      var todoId = evt.detail.todoId;
52      var todoId = evt.details.todoId;
```

Then, I'll run the site and enable “break on exceptions”, then watch it crash when I click a Todo to toggle its state...

```
tsc
lite-server
```

Notice how Chrome brings me to the line of JavaScript code that my error is on, which is line 31 in the generated JavaScript file:

**Poor debugging experience**

But, the bug is actually on line 51 of my TypeScript code! Luckily in this case it's pretty easy to figure out where this was in my TypeScript file, but other times it's not so straight-forward. Plus, I'm not working in JavaScript - I'm working in TypeScript, so why do I care about which JavaScript line has an error?

The answer to all of this is an emerging browser feature called a “Source map”. The source map feature allows language compilers to tell browsers where, exactly, a particular variable or expression lives in the original source code, regardless of what that source code is. Then, when I open up the page in the debugger, the browser will load that source file, read in that metadata, and if any errors occur, the browser can take me to the line in the SOURCE code where the error occurred, not the JavaScript code. In other words, I could create my own brand new language named “JessScript” and as long as I provide the correct source map information for the JavaScript that “JessScript” emits, I can have full, native debugging support for my new language right there in the browser.

What's this got to do with TypeScript?

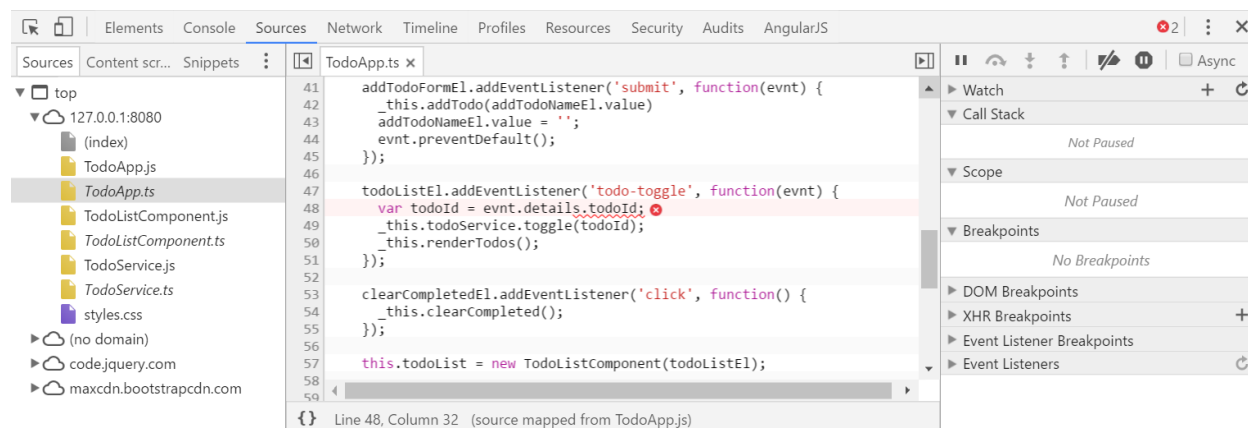
Well, try setting the `sourceMap` compiler option to `true` and run the TypeScript compiler again.

tsconfig.json

```
{
  "compilerOptions": {
    //...
    "sourceMap": true
  }
}
```

With that compiler option in place, TypeScript is now generating a new type of file for each TypeScript file - a `.js.map` file. If I open up one of these files, I can see all of the metadata that TypeScript has generated – this is the source mapping information that tells the browser which JavaScript expressions map to which lines of TypeScript code!

So now when I reload the browser and run into that exception again, the browser takes me to the offending line of code in my TypeScript file as opposed to the generated JavaScript output!

**Debugging source maps**

And, I've still got full debugging support - I can hover over variables, use the console to execute expressions, etc. Everything works just like I'd expect it to.

But, how'd the browser know which source map to load? Well, if I jump back to the generated JavaScript and scroll alllll the way to the end, I see this comment at the bottom:

```
//# sourceMappingURL=TodoService.js.map
```

But, this isn't an ordinary comment - it's actually metadata telling the browser where it can find the source map file. As soon as the browser sees this, it downloads the file - and any of the source files that the source map refers to - and loads them up in the debugger, ready for me to step through them.

Perhaps the most interesting thing about this feature is that it's not a TypeScript feature at all. I mean, TypeScript knows how to generate the source mapping metadata, but it's the browsers that do all of the hard work of downloading and parsing the source maps, then mapping all of the JavaScript code to my custom code. I don't know about you, but I find this feature pretty amazing, and it's proven very helpful to me several times. I'm sure it will help save you at some point, too!

11. Decorators

At this point in the book, I've shown you all the features that you'll need in order to leverage TypeScript in your day-to-day JavaScript development. I also walked you through how to convert an existing JavaScript application into TypeScript as well as how to debug TypeScript applications in the browser. Even though I've explained enough for you to be very productive with TypeScript, I've still got one last feature to show you in this chapter and that is **decorators**.

Decorators are a *proposed* ECMAScript syntax that allow you to implement the Decorator design pattern to modify the behavior of a class, method, property, or parameter in a declarative fashion. This powerful approach allows you to define common behavior in a central place and easily apply it across your application to reduce duplicate code and make your code more readable and maintainable all at the same time.

For instance, let's say I want to log every single time a method is called and after it's done executing. If I were to do this with normal, imperative, inline code it'd look something like this:

```
add(input): Todo {
    console.log(`add(${JSON.stringify(input)})`);
    // Add Todo
    console.log(`add(${JSON.stringify(input)}) => ${JSON.stringify(todo)}`);

    return todo;
};
```

You don't need TypeScript to apply the Decorator design pattern to this problem. If you really wanted to add the logging logic without changing the code inside of the add method, you could wrap the original method in another method.

For example, you can start like this:

```
var originalMethod = TodoService.prototype.add;

TodoService.prototype.add = function (...args) {
    let returnValue = originalMethod.apply(this, args);
    return returnValue;
}
```

This is your basic method decorator pattern - it replaces the original method with another method that duplicates the original method's behavior by actually calling that method and returning its result.

Once I have this basic structure in place, I can now start adding logic before and after the original method is called, like this:

```
1  var originalMethod = TodoService.prototype.add;
2
3  TodoService.prototype.add = function (...args) {
4
5      console.log(`add(${JSON.stringify(args)})`)
6
7      let returnValue = originalMethod.apply(this, args);
8
9      console.log(`add(${JSON.stringify(args)}) => ${JSON.stringify(returnValue)}`)
10
11     return returnValue;
12 }
```

Here I've added two logging statements, one on line 5 right at the entry into the call and the other on line 9, after the original method has been called but before I return the value that the original method returned. This approach even allows me to include the return value in my log statements.

The net result of all of this is a method that wraps another method, allowing me to attach behavior to the method while keeping the logic defined in the original method completely unchanged. Used appropriately, this approach can be incredibly powerful.

Ok, so that's an example of the decorator pattern. Now let's see ECMAScript Decorators in action.

In order to apply a decorator to a method, you simply apply an @ followed by the decorator name in front of the member that you want to decorate, like this:

```
@log
add(input): Todo {
```

Here I've told ECMAScript to attach the behavior defined in a method decorator named `log` to the `add` method – a method decorator that is going to have the same exact logic that I just showed a minute ago.

What's a decorator? Well, it's just a function with a special signature. As I mentioned earlier, ECMAScript Decorators support four different targets - classes, methods, properties, and parameters - and their signatures look like this:

```
declare type ClassDecorator = <TFunction extends Function>(  
    target: TFunction  
    ) => TFunction | void;  
  
declare type PropertyDecorator = (  
    target: Object,  
    propertyKey: string | symbol  
    ) => void;  
  
declare type MethodDecorator = <T>(  
    target: Object,  
    propertyKey: string | symbol,  
    descriptor: TypedPropertyDescriptor<T>  
    ) => TypedPropertyDescriptor<T> | void;  
  
declare type ParameterDecorator = (  
    target: Object,  
    propertyKey: string | symbol,  
    parameterIndex: number  
    ) => void;
```

By the way, if you're wondering how I got to this definition, it's one of the core definitions that ships with TypeScript so I just created a variable with one of these names

```
var x: ClassDecorator
```

Then, selected the type and said “Go to Definition”, which brought me here.

I'll show you how to create and use each of these decorators in this chapter, but in this section, I'll start by showing you how to implement a method decorator and which looks like this:

```
function log(  
    target: Object, methodName: string,  
    descriptor: TypedPropertyDescriptor<Function>  
)
```

- The `target` parameter is the object that the member lives on – in this example, it'll be an instance of a `TodoService`.
- The `methodName` parameter is the name of the method to be decorated
- And, finally, the `descriptor` is an object that contains all of the metadata for the method that you're looking to modify.

The descriptor object has a handful of properties that describe the member being decorated, but in this case the one I'm really looking for is the `value` property which is the method itself.

Remember that Decorator pattern example I just showed you? Well, `descriptor.value` is really just the value of the `originalMethod` method in my previous example!

That means that I can just copy that code into this log decorator function and modify it a bit, changing the references from `TodoService.add` to `descriptor.value` instead...

```
function log(
  target: Object, methodName: string,
  descriptor: TypedPropertyDescriptor<Function>
) {

  var originalMethod = descriptor.value;

  descriptor.value = function (...args) {
    console.log(`TodoService.add(${JSON.stringify(args)})`)

    let returnValue = originalMethod.apply(this, args);

    console.log(`TodoService.add(${JSON.stringify(args)}) => ${JSON.stringify\
y(returnValue)})`)

    return returnValue;
  }

}
```

Of course, I'll need to change the hard-coded method name to make this decorator say the name of the method that's being wrapped, but that's pretty easy:

```
`${methodName}(${JSON.stringify(args)})`
```

And that's how you write your own method decorator!

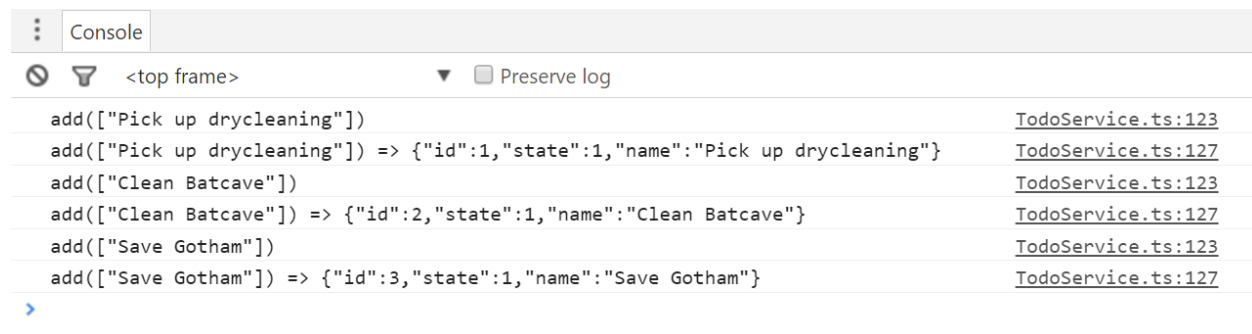
However, there is one thing stopping me from running this code right now... If I hover over the decorator, I can see that TypeScript is giving me an error. This is because, while TypeScript does allow you to create and consume decorators, they are still just an ECMAScript proposal at the time of this writing, so the compiler takes a very conservative approach and will refuse to allow the use of decorators unless you opt-in to the functionality by setting a compiler flag.

That means that in order to get this to compile properly, I'll need to open up the `tsconfig.json` file and set the "experimentalDecorators" flag to true.

tsconfig.json

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

With that setting in place, the TypeScript error goes away, I can compile my code, and run my application to see my new decorator in action.



```
add(["Pick up drycleaning"])
add(["Pick up drycleaning"]) => {"id":1,"state":1,"name":"Pick up drycleaning"}
add(["Clean Batcave"])
add(["Clean Batcave"]) => {"id":2,"state":1,"name":"Clean Batcave"}
add(["Save Gotham"])
add(["Save Gotham"]) => {"id":3,"state":1,"name":"Save Gotham"}
```

Log decorator in action

Pretty cool, huh? Well, if you think that's cool, what if I told you that in addition to being able to decorate methods, you can also define decorators for properties, accessors, classes, and even method parameters! What's more, they're all just variations of the method decorator function that I implemented in this section. So, if you're interested in seeing how to create these other types of decorators and use them to easily and effectively add dynamic behavior throughout your application, please check out the rest of this chapter!

Implementing Class Decorators

In the previous section, I introduced you to the concept of decorators and even showed you how to use them in order to quickly and easily add logic that executed before and after a method – without modifying the code in the method itself! In the rest of this chapter, I'm going to show you how to create and apply class and property decorators to implement a mini validation framework to enable runtime-checking of property and parameter values. And in this section I'll start with the easiest one: the class decorator.

I'll start by creating a new TypeScript file to hold everything and import my model types.

Validators.ts

```
import {Todo, TodoState} from './Model';
```

Then, I'll create a class that implements the Todo interface so that I have something to decorate.

Validators.ts

```
export class ValidatableTodo implements Todo {  
  
    id: number;  
    name: string;  
    state: TodoState;  
  
}
```

Just as method decorators are used to decorate a method, class decorators decorate a class – in other words, they can be used to modify the class's constructor function and its prototype members after the class has been defined.

One good example of using class decorators is to dynamically attach methods to a class definition. For example, let's say that I want to define one single `validate` method and be able to attach it to any class in my application.

Let's see what it'd look like if I did this manually, without decorators.

First, I'd create the `validate` function itself:

```
1  export function validate() {  
2  
3      let validators = [].concat(this._validators),  
4          errors = [];  
5  
6      for(let validator of validators) {  
7          let result = validator(this);  
8  
9          if(!result.isValid) {  
10             errors.push(result);  
11         }  
12     }  
13  
14     return errors;  
15 }
```

This function just tries to find an array of validators on the current instance and then iterates through them, calling each one and looking at its result. If the result is not valid, I add it to the list of errors and then return all the errors at the end. Notice how I used the `array.concat` method on line 3 to get the array of validators. The `array.concat` method accepts null or undefined values which means that if no validators have been registered, this array will simply be empty and I won't have to do any null checks.

And, actually, since this is a TypeScript book, let me add some types in here:

```
export interface IValidatable {
    validate(): IValidationResult[];
}

export interface IValidationResult {
    isValid: boolean;
    message: string;
    property?: string;
}

export interface IValidator {
    (instance: Object): IValidationResult;
}

export function validate() {
    let validators: IValidator[] = [].concat(this._validators),
        errors: IValidationResult[] = [];

    for(let validator of validators) {
        let result = validator(this);

        if(!result.isValid) {
            errors.push(result);
        }
    }

    return errors;
}
```

Now that I have my `validate` function, let's go ahead and add it to a class by assigning it to the class's prototype.


```
ValidatableTodo.prototype._validators = [];  
ValidatableTodo.prototype.validate = validate;
```

Ok, if that's how you'd attach behavior using the Decorator pattern *without* the decorator syntax, let's see how I can use a decorator 'to accomplish this same thing in a little bit cleaner way...

Just like the method decorator function, I'll start with a function that just has one parameter: the class's constructor function.

```
export function validatable(target: Function) {  
}
```

And inside this function I can apply the same exact logic I just applied to the `ValidatableTodo` class, but in a more generic way:

```
export function validatable(target: Function) {  
    target.prototype.validate = validate;  
  
}
```

Now, I can apply this decorator to any class using the same exact syntax used to decorate a method:

```
@validatable  
export class ValidatableTodo implements Todo {  
  
    id: number;  
    name: string;  
    state: TodoState;  
  
}
```

And, since the `validatable` decorator adds the `validate` method dynamically, I can let TypeScript know that this class now implements the `IValidatable` interface by adding a merged declaration that says exactly that:

```
export interface ValidatableTodo extends IValidatable {  
}
```

Then, to try it out, I can refactor the `add` method on the `TodoService` to use this new class to validate new `Todos` as they are created:

TodoService.ts

```
import { ValidatableTodo } from './Validators';

let todo = new ValidatableTodo();
todo.id = generateTodoId();
todo.state = TodoState.Active;

if (typeof input === 'string') {
  todo.name = input;
}
else if (typeof input.name === 'string') {
  todo.name = input.name;
} else {
  throw 'Invalid Todo name!';
}

let errors = todo.validate();

if(errors.length) {
  let combinedErrors = errors.map(x => `${x.property}: ${x.message}`)
  throw `Invalid Todo! ${errors.join('; ')}`;
}
```

With this new logic in place, all of the Todo objects will be validated with the various rules that I'll apply to the ValidatableTodo class in the rest of this chapter.

Speaking of which, let's move on to the next section where I'll show you how to use property decorators to add this validation logic!

Implementing Property Decorators

In the previous few sections, I showed you how to create decorator functions that you could use to decorate methods and classes. In this section, I'm going to show you how to implement property decorators that will allow you to dynamically add behavior to your properties.

The signature of a property decorator is right in between a class decorator and a method decorator:

```
function required(target: Object, propertyName: string) {
```

It takes two parameters: the target object and the name of the property on that object.

Then, within this function I can define the logic that I want to execute any time the property decorator is applied to a property.

In this instance, what I want to do is add a validation function to the list of validators on the target object. So, first, I'll need to get a reference to that list of validators like this:

```
1 function required(target: Object, propertyName: string) {
2
3     let validatable = <{ _validators: IValidator[] }>target,
4       validators = (validatable._validators || (validatable._validators = []));
5
6 }
```

Notice how I'm using an anonymous type on line 3 so that I can still get the benefits of typing just in this method even though this isn't an formally-defined type.

Then, I can define and add a validation function to this list.

```
function required(target: Object, propertyName: string) {

    let validatable = <{ _validators: IValidator[] }>target,
        validators = (validatable._validators || (validatable._validators = []))\
);

    validators.push(function (instance) {
    });
}
```

I'll start by getting the value of the property for this particular instance:

```
function required(target: Object, propertyName: string) {

    let validatable = <{ _validators: IValidator[] }>target,
        validators = (validatable._validators || (validatable._validators = []))\
);

    validators.push(function (instance) {
        let propertyValue = instance[propertyName];
    });
}
```

Then, since this decorator is enforcing the fact that the property must have a value, I'll check that property value to make sure it's assigned to something, and - if it's a string - that it contains at least one character.

```

validators.push(function (instance) {
  let propertyValue = instance[propertyName],
      isValid = propertyValue !== undefined;

  if (typeof propertyValue === 'string') {
    isValid = propertyValue && propertyValue.length > 0;
  }
});

```

And, finally, I'll return an object literal that meets the contract defined by the `IValidationResult` interface:

```

function required(target: Object, propertyName: string) {

  let validatable = <{ _validators: IValidator[] }>target,
      validators = (validatable._validators || (validatable._validators = []));

  validators.push(function (instance) {
    let propertyValue = instance[propertyName],
        isValid = propertyValue !== undefined;

    if (typeof propertyValue === 'string') {
      isValid = propertyValue && propertyValue.length > 0;
    }

    return {
      isValid,
      message: `${propertyName} is required`,
      property: propertyName
    };
  });
}

```

Now that I've defined the property decorator, I can go ahead and add it to my property, using the same syntax that I used for the method and class decorators in the previous sections: the `@` character, followed by the decorator function name:

```
@validatable
export class ValidatableTodo implements Todo {

    id: number;

    @required
    name: string;

    state: TodoState;

}
```

With all that in place, I can run the site and try to add an empty todo item to see the new validation in action.

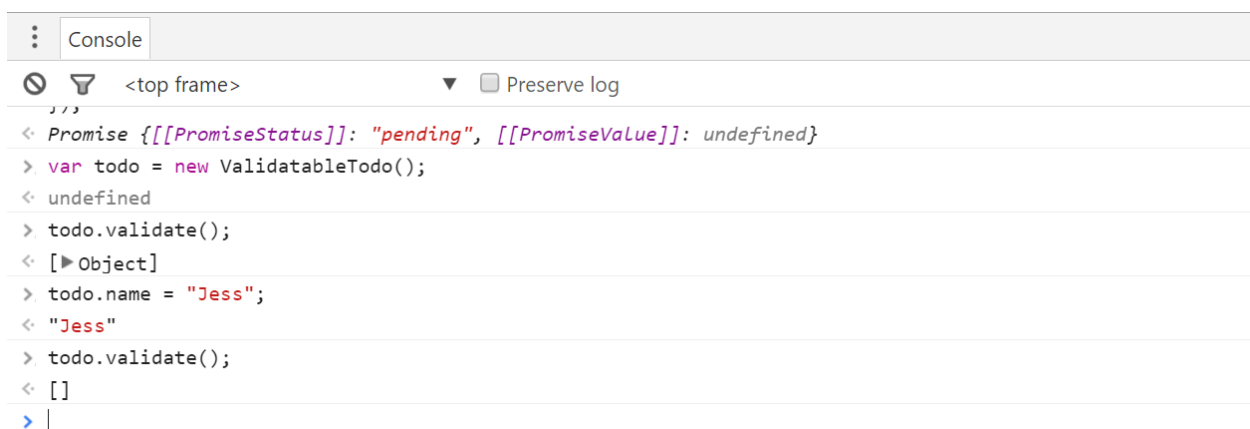


The screenshot shows a web browser's developer console with the 'Console' tab selected. The log shows the following sequence of events:

- A script execution starting with `System.import('Validators').then(function(module) { window.ValidatableTodo = module.ValidatableTodo });`.
- A `Promise` object being logged: `Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}`.
- A variable `todo` being assigned: `var todo = new ValidatableTodo();`.
- The `todo` variable being logged as `undefined`.
- The `todo.validate()` method being called.
- An `Object` being logged, representing the validation error: `{ isValid: false, message: "name is required", property: "name", __proto__: Object }`.

Required validation error

Of course, if I try to add a value that actually has some characters in it, it goes through just fine...



The screenshot shows a web browser's developer console with the 'Console' tab selected. The log shows the following sequence of events:

- A script execution starting with `System.import('Validators').then(function(module) { window.ValidatableTodo = module.ValidatableTodo });`.
- A `Promise` object being logged: `Promise {[[PromiseStatus]]: "pending", [[PromiseValue]]: undefined}`.
- A variable `todo` being assigned: `var todo = new ValidatableTodo();`.
- The `todo` variable being logged as `undefined`.
- The `todo.validate()` method being called.
- An `Object` being logged, representing the validation result: `{}`.
- The `todo.name` property being assigned: `todo.name = "Jess";`.
- The `todo.name` property being logged as `"Jess"`.
- The `todo.validate()` method being called again.
- An `Object` being logged, representing the validation result: `{}`.

No more validation errors

It's probably best to point out that if this were a real application I wouldn't be throwing an error like this – instead, I'd send the error objects up to the view and display them to the user so they have some idea of what properties are invalid and why. However, all of that is just slightly outside of the scope of this book.

But, you know what's *not* outside of the scope of this book? The next section, where I'll show you how to pass parameters into a decorator using a pattern called a decorator factory.

Implementing Decorator Factories

So far in this chapter I have shown you how to create and apply class, method, and property decorators and, until now, all of the decorators I've shown you have been functions whose parameters are the things they are decorating - for example, class constructors, methods, or properties, such as this property decorator:

```
function regex(target: Object, propertyName: string) {  
}
```

But, what if you wanted to pass a parameter to one of those decorator functions? In the case of this decorator, what if I wanted to pass a regex string that I could execute in order to validate the property value?

Well, that wouldn't work because decorators need to have a specific signature and don't leave any room for other parameters. In cases like these you need a **decorator factory**.

The concept of a decorator factory is actually quite simple: it's a function that returns a decorator function. In other words, the factory function itself doesn't have to match the signature of, say, a property decorator - it just needs to return a function with a signature that matches a property decorator... like this:

```
function regex(pattern: string, flags?: string) {  
  
    return function (target: Object, propertyName: string) {  
    }  
  
}
```

And, as with any other inner JavaScript function, the decorator function created inside of the decorator factory has full access to any of the parameters passed into the decorator factory function itself.

In other words, take the `pattern` parameter that's passed into the decorator factory – I'll have access to that inside of the decorator function to validate the value of the property at runtime. So, if I make a copy of the required validator from before...

```

function regex(pattern: string, flags?: string) {

    return function (target: Object, propertyName: string) {

        let validatable = <{ _validators: IValidator[] }>target,
            validators = (validatable._validators || (validatable._validators = \
[]));

        validators.push(function(instance) {
            let propertyValue = instance[propertyName],
                isValid = propertyValue !== undefined;

            return {
                isValid,
                message: `${propertyName} is required`,
                property: propertyName
            };
        });
    }
}

```

I can modify the validation logic from checking for an undefined property value to checking to see if the property value is a string that matches the regular expression:

```

function regex(pattern: string, flags?: string) {

    let expression = new RegExp(pattern, flags);

    return function(target: Object, propertyName: string) {

        let validatable = <{ _validators: IValidator[] }>target,
            validators = (validatable._validators || (validatable._validators = \
[]));

        validators.push(function(instance) {
            let propertyValue = instance[propertyName],
                isValid = expression.match(propertyValue);

            return {
                isValid,
                message: `${propertyName} does not match ${expression}`,
            };
        });
    }
}

```

```

        property: propertyName
    };
});

}

}

```

Then I use this new decorator factory almost exactly as I use a normal decorator function, except that I call it as a function with parentheses and pass in parameters.

For instance, if I wanted to make sure that the `name` property only included alpha-numeric characters or spaces, I could apply the new decorator factory with this regular expression:

```

@validatable
export class ValidatableTodo implements Todo {

    id: number;

    @required
    @regex(`^[a-zA-Z ]*$`)
    name: string;

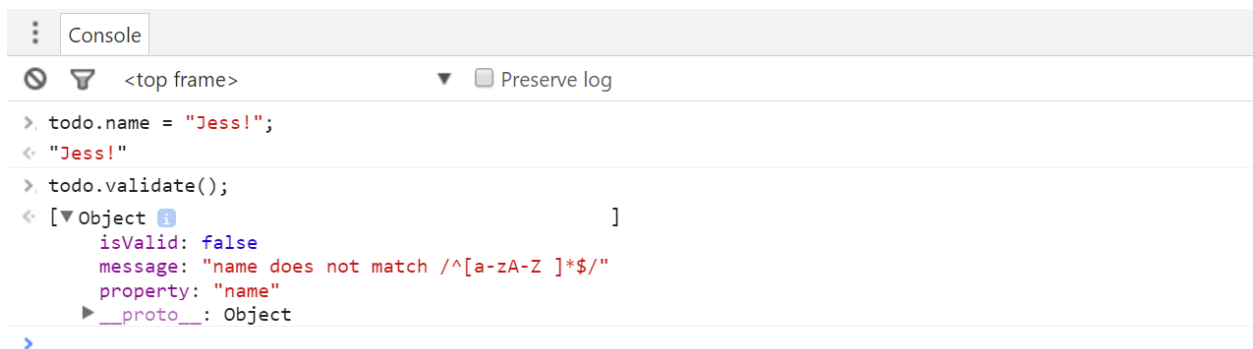
    state: TodoState;

}

```

And, yes - it's totally valid to apply multiple decorators to a single property, class, or method!

With this in place, I can run the site and if I give the `name` property a value - but that value has non-alphanumeric characters - I can see my new regex validator kicking in!



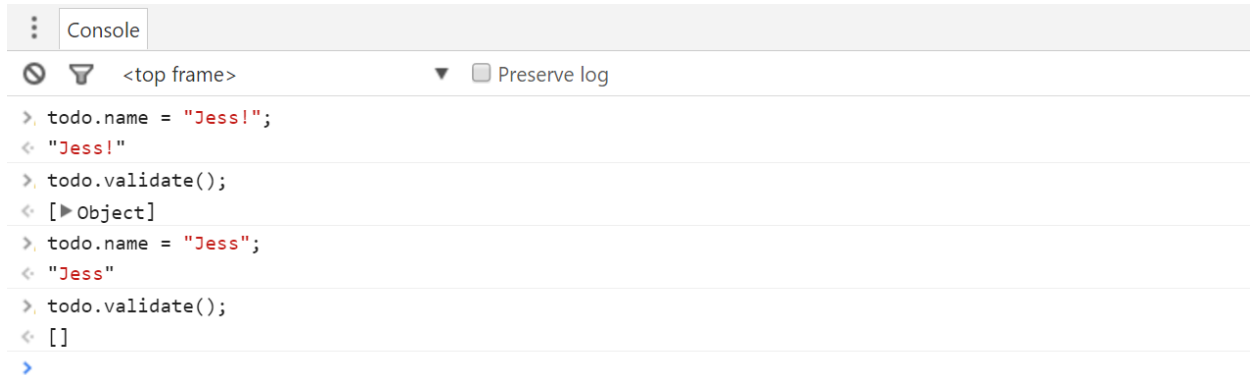
```

> todo.name = "Jess!";
< "Jess!"
> todo.validate();
< {
  isValid: false,
  message: "name does not match /^[a-zA-Z ]*$/",
  property: "name",
  __proto__: Object
}

```

Regex validation error

And, of course, if I update the name property to include only alphanumeric characters, I no longer get any validation errors.



The screenshot shows a web browser's developer console with the 'Console' tab selected. The console displays the following code and output:

```
> todo.name = "Jess!";  
< "Jess!"  
> todo.validate();  
< [▶Object]  
> todo.name = "Jess";  
< "Jess"  
> todo.validate();  
< []  
>
```

No more validation errors!

Frankly, I found decorator factories a bit awkward at first – a function that returns a decorator function can seem pretty weird. But, the approach grew on me and now I use both decorators and decorator factories on a regular basis to centralize all of the cross-cutting concerns in my application such as logging and validation. I have even used decorators to implement far more complex concepts such as dependency injection! I consider decorators to be a great help and I'm willing to bet that once you start using them, it's going to be hard for you to stop!

12. Appendix: Syntax Examples

ECMAScript 6 Features

Optional/Default parameters

```
// "final" and "interval" parameters are optional.
// If omitted, they will be defaulted to 0 and 1 (respectively)
function countdown(initial, final = 0, interval = 1) {
    // Countdown logic
}

// You can then call the method with one, two, or three parameters:
countdown(10)           // -> countdown(10, 0, 1)
countdown(10, -10)      // -> countdown(10, -10, 1)
countdown(10, -10, 5)
```

Template strings

```
var name = 'Ada Lovelace';
var birthDate = new Date(1815, 12, 10);

var template = `
    <div class="user">
        <p class='name'>${name}</p>
        <p>
            <span class='birthDate'>${birthDate.toDateString()}</span>
            <span class='age'>
                ${ Math.floor( (new Date() - birthDate) / 31536000000 ) }
                years old
            </span>
        </p>
    </div>
`;
```

let and const

let

Use `let` anywhere that you previously used `var`, especially in places where you wouldn't expect variables to creep out of their scope, such as:

```
for(var x = 0; x <= 5; x++) {  
    var counter = x;  
}
```

```
console.log(counter); // Prints 5... how odd!
```

Instead, with `let`:

```
for(let x = 0; x <= 5; x++) {  
    let counter = x;  
}
```

```
console.log(counter); // Browser says "I don't know about 'counter'" (like you'd\ expect!)
```

const

```
for(let x = 0; x < 5; x++) {  
    const counter = x; // Assign a constant to an initial value...  
    counter = 1;        // Try to reassign and get a runtime error, "can't reass\  \n    ign constant variable"  
}
```

For..of loops

Use `for/of` anywhere that you previously would have used `for/in`:

```
var array = [ "Pick up drycleaning", "Clean Batcave", "Save Gotham" ];

// Old way:
for(var index in array) {
    var value = array[index];
    console.log(value);
}

// New way:
for(var value of array) {
    console.log(value);
}
```

Arrow Functions

Use arrow functions *almost* anywhere that you would have used an anonymous function:

```
var todos = [
    { name: "Pick up drycleaning", completed: false },
    { name: "Clean Batcave", completed: true },
    { name: "Save Gotham", completed: false }
];

// Old way:
var completed = todos.filter(function(todo) {
    return todo.completed;
});

// New way:
var completed = todos.filter((todo) => { return todo.completed; });

// Of course, the parentheses around the parameter list are optional if there \
is only one parameter:
var completed = todos.filter(todo => { return todo.completed; });

// And if you've only got one line/expression, you don't even need the bracket\
s.
// (Bonus: the return value is the result of the expression!)
var completed = todos.filter(todo => todo.completed);
```

Caveat: the **this** keyword

The only time arrow functions actually change the functionality of a function is by saving a reference to the **this** keyword. Often, this is exactly what you want to happen!

```
function CounterButton(element) {

  this.counter = 0;

  element.addEventListener('click', function() {
    this.counter += 1; // Doesn't work :(
  })

  // Works great!
  element.addEventListener('click', () => this.counter += 1);

}
```

Destructuring

Assigning a set of variables from a simple array:

```
var array = [123, "Pick up drycleaning", false];
var [id, title, completed] = array; // -> id = 123, title = "Pick up drycle\
aning", completed = false
```

Swapping the values of variables:

```
var a = 1;
var b = 5;

[a, b] = [b, a]; // -> a = 5, b = 1
```

Assigning a set of variables from an object:

```
var todo = {
  id: 123,
  title: "Pick up drycleaning",
  completed: false
};

var { id, title, completed } = todo; // -> id = 123, title = "Pick up drycle\
eaning", completed = false
```

Assigning a set of variables from an object where the target variable name is different from the object property name:

```
var { completed: isCompleted, title, id } = todo; // -> id = 123, title = "\
Pick up drycleaning", isCompleted = false
```

You can even set default values:

```
var todo { name: 'Pick up drycleaning' };
var { name, isCompleted = false } = todo; // -> name = 'Pick up drycleaning'\
, isCompleted = false
```

And, perhaps the coolest usage: destructuring a function parameter object to a set of variables in the function (with default values!):

```
function countdown({ initial, final: final = 0, interval: interval = 1, initia\
l: current }) {
  // Countdown logic...
}

countdown({ initial: 20 }) // -> initial = 20, fina\
l = 0, interval = 1, current = 20
countdown({ initial: 20, final: -10 }) // -> initial = 20, fina\
l = -10, interval = 1, current = 20
countdown({ initial: 20, final: -10, interval: 5 }) // -> initial = 20, fina\
l = -10, interval = 5, current = 20
```

The spread operator

Use the spread operator (...) to accept any number of function parameters into a single array variable:

```
function add(...values) {
  return values.reduce( (a, b) => a + b );
}
```

The function can still have other parameters, as long as the spread parameter comes last:

```
function calculate(action, ...values) {  
  switch (action) {  
  
    case 'add':  
      return values.reduce( (a, b) => a + b );  
  
    case 'subtract':  
      return values.reduce( (a, b) => a - b );  
  
    default:  
      throw `Unknown action ${action}!`;  
  }  
}
```

Or, use it to concatenate arrays together:

```
var source = [ 3, 4, 5 ];  
var target = [ 1, 2, ...source, 6, 7 ]; // -> [ 1, 2, 3, 4, 5, 6, 7 ]
```

Computed properties

Define property names on the fly:

```
const osPrefix = 'os_';  
  
var support = {  
  [osPrefix + 'Windows']: false,  
  [osPrefix + 'iOS']: true,  
  [osPrefix + 'Android']: true,  
};  
  
support // -> { os_Windows: false, os_iOS: true, os_Android: true }
```

Fundamentals

Specifying a type

Declaring the type of a variable:

```
var name: string;
```

Or, using type inference, just assign it to a string!

```
var name = 'Jess';
```

Or, if you really want to be explicit, do both:

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

Alternatively, if you want to completely and explicitly **opt out** of typing, you can specify the any type:

```
var name: any;
```

Of course, function parameters are just variables, so you can use the same syntax there, too:

```
function sayName(name: string) { }
```

And speaking of functions, use the same syntax to identify the function's return type:

```
function add(x: number, y: number): number { return x + y; }
```

Union Types

Want to accept multiple different types for the same parameter?

No problem (as long as they share the same base type... like `object` for instance). You can even access any properties that they both share!

```
function getLength(x: (string | any[])): number { return x && x.length; }
```

Function overloads

Many people prefer the function overload option to define a function that accepts multiple different parameter types, like this:

```
function getLength(x: string): number function getLength(x: any[]): number function getLength(x: any): number { return x && x.length; }
```

Custom Types

TypeScript is all about types - heck, it's even in the name!

In addition to the standard JavaScript types, there are multiple ways to define your own custom types.

Interface

The TypeScript syntax to define an interface should be incredibly familiar to anyone who's used C, C++, C#, or Java:

```
interface IAnimal {  
  
    // Can have properties:  
    name: string;  
    birthYear: number;  
  
    // Or methods:  
    getAge(): number;  
  
}
```

Enum

The TypeScript syntax to define an enum should be incredibly familiar to anyone who's used C, C++, C#, or Java:

```
enum Gender { Female, Male, Other }
```

You can even give them custom values:

```
enum Gender { Female = 20, Male = 30, Other // Guess what value this is? (Hint: Male + 1) }
```

Once you've defined an enum, you can access its numeric value or its textual value:

```
Colors.Female // -> 20 Colors[Colors.Female] // -> "Female"
```

Anonymous type

You don't have to define a type in order to use it as a constraint - just define an anonymous type (looks a lot like a JSON object) right there inline in place of where you'd use a type name:

```
// Accepts any object that has a "length" property that's a number type function getLength(x: {  
length: number }): number { return x.length; }
```

Classes

Basic Class

```
class Animal {  
  
    // Can have properties:  
    name: string;  
    birthYear: number;  
  
    // Even private/protected properties:  
    private _birthYear: number;  
    private _id: number;  
  
    // Can have getter/setter properties  
    // Note: they look like properties from the outside,  
    //       which means that they even satisfy an interface!  
    get birthYear(): number {  
        return this._birthYear;  
    }  
    set birthYear(year: number) {  
        this._birthYear = year;  
    }  
  
    get id(): number {  
        return this._id;  
    }  
  
    // A constructor (aka the constructor function)  
    constructor(name: string) {  
        this.name = name;  
        this._id = Animal.generateId();  
    }  
  
    // Or methods:  
    getAge(): number {  
        return new Date().getFullYear() - this.birthYear;  
    }  
  
    // Even static methods and properties...  
  
    static _lastId = 0;
```

```
static generateId(): number {
  return Animal._lastId += 1;
}

}
```

And, by the way, since it is common to accept a constructor parameter and immediately save the value to a property, there's a shortcut for that:

```
class Animal {

  // Automatically creates a string property called "name"
  constructor(public name: string) {
  }

}

// Example usage:
let animal = new Animal('Fido');
console.log(animal.name);           // -> "Fido"
```

Inheritance

In order to inherit from another class, just use the `extends` keyword:

```
class Dog extends Animal {

  speak(): string {
    return 'Woof!';
  }

}
```

Extended classes also inherit constructors from their base classes:

```
let animal = new Dog('Fido'); // Works!
```

Abstract base class

If you want to make sure that a class is only ever used as a base class, you can mark it as `abstract`:

```
abstract class Animal { // ... }
```

You can also make methods abstract as well:

```
abstract class Animal { // ...
```

```
abstract speak(): string;
```

```
}
```

Then you extend the base class and implement its abstract members just like you would if it were a regular class:

```
class Dog extends Animal {
```

```
    speak(): string {
```

```
        return 'Woof!';
```

```
    }
```

```
}
```

Implementing an interface

You can also implement an interface, which looks similar to inheriting from a class, only using the `implements` keyword instead of `extends`:

```
class Dog implements IAnimal { // ... }
```

And there's nothing stopping a class from both implementing an interface and extending another class:

```
class Dog extends Animal implements IAnimal { // ... }
```

Plus, TypeScript is smart enough to know that if a base class implements an interface, so do all of the classes that inherit/extend from that base class:

```
abstract class Animal implements IAnimal { // ... }
```

```
// Also implements IAnimal! class Dog extends Animal { // ... }
```

```
let animal: IAnimal = new Dog('Fido'); // Works!
```

Generics

Generic function

```
function clone<T>(value: T): T {  
    return JSON.parse(JSON.stringify(value));  
}
```

Generic class

```
class KeyValuePair<T, U> { key: T; value: U; }
```

Generic constraints

You can constrain the types you can use as generic parameters by saying that it “extends” a type:

```
function totalLength<T extends { length: number }>(x: T, y: T) {  
    return x.length + y.length;  
}
```

That’s an example with an anonymous type, but of course you can use an interface or class as well:

```
interface IHaveALength { length: number; }
```

```
function totalLength<T extends IHaveALength>(x: T, y: T) {  
    return x.length + y.length;  
}
```

Modules

Internal Modules (Namespaces)

Use the namespace keyword, which acts kind of like an IIFE:

```
namespace TodoApp.Model {  
  
    interface Todo {  
        id: number;  
        name: string;  
        state: TodoState;  
    }  
  
    enum TodoState {  
        New = 1,  
        Active,  
        Complete,  
        Deleted  
    }  
  
    var someVariable; // only accessible within this block of code  
}
```

var todo: TodoApp.Model.Todo; // ERROR! Don't know about that type!(!?)

If you want to access members of a namespace (even in the same file), you need to export them:

```
namespace TodoApp.Model {  
  
    export interface Todo {  
        id: number;  
        name: string;  
        state: TodoState;  
    }  
  
    export enum TodoState {  
        New = 1,  
        Active,  
        Complete,  
        Deleted  
    }  
  
}
```

```
var todo: TodoApp.Model.Todo; // Ah, now it works!
```

You can define an alias so that you don't need to keep referring to a namespace with its full-fledged name:

```
namespace TodoApp.App {  
  
import Model = TodoApp.Model;  
  
var todo: Model.Todo;  
  
}
```

External Modules (exports)

They work the same as namespaces, except the *file* is the “namespace”.

Just be sure to set the `compileSettings.module` setting:

tsconfig.json

```
{  
  "compilerOptions": {  
    "target": "es5",  
    "module": "system",  
  }  
}
```

Import with require

Say I'm in other file in the same folder as the `Model.ts` file I've been showing. I can refer to items in that `Model.ts` file with this syntax:

```
import Model = require('./Model');
```

I can even establish an alias for items in that file like this:

```
import Model = require('./Model');  
import Todo = Model.Todo;
```

Import with ECMAScript `import` keyword

Say I'm in other file in the same folder as the `Model.ts` file I've been showing. I can refer to items in that `Model.ts` file with this syntax:


```
import * as Model from './Model';
```

I can even establish an alias for items in that file by importing them explicitly like this:

```
import { Todo } from './model';
```

Decorators

Since decorators are an experimental feature, you first have to enable them in your configuration:

tsconfig.json

```
{
  "compilerOptions": {
    "experimentalDecorators": true
  }
}
```

Class Decorator

First, create your decorator function, which is just a plain old function that accepts the target parameter:

```
function validatable(target: Function) { target.prototype.validate = function() { console.log('TODO: implement validation!'); }; }
```

Then, apply it to your class:

```
@validatable class Request { }
```

In this example, the `validatable` decorator dynamically adds a new method called `validate` to every `Request` object, which means that I can do this:

```
let request = new Request(); request.validate(); // -> "TODO: implement validation!"
```

Method Decorator

Method decorators are the same basic principle as Class Decorators, only with a slightly more complex signature that looks like this:

Here's a working example of a "log" decorator that wraps a method and logs its parameters and return values:

```
function log(
  target: Object, methodName: string,
  descriptor: TypedPropertyDescriptor<Function>
) {

  var originalMethod = descriptor.value;

  descriptor.value = function (...args) {
    console.log(`${methodName}(${JSON.stringify(args)})`)
  }
}
```

```

    let returnValue = originalMethod.apply(this, args);

    console.log(`${methodName}(${JSON.stringify(args)}) => ${JSON.stringify(\
returnValue)})`)

    return returnValue;
  }
}

```

Then just apply it like you apply a class decorator:

```
@log function getLength(x: { length: number }) { return x.length; }
```

Property Decorator

A property decorator is just a function with this signature:

```
function name(target: Object, propertyKey: string | symbol) => void;
```

target is the instance of the object being created.

For example:

```

function required(target: Object, propertyName: string) {
  console.log(`${propertyName} is required`);
}

class SomeClass {

  @required
  importantProperty;

}

new SomeClass(); // -> "importantProperty is required"

```

Decorator Factory

A decorator factory is a function that returns a decorator function, like this:

```
function regex(pattern: string) {  
    return function (target: Object, propertyName: string) {  
        console.log(`${propertyName} must match regex ${pattern}`);  
    }  
}
```

```
class SomeClass {
```

```
    @regex(' [a-zA-Z0-9] ')  
    importantProperty;
```

```
}
```

```
new SomeClass(); // -> "importantProperty must match regex [a-zA-Z0-9]"
```