Mastering TypeScript

The TypeScript compiler and language has brought JavaScript development up to the enterprise level, yet still maintains backward compatibility with existing JavaScript browsers and libraries.

Packed with practical code samples, this book brings the benefits of strongly typed, object-oriented programming and design principles into the JavaScript development space. Starting with core language features, and working through more advanced topics such as generics and modules, you will learn how to gain maximum benefit from your JavaScript development with TypeScript. With a strong focus on test-driven development and coverage of many popular JavaScript frameworks, you can fast-track your TypeScript knowledge to a professional level. By the end of this book, you will be able to confidently implement a TypeScript application from scratch.

Who this book is written for

Whether you are a JavaScript developer aiming to learn TypeScript, or an experienced TypeScript developer wanting to take your skills to the next level, this book is for you. From basic to advanced language constructs, test-driven development, and object-oriented techniques, you will learn how to get the most out of the TypeScript language.

What you will learn from this book

- Gain an insight into core and advanced TypeScript language features including inheritance and generics
- Integrate your existing JavaScript code and third-party JavaScript libraries by writing and using declaration files
- Write TypeScript code to target popular JavaScript frameworks such as jQuery, Backbone, Angular, Node, and Marionette
- Create extensive testing suites for your application, including unit testing, integration testing, and browser automation with Jasmine and Selenium
- Organize your application code using modules, and utilize Asynchronous Module Loading with require.js
- Explore advanced object-oriented techniques including dependency injection and strongly typed domain events
- Build a complete single-page web application with TypeScript and Marionette, incorporating object-oriented design patterns along the way

In this package, you will find:

- The author biography
- A preview chapter from the book, Chapter 2 'Types, Variables and Function Techniques'
- A synopsis of the book’s content
- More information on Mastering TypeScript

About the Author

Nathan Rozental has been writing commercial software for over 24 years. Starting with mainframe COBOL, then moving on to C, followed by C++ and Java, and finally settling on C# and ASP.NET. He has been working with and writing blogs about the TypeScript language, since its release towards the end of 2012. In TypeScript, he found a language through which he could bring all of the object-oriented design patterns and unit testing practices that he had learned over the years, through a variety of languages, to JavaScript.

Nathan currently works in the Health Industry, bringing touch-screen interfaces to medical systems; thereby enabling Bring Your Own Device (BYOD) solutions for clinicians and hospital staff. He is passionate about code quality, unit testing, and Continuous Integration (CI) and has helped many large teams implement CI, across many different software projects, in many different languages.

When he is not coding, Nathan loves windsurfing and playing soccer; he is also an avid Liverpool FC supporter. You can find Nathan's blog at http://blorkfish.wordpress.com.
Mastering TypeScript

The TypeScript language and compiler has been a huge success story since its release in late 2012. It has quickly carved out a solid footprint in the JavaScript development community, and continues to go from strength to strength. Many large-scale JavaScript projects, including projects by Adobe, Mozilla, and Asana, have made the decision to switch their code base from JavaScript to TypeScript. Recently, the Microsoft and Google teams announced that Angular 2.0 will be developed with TypeScript, thereby merging the AtScript and TypeScript languages into one.

This large-scale industry adoption of TypeScript shows the value of the language, the flexibility of the compiler, and the productivity gains that can be realized with its rich development toolset. On top of this industry support, the ECMAScript 6 standard is getting closer and closer to publication, and TypeScript provides a way to use features of this standard in our applications today.

Writing JavaScript single page applications in TypeScript has been made even more appealing with the large collection of declaration files that have been built by the TypeScript community. These declaration files seamlessly integrate a large range of existing JavaScript frameworks into the TypeScript development environment, bringing with it increased productivity, early error detection, and advanced IntelliSense features.

This book is a guide for both experienced TypeScript developers, as well as those who are just beginning their TypeScript journey. With a focus on Test Driven Development, detailed information on integration with many popular JavaScript libraries, and an in-depth look at TypeScript's features, this book will help you with your exploration of the next step in JavaScript development.

What This Book Covers

Chapter 1, TypeScript – Tools and Framework Options, sets the scene for beginning TypeScript development, by firstly looking at the various benefits of using TypeScript, and then discussing how to set up a development environment.

Chapter 2, Types, Variables and Function Techniques, introduces the reader to the TypeScript language, starting with basic types and type inferences, and then moving on to discusses variables and functions.

Chapter 3, Interfaces, Classes and Generics, builds on the work from the previous chapter, and introduces the object-oriented concepts of interfaces, classes, and inheritance. It then introduces the reader to the syntax and usage of generics within TypeScript.
Chapter 4, Writing and Using Declaration Files, walks the reader through building a declaration file for an existing body of JavaScript code, and then lists some of the most common syntax used when writing declaration files. This syntax is designed to be a quick reference guide to declaration file syntax, or a cheat sheet.

Chapter 5, Third Party Libraries, shows the reader how to use declaration files from the DefinitelyTyped repository within the development environment. It then moves on to show the reader how to write TypeScript that is compatible with three popular JavaScript frameworks—Backbone, Angular, and ExtJs.

Chapter 6, Test Driven Development, starts with a discussion on what Test Driven Development is, and then guides the reader through the process of creating various types of unit tests using the Jasmine library, including data-driven and asynchronous tests. The chapter finishes with a discussion on integration testing, test reporting, and using continuous integration build servers.

Chapter 7, Modularization, looks at the two types of module generation that the TypeScript compiler uses: CommonJS and AMD. This chapter shows the reader how to build a CommonJS module for use with Node, and then discusses building AMD modules with Require, Backbone, AMD plugins, and jQuery plugins.

Chapter 8, Object-oriented Programming with TypeScript, discusses advanced object-oriented design patterns, including the Service Location Design Pattern, Dependency Injection, and the Domain Events Design Pattern. The reader is taken through the concepts and ideas of each pattern, and then shown how one might implement these patterns using TypeScript.

Chapter 9, Let's Get Our Hands Dirty, builds a single-page application using TypeScript and Marionette from the ground up. This chapter starts with a discussion on page layout and transition, using an HTML-only version of the application. It then moves on to discuss, build and test the underlying data models and Marionette views that will be used within the application. Finally, the State and Mediator Design Pattern is implemented to manage page transitions and graphical elements.
Types, Variables and Function Techniques

TypeScript introduces strong typing to JavaScript through a simple syntax, referred to by Anders Hejlsberg as "syntactic sugar".

This chapter is an introduction to the syntax used in the TypeScript language to apply strong typing to JavaScript. It is intended for readers that have not used TypeScript before, and covers the transition from standard JavaScript to TypeScript. If you already have experience with TypeScript, and have a good understanding of the topics listed below, then by all means have a quick read through, or skip to the next chapter.

We will cover the following topics in this chapter:

- Basic types and type syntax: strings, numbers, and booleans
- Inferred typing and duck-typing
- Arrays and enums
- The any type and explicit casting
- Functions and anonymous functions
- Optional and default function parameters
- Argument arrays
- Function callbacks and function signatures
- Function scoping rules and overloads
Basic types

JavaScript variables can hold a number of data types, including numbers, strings, arrays, objects, functions, and more. The type of an object in JavaScript is determined by its assignment – so if a variable has been assigned a string value, then it will be of type string. This can, however, introduce a number of problems in our code.

JavaScript is not strongly typed

As we saw in Chapter 1, TypeScript – Tools and Framework Options, JavaScript objects and variables can be changed or reassigned on the fly. As an example of this, consider the following JavaScript code:

```javascript
var myString = "test";
var myNumber = 1;
var myBoolean = true;
```

We start by defining three variables, named `myString`, `myNumber` and `myBoolean`. The `myString` variable is set to a string value of "test", and as such will be of type string. Similarly, `myNumber` is set to the value of 1, and is therefore of type number, and `myBoolean` is set to true, making it of type boolean. Now let's start assigning these variables to each other, as follows:

```javascript
myString = myNumber;
myBoolean = myString;
myNumber = myBoolean;
```

We start by setting the value of `myString` to the value of `myNumber` (which is the numeric value of 1). We then set the value of `myBoolean` to the value of `myString`, (which would now be the numeric value of 1). Finally, we set the value of `myNumber` to the value of `myBoolean`. What is happening here, is that even though we started out with three different types of variables—a string, a number, and a boolean—we are able to reassign any of these variables to one of the other types. We can assign a number to a string, a string to boolean, or a boolean to a number.

While this type of assignment in JavaScript is legal, it shows that the JavaScript language is not strongly typed. This can lead to unwanted behaviour in our code. Parts of our code may be relying on the fact that a particular variable is holding a string, and if we inadvertently assign a number to this variable, our code may start to break in unexpected ways.
TypeScript is strongly typed

TypeScript, on the other hand, is a strongly typed language. Once you have declared a variable to be of type `string`, you can only assign `string` values to it. All further code that uses this variable must treat it as though it has a type of `string`. This helps to ensure that code that we write will behave as expected. While strong typing may not seem to be of any use with simple strings and numbers—it certainly does become important when we apply the same rules to objects, groups of objects, function definitions and classes. If you have written a function that expects a `string` as the first parameter and a `number` as the second, you cannot be blamed, if someone calls your function with a `boolean` as the first parameter and something else as the second.

JavaScript programmers have always relied heavily on documentation to understand how to call functions, and the order and type of the correct function parameters. But what if we could take all of this documentation and include it within the IDE? Then, as we write our code, our compiler could point out to us—automatically—that we were using objects and functions in the wrong way. Surely this would make us more efficient, more productive programmers, allowing us to generating code with fewer errors?

TypeScript does exactly that. It introduces a very simple syntax to define the type of a variable or a function parameter to ensure that we are using these objects, variables, and functions in the correct manner. If we break any of these rules, the TypeScript compiler will automatically generate errors, pointing us to the lines of code that are in error.

This is how TypeScript got its name. It is JavaScript with strong typing - hence TypeScript. Let's take a look at this very simple language syntax that enables the "Type" in TypeScript.

Type syntax

The TypeScript syntax for declaring the type of a variable is to include a colon (`:`), after the variable name, and then indicate its type. Consider the following TypeScript code:

```typescript
var myString : string = "test";
var myNumber: number = 1;
var myBoolean : boolean = true;
```
This code snippet is the TypeScript equivalent of our preceding JavaScript code, and shows an example of the TypeScript syntax for declaring a type for the `myString` variable. By including a colon and then the keyword `string` (: `string`), we are telling the compiler that the `myString` variable is of type `string`. Similarly, the `myNumber` variable is of type `number`, and the `myBoolean` variable is of type `boolean`. TypeScript has introduced the `string`, `number` and `boolean` keywords for each of these basic JavaScript types.

If we attempt to assign a value to a variable that is not of the same type, the TypeScript compiler will generate a compile-time error. Given the variables declared in the preceding code, the following TypeScript code will generate some compile errors:

```typescript
myString = myNumber;
myBoolean = myString;
myNumber = myBoolean;
```

The TypeScript compiler is generating compile errors, because we are attempting to mix these basic types. The first error is generated by the compiler because we cannot assign a `number` value to a variable of type `string`. Similarly, the second compile error indicates that we cannot assign a `string` value to a variable of type `boolean`. Again, the third error is generated because we cannot assign a `boolean` value to a variable of type `number`.
The strong typing syntax that the TypeScript language introduces, means that we need to ensure that the types on the left-hand side of an assignment operator (=) are the same as the types on the right-hand side of the assignment operator.

To fix the preceding TypeScript code, and remove the compile errors, we would need to do something similar to the following:

```typescript
myString = myNumber.toString();
myBoolean = (myString === "test");
if (myBoolean) {
    myNumber = 1;
}
```

Our first line of code has been changed to call the `toString()` function on the `myNumber` variable (which is of type `number`), in order to return a value that is of type `string`. This line of code, then, does not generate a compile error because both sides of the equal sign are of the same type.

Our second line of code has also been changed so that the right hand side of the assignment operator returns the result of a comparison, `myString === "test"`, which will return a value of type `boolean`. The compiler will therefore allow this code, because both sides of the assignment resolve to a value of type `boolean`.

The last line of our code snippet has been changed to only assign the value `1` (which is of type `number`) to the `myNumber` variable, if the value of the `myBoolean` variable is `true`.

Anders Hejlsberg describes this feature as "syntactic sugar". With a little sugar on top of comparable JavaScript code, TypeScript has enabled our code to conform to strong typing rules. Whenever you break these strong typing rules, the compiler will generate errors for your offending code.

**Inferred typing**

TypeScript also uses a technique called inferred typing, in cases where you do not explicitly specify the type of your variable. In other words, TypeScript will find the first usage of a variable within your code, figure out what type the variable is first initialized to, and then assume the same type for this variable in the rest of your code block. As an example of this, consider the following code:

```typescript
var myString = "this is a string";
var myNumber = 1;
myNumber = myString;
```
We start by declaring a variable named `myString`, and assign a string value to it. TypeScript identifies that this variable has been assigned a value of type `string`, and will, therefore, infer any further usages of this variable to be of type `string`. Our second variable, named `myNumber` has a number assigned to it. Again, TypeScript is inferring the type of this variable to be of type `number`. If we then attempt to assign the `myString` variable (of type `string`) to the `myNumber` variable (of type `number`) in the last line of code, TypeScript will generate a familiar error message:

`error TS2011: Build: Cannot convert 'string' to 'number'`

This error is generated because of TypeScript's inferred typing rules.

**Duck-typing**

TypeScript also uses a method called duck-typing for more complex variable types. Duck-typing means that if it looks like a duck, and quacks like a duck, then it probably is a duck. Consider the following TypeScript code:

```typescript
var complexType = { name: "myName", id: 1 };
complexType = { id: 2, name: "anotherName" };
```

We start with a variable named `complexType` that has been assigned a simple JavaScript object with a `name` and `id` property. On our second line of code, we can see that we are re-assigning the value of this `complexType` variable to another object that also has an `id` and a `name` property. The compiler will use duck-typing in this instance to figure out whether this assignment is valid. In other words, if an object has the same set of properties as another object, then they are considered to be of the same type.

To further illustrate this point, let's see how the compiler reacts if we attempt to assign an object to our `complexType` variable that does not conform to this duck-typing:

```typescript
var complexType = { name: "myName", id: 1 };  // Good
complexType = { id: 2 };                     // Good
complexType = { name: "anotherName" };       // Good
complexType = { address: "address" };         // Error
```

The first line of this code snippet defines our `complexType` variable, and assigns to it an object that contains both an `id` and `name` property. From this point, TypeScript will use this inferred type on any value we attempt to assign to the `complexType` variable. On our second line of code, we are attempting to assign a value that has an `id` property but not the `name` property. On the third line of code, we again attempt to assign a value that has a `name` property, but does not have an `id` property. On the last line of our code snippet, we have completely missed the mark. Compiling this code will generate the following errors:
error TS2012: Build: Cannot convert '{ id: number; }' to '{ name: string; id: number; }':
error TS2012: Build: Cannot convert '{ name: string; }' to '{ name: string; id: number; }':
error TS2012: Build: Cannot convert '{ address: string; }' to '{ name: string; id: number; }':

As we can see from the error messages, TypeScript is using duck-typing to ensure type safety. In each message, the compiler gives us clues as to what is wrong with the offending code – by explicitly stating what it is expecting. The complexType variable has both an id and a name property. To assign a value to the complexType variable, then, this value will need to have both an id and a name property. Working through each of these errors, TypeScript is explicitly stating what is wrong with each line of code.

Note that the following code will not generate any error messages:

```javascript
var complexType = { name: "myName", id: 1 };
complexType = { name: "name", id: 2, address: "address" };
```

Again, our first line of code defines the complexType variable, as we have seen previously, with an id and a name property. Now, look at the second line of this example. The object we are using actually has three properties: name, id, and address. Even though we have added a new address property, the compiler will only check to see if our new object has both an id and a name. Because our new object has these properties, and will therefore match the original type of the variable, TypeScript will allow this assignment through duck-typing.

Inferred typing and duck-typing are powerful features of the TypeScript language – bringing strong typing to our code, without the need to use explicit typing, that is, a colon : and then the type specifier syntax.

## Arrays

Besides the base JavaScript types of string, number, and boolean, TypeScript has two other data types: Arrays and enums. Let’s look at the syntax for defining arrays.

An array is simply marked with the [] notation, similar to JavaScript, and each array can be strongly typed to hold a specific type as seen in the code below:

```javascript
var arrayOfNumbers: number[] = [1, 2, 3];
arrayOfNumbers = [3, 4, 5];
arrayOfNumbers = ["one", "two", "three"];
```
On the first line of this code snippet, we are defining an array named `arrayOfNumbers`, and further specify that each element of this array must be of type `number`. The second line then reassigns this array to hold some different numerical values.

The last line of this snippet, however, will generate the following error message:

```
error TS2012: Build: Cannot convert 'string[]' to 'number[]':
```

This error message is warning us that the variable `arrayOfNumbers` is strongly typed to only accept values of type `number`. Our code tries to assign an array of strings to this array of numbers, and is therefore, generating a compile error.

### The any type

All this type checking is well and good, but JavaScript is flexible enough to allow variables to be mixed and matched. The following code snippet is actually valid JavaScript code:

```javascript
var item1 = { id: 1, name: "item 1" };  // 1st line
item1 = { id: 2 };  // 2nd line
```

Our first line of code assigns an object with an `id` property and a `name` property to the variable `item1`. The second line then re-assigns this variable to an object that has an `id` property but not a `name` property. Unfortunately, as we have seen previously, TypeScript will generate a compile time error for the preceding code:

```
error TS2012: Build: Cannot convert '{ id: number; }' to '{ id: number; name: string; }'
```

TypeScript introduces the `any` type for such occasions. Specifying that an object has a type of `any` in essence relaxes the compiler's strict type checking. The following code shows how to use the `any` type:

```javascript
var item1: any = { id: 1, name: "item 1" };  // 1st line
item1 = { id: 2 };  // 2nd line
```

Note how our first line of code has changed. We specify the type of the variable `item1` to be of type: `any` so that our code will compile without errors. Without the type specifier of: `any`, the second line of code, would normally generate an error.

### Explicit casting

As with any strongly typed language, there comes a time where you need to explicitly specify the type of an object. This concept will be expanded upon more thoroughly in the next chapter, but it is worthwhile to make a quick note of explicit casting here. An object can be cast to the type of another by using the `< >` syntax.

---

[42]
This is not a cast in the strictest sense of the word; it is more of an assertion that is used at runtime by the TypeScript compiler. Any explicit casting that you use will be compiled away in the resultant JavaScript and will not affect the code at runtime.

Let's modify our previous code snippet to use explicit casting:

```javascript
var item1 = <any>{ id: 1, name: "item 1" };
item1 = { id: 2 };
```

Note that on the first line of this snippet, we have now replaced the `: any` type specifier on the left hand side of the assignment, with an explicit cast of `<any>` on the right hand side. This snippet of code is telling the compiler to explicitly cast, or to explicitly treat the `{ id: 1, name: "item 1" } object on the right-hand side as a type of any. So the `item1` variable, therefore, also has the type of any (due to TypeScript's inferred typing rules). This then allows us to assign an object with only the `{ id: 2 }` property to the variable `item1` on the second line of code. This technique of using the `< >` syntax on the right hand side of an assignment, is called explicit casting.

While the `any` type is a necessary feature of the TypeScript language – its usage should really be limited as much as possible. It is a language shortcut that is necessary to ensure compatibility with JavaScript, but over-use of the `any` type will quickly lead to coding errors that will be difficult to find. Rather than using the type `any`, try to figure out the correct type of the object you are using, and then use this type instead. We use an acronym within our programming teams: S.F.I.A.T. (pronounced sviat or sveat).

**Simply Find an Interface for the Any Type.** While this may sound silly – it brings home the point that the `any` type should always be replaced with an interface – so simply find it. An interface is a way of defining custom types in TypeScript, and we will cover interfaces in the next chapter. Just remember that by actively trying to define what an object's type should be, we are building strongly typed code, and therefore protecting ourselves from future coding errors and bugs.

## Enums

Enums are a special type that has been borrowed from other languages such as C#, and provide a solution to the problem of special numbers. An enum associates a human-readable name for a specific number. Consider the following code:

```javascript
enum DoorState {  
  Open,  
  Closed,  
  Ajar  
}
```
In this code snippet, we have defined an enum called `DoorState` to represent the state of a door. Valid values for this door state are Open, Closed, or Ajar. Under the hood (in the generated JavaScript), TypeScript will assign a numeric value to each of these human-readable enum values. In this example, the `DoorState.Open` enum value will equate to a numeric value of 0. Likewise, the enum value `DoorState.Closed` will be equate to the numeric value of 1, and the `DoorState.Ajar` enum value will equate to 2. Let's have a quick look at how we would use these enum values:

```javascript
window.onload = () => {
  var myDoor = DoorState.Open;
  console.log("My door state is " + myDoor.toString());
};
```

The first line within the `window.onload` function creates a variable named `myDoor`, and sets its value to `DoorState.Open`. The second line simply logs the value of `myDoor` to the console. The output of this `console.log` function would be:

**My door state is 0**

This clearly shows that the TypeScript compiler has substituted the enum value of `DoorState.Open` with the numeric value 0. Now let's use this enum in a slightly different way:

```javascript
window.onload = () => {
  var openDoor = DoorState["Closed"];
  console.log("My door state is " + openDoor.toString());
};
```

This code snippet uses a string value of "Closed" to lookup the enum type, and assign the resulting enum value to the `openDoor` variable. The output of this code would be:

**My door state is 1**

This sample clearly shows that the enum value of `DoorState.Closed` is the same as the enum value of `DoorState["Closed"]`, because both variants resolve to the numeric value of 1. Finally, let's have a look at what happens when we reference an enum using an array type syntax:

```javascript
window.onload = () => {
  var ajarDoor = DoorState[2];
  console.log("My door state is " + ajarDoor.toString());
};
```
Here, we assign the variable `openDoor` to an enum value based on the 2nd index value of the `DoorState` enum. The output of this code, though, is surprising:

**My door state is Ajar**

You may have been expecting the output to be simply `2`, but here we are getting the string "Ajar" – which is a string representation of our original enum name. This is actually a neat little trick – allowing us to access a string representation of our enum value. The reason that this is possible is down to the JavaScript that has been generated by the TypeScript compiler. Let’s have a look, then, at the closure that the TypeScript compiler has generated:

```javascript
var DoorState;
(function (DoorState) {
    DoorState[DoorState["Open"] = 0] = "Open";
    DoorState[DoorState["Closed"] = 1] = "Closed";
    DoorState[DoorState["Ajar"] = 2] = "Ajar";
})(DoorState || (DoorState = {}));
```

This strange looking syntax is building an object that has a specific internal structure. It is this internal structure that allows us to use this enum in the various ways that we have just explored. If we interrogate this structure while debugging our JavaScript, we will see the internal structure of the `DoorState` object is as follows:

```javascript
DoorState
{
    [prototype]: {...}
    [0]: "Open"
    [1]: "Closed"
    [2]: "Ajar"
    [prototype]: []
    Ajar: 2
    Closed: 1
    Open: 0
}
```

The `DoorState` object has a property called "0", which has a string value of "Open". Unfortunately, in JavaScript the number `0` is not a valid property name, so we cannot access this property by simply using `DoorState.0`. Instead, we must access this property using either `DoorState[0]` or `DoorState["0"]`. The `DoorState` object also has a property named `Open`, which is set to the numeric value `0`. The word `Open` IS a valid property name in JavaScript, so we can access this property using `DoorState["Open"]`, or simply `DoorState.Open`, which equate to the same property in JavaScript.
While the underlying JavaScript can be a little confusing, all we need to remember about enums is that they are a handy way of defining an easily remembered, human-readable name to a special number. Using human-readable enums, instead of just scattering various special numbers around in our code, also makes the intent of the code clearer. Using an application wide value named `DoorState.Open` or `DoorState.Closed` is far simpler than remembering to set a value to 0 for `Open`, 1 for `Closed`, and 3 for `Ajar`. As well as making our code more readable, and more maintainable, using enums also protects our code base whenever these special numeric values change – because they are all defined in one place.

One last note on enums – we can set the numeric value manually, if needs be:

```javascript
enum DoorState {
    Open = 3,
    Closed = 7,
    Ajar = 10
}
```

Here, we have overridden the default values of the enum to set `DoorState.Open` to 3, `DoorState.Closed` to 7, and `DoorState.Ajar` to 10.

### Const enums

With the release of TypeScript 1.4, we are also able to define `const` enums as follows:

```javascript
const enum DoorStateConst {
    Open,
    Closed,
    Ajar
}
```

```javascript
var myState = DoorStateConst.Open;
```

These types of enums have been introduced largely for performance reasons, and the resultant JavaScript will not contain the full closure definition for the `DoorStateConst` enum as we saw previously. Let's have a quick look at the JavaScript that is generated from this `DoorStateConst` enum:

```javascript
var myState = 0 /* Open */;
```

Note how we do not have a full JavaScript closure for the `DoorStateConst` enum at all. The compiler has simply resolved the `DoorStateConst.Open` enum to its internal value of 0, and removed the `const` enum definition entirely.
With const enums, we therefore cannot reference the internal string value of an enum, as we did in our previous code sample. Consider the following example:

```javascript
// generates an error
console.log(DoorStateConst[0]);
// valid usage
console.log(DoorStateConst["Open"]);
```

The first console.log statement will now generate a compile time error – as we do not have the full closure available with the property of [0] for our const enum. The second usage of this const enum is valid, however, and will generate the following JavaScript:

```javascript
console.log(0 /* "Open" */);
```

When using const enums, just keep in mind that the compiler will strip away all enum definitions and simply substitute the numeric value of the enum directly into our JavaScript code.

**Functions**

JavaScript defines functions using the `function` keyword, a set of braces, and then a set of curly braces. A typical JavaScript function would be written as follows:

```javascript
function addNumbers(a, b) {
    return a + b;
}
```

```javascript
var result = addNumbers(1, 2);
var result2 = addNumbers("1", "2");
```

This code snippet is fairly self-explanatory; we have defined a function named `addNumbers` that takes two variables and returns their sum. We then invoke this function, passing in the values of 1 and 2. The value of the variable `result` would then be $1 + 2$, which is 3. Now have a look at the last line of code. Here, we are invoking the `addNumbers` function, passing in two strings as arguments, instead of numbers. The value of the variable `result2` would then be a string, "12". This string value seems like it may not be the desired result, as the name of the function is `addNumbers`. 
Copying the preceding code into a TypeScript file would not generate any errors, but let's insert some type rules to the preceding JavaScript to make it more robust:

```javascript
function addNumbers(a: number, b: number): number {
    return a + b;
};

var result = addNumbers(1, 2);
var result2 = addNumbers("1", "2");
```

In this TypeScript code, we have added a :number type to both of the parameters of the addNumbers function (a and b), and we have also added a :number type just after the ( ) braces. Placing a type descriptor here means that the return type of the function itself is strongly typed to return a value of type number. In TypeScript, the last line of code, however, will cause a compilation error:

```javascript
e Error TS2082: Build: Supplied parameters do not match any signature of call target:
```

This error message is generated because we have explicitly stated that the function should accept only numbers for both of the arguments a and b, but in our offending code, we are passing two strings. The TypeScript compiler, therefore, cannot match the signature of a function named addNumbers that accepts two arguments of type string.

### Anonymous functions

The JavaScript language also has the concept of anonymous functions. These are functions that are defined on the fly and don't specify a function name. Consider the following JavaScript code:

```javascript
var addVar = function(a, b) {
    return a + b;
};

var result = addVar(1, 2);
```

This code snippet defines a function that has no name and adds two values. Because the function does not have a name, it is known as an anonymous function. This anonymous function is then assigned to a variable named addVar. The addVar variable, then, can then be invoked as a function with two parameters, and the return value will be the result of executing the anonymous function. In this case, the variable result will have a value of 3.
Let's now rewrite the preceding JavaScript function in TypeScript, and add some type syntax, in order to ensure that the function only accepts two arguments of type number, and returns a value of type number:

```javascript
var addVar = function(a: number, b: number): number {
    return a + b;
}

var result = addVar(1, 2);
var result2 = addVar("1", "2");
```

In this code snippet, we have created an anonymous function that accepts only arguments of type number for the parameters a and b, and also returns a value of type number. The types for both the a and b parameters, as well as the return type of the function, are now using the :number syntax. This is another example of the simple "syntactic sugar" that TypeScript injects into the language. If we compile this code, TypeScript will reject the code on the last line, where we try to call our anonymous function with two string parameters:

```javascript
error TS2082: Build: Supplied parameters do not match any signature of call target:
```

### Optional parameters

When we call a JavaScript function that has is expecting parameters, and we do not supply these parameters, then the value of the parameter within the function will be undefined. As an example of this, consider the following JavaScript code:

```javascript
var concatStrings = function(a, b, c) {
    return a + b + c;
}

console.log(concatStrings("a", "b", "c"));
console.log(concatStrings("a", "b"));
```

Here, we have defined a function called `concatStrings` that takes three parameters, a, b, and c, and simply returns the sum of these values. If we call this function with all three parameters, as seen in the second last line of this snippet, we will end up with the string "abc" logged to the console. If, however, we only supply two parameters, as seen in the last line of this snippet, the string "abundefined" will be logged to the console. Again, if we call a function and do not supply a parameter, then this parameter, c in our case, will be simply undefined.
TypeScript introduces the question mark ? syntax to indicate optional parameters. Consider the following TypeScript function definition:

```javascript
var concatStrings = function(a: string, b: string, c?: string) {
    return a + b + c;
}

console.log(concatStrings("a", "b", "c"));
console.log(concatStrings("a", "b"));
console.log(concatStrings("a"));
```

This is a strongly typed version of the original concatStrings JavaScript function that we were using previously. Note the addition of the ? character in the syntax for the third parameter: c?: string. This indicates that the third parameter is optional, and therefore, all of the preceding code will compile cleanly, except for the last line. The last line will generate an error:

```javascript
error TS2081: Build: Supplied parameters do not match any signature of call target.
```

This error is generated because we are attempting to call the concatStrings function with only a single parameter. Our function definition, though, requires at least two parameters, with only the third parameter being optional.

The optional parameters must be the last parameters in the function definition. You can have as many optional parameters as you want, as long as non-optional parameters precede the optional parameters.

**Default parameters**

A subtle variant on the optional parameter function definition, allows us to specify the default value of a parameter if it is not passed in as an argument from the calling code. Let's modify our preceding function definition to use an optional parameter:

```javascript
var concatStrings = function(a: string, b: string, c: string = "c") {
    return a + b + c;
}

console.log(concatStrings("a", "b", "c"));
console.log(concatStrings("a", "b"));
```
This function definition has now dropped the ? optional parameter syntax, but instead has assigned a value of "c" to the last parameter: c:string = "c". By using default parameters, if we do not supply a value for the final parameter named c, the concatStrings function will substitute the default value of "c" instead. The argument c, therefore, will not be undefined. The output of the last two lines of code will both be "abc".

```
// Note that using the default parameter syntax will automatically
// make the parameter optional.
```

### The arguments variable

The JavaScript language allows a function to be called with a variable number of arguments. Every JavaScript function has access to a special variable, named arguments, that can be used to retrieve all arguments that have been passed into the function. As an example of this, consider the following JavaScript code:

```javascript
function testParams() {
    if (arguments.length > 0) {
        for (var i = 0; i < arguments.length; i++) {
            console.log("Argument " + i + " = " + arguments[i]);
        }
    }
}

testParams(1, 2, 3, 4);
testParams("first argument");
```

In this code snippet, we have defined a function name testParams that does not have any named parameters. Note, though, that we can use the special variable, named arguments, to test whether the function was called with any arguments. In our sample, we can simply loop through the arguments array, and log the value of each argument to the console, by using an array indexer: arguments[i]. The output of the console.log calls are as follows:

```
Argument 0 = 1
Argument 1 = 2
Argument 2 = 3
Argument 3 = 4
Argument 0 = first argument
```
So, how do we express a variable number of function parameters in TypeScript? The answer is to use what are called rest parameters, or the three dots (…) syntax. Here is the equivalent testParams function, expressed in TypeScript:

```typescript
function testParams(...argArray: number[]) {
    if (argArray.length > 0) {
        for (var i = 0; i < argArray.length; i++) {
            console.log("argArray " + i + " = " + argArray[i]);
            console.log("arguments " + i + " = " + arguments[i]);
        }
    }
}

testParams(1);
testParams(1, 2, 3, 4);
testParams("one", "two");
```

Note the use of the `...argArray: number[]` syntax for our `testParams` function. This syntax is telling the TypeScript compiler that the function can accept any number of arguments. This means that our usages of this function, i.e. calling the function with either `testParams(1)` or `testParams(1, 2, 3, 4)`, will both compile correctly. In this version of the `testParams` function, we have added two `console.log` lines, just to show that the arguments array can be accessed by either the named rest parameter, `argArray[i]`, or through the normal JavaScript array, `arguments[i]`.

The last line in this sample will, however, generate a compile error, as we have defined the rest parameter to only accept numbers, and we are attempting to call the function with strings.

The subtle difference between using `argArray` and `arguments` is the inferred type of the argument. Since we have explicitly specified that `argArray` is of type `number`, TypeScript will treat any item of the `argArray` array as a number. However, the internal `arguments` array does not have an inferred type, and so will be treated as the `any` type.

We can also combine normal parameters along with rest parameters in a function definition, as long as the rest parameters are the last to be defined in the parameter list, as follows:

```typescript
function testParamsTs2(arg1: string,
                       arg2: number, ...ArgArray: number[]) {
}
```
Here, we have two normal parameters named \texttt{arg1} and \texttt{arg2} and then an \texttt{argArray} rest parameter. Mistakenly placing the rest parameter at the beginning of the parameter list will generate a compile error.

**Function callbacks**

One of the most powerful features of JavaScript—and in fact the technology that Node was built on—is the concept of callback functions. A callback function is a function that is passed into another function. Remember that JavaScript is not strongly typed, so a variable can also be a function. This is best illustrated by having a look at some JavaScript code:

```javascript
function myCallBack(text) {
    console.log("inside myCallback " + text);
}

function callingFunction(initialText, callback) {
    console.log("inside CallingFunction");
    callback(initialText);
}

callingFunction("myText", myCallBack);
```

Here, we have a function named \texttt{myCallBack} that takes a parameter and logs its value to the console. We then define a function named \texttt{callingFunction} that takes two parameters: \texttt{initialText} and \texttt{callback}. The first line of this function simply logs "inside CallingFunction" to the console. The second line of the \texttt{callingFunction} is the interesting bit. It assumes that the \texttt{callback} argument is in fact a function, and invokes it. It also passes the \texttt{initialText} variable to the \texttt{callback} function. If we run this code, we will get two messages logged to the console, as follows:

```
inside CallingFunction
inside myCallback myText
```

But what happens if we do not pass a function as a callback? There is nothing in the preceding code that signals to us that the second parameter of \texttt{callingFunction} must be a function. If we inadvertently called the \texttt{callingFunction} function with a string, instead of a function as the second parameter, as follows:

```javascript
callingFunction("myText", "this is not a function");
```

We would get a JavaScript runtime error:

```
0x800a138a - JavaScript runtime error: Function expected
```
Defensive minded programmers, however, would first check whether the callback parameter was in fact a function before invoking it, as follows:

```javascript
function callingFunction(initialText, callback) {
    console.log("inside CallingFunction");
    if (typeof callback == "function") {
        callback(initialText);
    } else {
        console.log(callback + " is not a function");
    }
}

callingFunction("myText", "this is not a function");
```

Note the third line of this code snippet, where we check the type of the callback variable before invoking it. If it is not a function, we then log a message to the console. On the last line of this snippet, we are executing the callingFunction, but this time passing a string as the second parameter.

The output of the code snippet would be:

```
inside CallingFunction
this is not a function is not a function
```

When using function callbacks, then, JavaScript programmers need to do two things; firstly, understand which parameters are in fact callbacks and secondly, code around the invalid use of callback functions.

**Function signatures**

The TypeScript "syntactic sugar" that enforces strong typing, is not only intended for variables and types, but for function signatures as well. What if we could document our JavaScript callback functions in code, and then warn users of our code when they are passing the wrong type of parameter to our functions?

TypeScript does this through function signatures. A function signature introduces a fat arrow syntax, `() =>`, to define what the function should look like. Let's re-write the preceding JavaScript sample in TypeScript:

```typescript
function myCallBack(text: string) {
    console.log("inside myCallback " + text);
}
```
function callingFunction(initialText: string, 
  callback: (text: string) => void) 
{
  callback(initialText);
}

callingFunction("myText", myCallBack);
callingFunction("myText", "this is not a function");

Our first function definition, myCallBack now strongly types the text parameter to be of type string. Our callingFunction function has two parameters; initialText, which is of type string, and callback, which now has the new function signature syntax. Let’s look at this function signature more closely:

callback: (text: string) => void

What this function definition is saying, is that the callback argument is typed (by the : syntax) to be a function, using the fat arrow syntax () =>. Additionally, this function takes a parameter named text that is of type string. To the right of the fat arrow syntax, we can see a new TypeScript basic type, called void. Void is a keyword to denote that a function does not return a value.

So, the callingFunction function will only accept, as its second argument, a function that takes a single string parameter and returns nothing. Compiling the preceding code will correctly highlight an error in the last line of the code snippet, where we passing a string as the second parameter, instead of a callback function:

error TS2082: Build: Supplied parameters do not match any signature of call target:
Type '(text: string) => void' requires a call signature, but type 'String' lacks one

Given the preceding function signature for the callback function, the following code would also generate compile time errors:

function myCallBackNumber(arg1: number) {
  console.log("arg1 = " + arg1);
}

callingFunction("myText", myCallBackNumber);

Here, we are defining a function named myCallBackNumber, that takes a number as its only parameter. When we attempt to compile this code, we will get an error message indicating that the callback parameter, which is our myCallBackNumber function, also does not have the correct function signature:

Call signatures of types 'typeof myCallBackNumber' and '(text: string) => void' are incompatible.
The function signature of `myCallBackNumber` would actually be `(arg1:number) => void`, instead of the required `(text: string) => void`, hence the error.

In function signatures, the parameter name (`arg1` or `text`) does not need to be the same. Only the number of parameters, their types, and the return type of the function need to be the same.

This is a very powerful feature of TypeScript — defining in code what the signatures of functions should be, and warning users when they do not call a function with the correct parameters. As we saw in our introduction to TypeScript, this is most significant when we are working with third-party libraries. Before we are able to use third-party functions, classes, or objects in TypeScript, we need to define what their function signatures are. These function definitions are put into a special type of TypeScript file, called a declaration file, and saved with a `.d.ts` extension. We will take an in-depth look at declaration files in Chapter 4, Writing and Using Declaration Files.

**Function callbacks and scope**

JavaScript uses lexical scoping rules to define the valid scope of a variable. This means that the value of a variable is defined by its location within the source code. Nested functions have access to variables that are defined in their parent scope. As an example of this, consider the following TypeScript code:

```typescript
function testScope() {
    var testVariable = "myTestVariable";
    function print() {
        console.log(testVariable);
    }
}

console.log(testVariable);
```

This code snippet defines a function named `testScope`. The variable `testVariable` is defined within this function. The `print` function is a child function of `testScope`, so it has access to the `testVariable` variable. The last line of the code, however, will generate a compile error, because it is attempting to use the variable `testVariable`, which is lexically scoped to be valid only inside the body of the `testScope` function:

`error TS2095: Build: Could not find symbol 'testVariable'.`

Simple, right? A nested function has access to variables depending on its location within the source code. This is all well and good, but in large JavaScript projects, there are many different files and many areas of the code are designed to be re-usable.
Let's take a look at how these scoping rules can become a problem. For this sample, we will use a typical callback scenario—using jQuery to execute an asynchronous call to fetch some data. Consider the following TypeScript code:

```typescript
var testVariable = "testValue";

function getData() {
    var testVariable_2 = "testValue_2";
    $.ajax({
        url: "/sample_json.json",
        success: (data, status, jqXhr) => {
            console.log("success : testVariable is " + testVariable);
            console.log("success : testVariable_2 is" + testVariable_2);
        },
        error: (message, status, stack) => {
            alert("error " + message);
        }
    });
}

dataGet();
```

In this code snippet, we are defining a variable named `testVariable` and setting its value. We then define a function called `getData`. The `getData` function sets another variable called `testVariable_2`, and then calls the jQuery `$ajax` function. The `$ajax` function is configured with three properties: `url`, `success`, and `error`. The `url` property is a simple string that points to a `sample_json.json` file in our project directory. The `success` property is an anonymous function callback, that simply logs the values of `testVariable` and `testVariable_2` to the console. Finally, the `error` property is also an anonymous function callback, that simply pops up an alert.

This code runs as expected, and the `success` function will log the following results to the console:

```
success : testVariable is :testValue
success : testVariable_2 is :testValue_2
```
So far so good. Now, let's assume that we are trying to refactor the preceding code, as we are doing quite a few similar $.ajax calls, and want to reuse the success callback function elsewhere. We can easily switch out this anonymous function, and create a named function for our success callback, as follows:

```javascript
var testVariable = "testValue";

function getData() {
    var testVariable_2 = "testValue_2";
    $.ajax(
        {
            url: "/sample_json.json",
            success: successCallback,
            error: (message, status, stack) => {
                alert("error " + message);
            }
        }
    );
}

function successCallback(data, status, jqXhr) {
    console.log("success : testVariable is :" + testVariable);
    console.log("success : testVariable_2 is :" + testVariable_2);
}

dataGet();
```

In this sample, we have created a new function named `successCallback` with the same parameters as our previous anonymous function. We have also modified the $.ajax call to simply pass this function in, as a callback function for the success property: `success: successCallback`. If we were to compile this code now, TypeScript would generate an error, as follows:

```text
error TS2095: Build: Could not find symbol "testVariable_2".
```

Since we have changed the lexical scope of our code, by creating a named function, the new `successCallback` function no longer has access the variable `testVariable_2`.

It is fairly easy to spot this sort of error in a trivial example, but in larger projects, and when using third-party libraries, these sorts of errors become more difficult to track down. It is, therefore, worth mentioning that when using callback functions, we need to understand this lexical scope. If your code expects a property to have a value, and it does not have one after a callback, then remember to have a look at the context of the calling code.
Function overloads

As JavaScript is a dynamic language, we can often call the same function with different argument types. Consider the following JavaScript code:

```javascript
function add(x, y) {
    return x + y;
}

console.log("add(1,1)=" + add(1,1));
console.log("add(''1'',''1'')=" + add("1", "1");
console.log("add(true,false)=" + add(true, false));
```

Here, we are defining a simple `add` function that returns the sum of its two parameters, `x` and `y`. The last three lines of this code snippet simply log the result of the `add` function with different types: two numbers, two strings, and two boolean values. If we run this code, we will see the following output:

```
add(1,1)=2
add('1','1')=11
add(true,false)=1
```

TypeScript introduces a specific syntax to indicate multiple function signatures for the same function. If we were to replicate the preceding code in TypeScript, we would need to use the function overload syntax:

```typescript
function add(arg1: string, arg2: string): string;
function add(arg1: number, arg2: number): number;
function add(arg1: boolean, arg2: boolean): boolean;
function add(arg1: any, arg2: any): any {
    return arg1 + arg2;
}

console.log("add(1,1)=" + add(1, 1));
console.log("add(''1'',''1'')=" + add("1", "1");
console.log("add(true,false)=" + add(true, false));
```

The first line of this code snippet specifies a function overload signature for the `add` function that accepts two strings and returns a `string`. The second line specifies another function overload that uses numbers, and the third line uses booleans. The fourth line contains the actual body of the function and uses the type specifier of `any`. The last three lines of this snippet show how we would use these function signatures, and are similar to the JavaScript code that we have been using previously.
There are three points of interest in the preceding code snippet. Firstly, none of the function signatures on the first three lines of the snippet actually have a function body. Secondly, the final function definition uses the type specifier of any and eventually includes the function body. The function overload syntax must follow this structure, and the final function signature, that includes the body of the function must use the any type specifier, as anything else will generate compile-time errors.

The third point to note, is that we are limiting the add function, by using these function overload signatures, to only accept two parameters that are of the same type. If we were to try and mix our types; for example, if we call the function with a boolean and a string, as follows:

```typescript
console.log("add(true,"'1'"), add(true, "1");
```

TypeScript would generate compile errors:

```typescript
error TS2082: Build: Supplied parameters do not match any signature of call target:
error TS2087: Build: Could not select overload for '"call'" expression.
```

This seems to contradict our final function definition though. In the original TypeScript sample, we had a function signature that accepted \( (\text{arg1}: \text{any}, \text{arg2}: \text{any}) \); so, in theory, this should be called when we try to add a boolean and a number. The TypeScript syntax for function overloads, however, does not allow this. Remember that the function overload syntax must include the use of the any type for the function body, as all overloads eventually call this function body. However, the inclusion of the function overloads above the function body indicates to the compiler that these are the only signatures that should be available to the calling code.

### Union types

With the release of TypeScript 1.4, we now have the ability to combine one or two types using the pipe symbol (|) to denote a Union Type. We can, therefore, rewrite our add function overrides in the previous code snippet as follows:

```typescript
function addWithUnion(
    arg1: string | number | boolean,
    arg2: string | number | boolean
): string | number | boolean {
    if (typeof arg1 === "string") {
        // arg1 is treated as a string here
        return arg1 + "is a string";
    }
}
```
if (typeof arg1 === "number") {
    // arg1 is treated as a number here
    return arg1 + 10;
}
if (typeof arg1 === "boolean") {
    // arg1 is treated as a boolean here
    return arg1 && false;
}
}

This function, named `addWithUnion` has two arguments, `arg1` and `arg2`. These arguments are now using the union type syntax to specify that these arguments can be either `string`, `number`, or `boolean`. Notice too that our return type for the function is again using union types, meaning that the function will return one of these types as well.

**Type guards**

Within the body of the `addWithUnion` function in the preceding code snippet, we check whether the type of the `arg1` argument is a string, with the statement `typeof arg1 === "string"`. This is known as a type guard and means that the type of `arg1` will be treated as a `string` within the if statement block. Within the body of the next if statement, the type of `arg1` will be treated as a number, allowing us to add 10 to its value, and in the body of the last if statement, the type will be treated as a `boolean` by the compiler.

**Type aliases**

We are also able to define an alias for a type, a union type, or a function definition. Type aliases are denoted by using the `type` keyword. We can, therefore, write our preceding `add` function as follows:

```javascript
    type StringNumberOrBoolean = string | number | boolean;

    function addWithAliases(
        arg1: StringNumberOrBoolean,
        arg2: StringNumberOrBoolean
    ): StringNumberOrBoolean {
    }
```

Here, we have defined a type alias named `StringNumberOrBoolean` that is a type union of the `string`, `number`, and `boolean` types.
Type aliases can also be used for function signatures as follows:

```typescript
type CallbackWithString = (string) => void;

function usingCallback(callback: CallbackWithString) {
    callback("this is a string");
}
```

Here, we have defined a type alias named `CallbackWithString` that is a function that takes a single `string` parameter and returns a `void`. Our `usingCallback` function accepts this type alias within the function signature as the type for the `callback` argument.

**Summary**

In this chapter, we have discussed TypeScript's basic types, variables, and function techniques. We saw how TypeScript introduces "syntactic sugar" on top of normal JavaScript code, to ensure strongly typed variables and function signatures. We also saw how TypeScript uses duck-typing and explicit casting, and finished up with a discussion on TypeScript functions, function signatures, and overloading. In the next chapter, we will build on this knowledge and see how TypeScript extends these strongly typed rules into interfaces, classes and generics.
**Where to buy this book**

You can buy Mastering TypeScript from the Packt Publishing website.

Alternatively, you can buy the book from Amazon, BN.com, Computer Manuals and most internet book retailers.

Click here for ordering and shipping details.