CHAPTER 6

JavaScript and TypeScript: Part 2

In this chapter, I describe some of the more advanced JavaScript features that are useful for Angular development. I explain how JavaScript deals with objects, including support for classes, and I explain how JavaScript functionality is packaged into JavaScript modules. I also introduce some of the features that TypeScript provides that are not part of the JavaScript specification and that I rely on for some of the examples later in the book. Table 6-1 summarizes the chapter.

Table 6-1. *Chapter Summary*

Problem	Solution	Listing
Create an object by specifying properties and values	Use the `new` keyword or use an object literal	1–3
Create an object using a template	Define a class	4, 5
Inherit behavior from another class	Use the `extends` keyword	6
Package JavaScript features together	Create a JavaScript module	7
Declare a dependency on a module	Use the `import` keyword	8–12
Declare the types used by properties, parameters, and variables	Use TypeScript type annotations	13–18
Specify multiple types	Use union types	19–21
Create ad hoc groups of types	Use tuples	22
Group values together by key	Use indexable types	23
Control access to the methods and properties in a class	Use the access control modifiers	24

Preparing the Example Project

For this chapter, I continue using the `JavaScriptPrimer` project from Chapter 5. No changes are required to prepare for this chapter, and running the following command in the `JavaScriptPrimer` folder will start the TypeScript compiler and the development HTTP server:

```
ng serve --port 3000 --open
```

A. Freeman, *Pro Angular 6*, https://doi.org/10.1007/978-1-4842-3649-9_6

A new browser window will open, but it will be empty because I removed the placeholder content in the previous chapter. The examples in this chapter rely on the browser's JavaScript console to display messages. If you look at the console, you will see the following result:

```
Total value: $2864.99
```

Working with Objects

There are several ways to create objects in JavaScript. Listing 6-1 gives a simple example to get started.

■ **Note** Some of the examples in this chapter cause the TypeScript compiler to report errors. The examples still work, and you can ignore these messages, which arise because TypeScript provides some extra features that I don't describe until later in this chapter.

Listing 6-1. Creating an Object in the main.ts File in the src Folder

```
let myData = new Object();
myData.name = "Adam";
myData.weather = "sunny";

console.log("Hello " + myData.name + ".");
console.log("Today is " + myData.weather + ".");
```

I create an object by calling `new Object()`, and I assign the result (the newly created object) to a variable called `myData`. Once the object is created, I can define properties on the object just by assigning values, like this:

```
...
myData.name = "Adam";
...
```

Prior to this statement, my object doesn't have a property called `name`. When the statement has executed, the property does exist, and it has been assigned the value `Adam`. You can read the value of a property by combining the variable name and the property name with a period, like this:

```
...
console.log("Hello " + myData.name + ".");
...
```

The result from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

Using Object Literals

You can define an object and its properties in a single step using the *object literal* format, as shown in Listing 6-2.

Listing 6-2. Using the Object Literal Format in the main.ts File in the src Folder

```
let myData = {
    name: "Adam",
    weather: "sunny"
};

console.log("Hello " + myData.name + ". ");
console.log("Today is " + myData.weather + ".");
```

Each property that you want to define is separated from its value using a colon (`:`), and properties are separated using a comma (`,`). The effect is the same as in the previous example, and the result from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

Using Functions as Methods

One of the features that I like most about JavaScript is the way you can add functions to objects. A function defined on an object is called a *method*. Listing 6-3 shows how you can add methods in this manner.

Listing 6-3. Adding Methods to an Object in the main.ts File in the src Folder

```
let myData = {
    name: "Adam",
    weather: "sunny",
    printMessages: function () {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
    }
};
myData.printMessages();
```

In this example, I have used a function to create a method called `printMessages`. Notice that to refer to the properties defined by the object, I have to use the `this` keyword. When a function is used as a method, the function is implicitly passed the object on which the method has been called as an argument through the special variable `this`. The output from the listing is as follows:

```
Hello Adam.
Today is sunny.
```

Defining Classes

Classes are templates that are used to create objects that have identical functionality. Support for classes is a recent addition to the JavaScript specification intended to make working with JavaScript more consistent with other mainstream programming languages, and classes are used throughout Angular development. Listing 6-4 shows how the functionality defined by the object in the previous section can be expressed using a class.

Listing 6-4. Defining a Class in the main.ts File in the src Folder

```
class MyClass {

    constructor(name, weather) {
        this.name = name;
        this.weather = weather;
    }

    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log("Today is " + this.weather + ".");
    }
}

let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

JavaScript classes will be familiar if you have used another mainstream language such as Java or C#. The `class` keyword is used to declare a class, followed by the name of the class, which is `MyClass` in this case.

The `constructor` function is invoked when a new object is created using the class, and it provides an opportunity to receive data values and do any initial setup that the class requires. In the example, the constructor defines `name` and `weather` parameters that are used to create variables with the same names. Variables defined like this are known as *properties*.

Classes can have methods, which defined as functions, albeit without needing to use the function keyword. There is one method in the example, called `printMessages`, and it uses the values of the `name` and `weather` properties to write messages to the browser's JavaScript console.

■ **Tip** Classes can also have static methods, denoted by the `static` keyword. Static methods belong to the class rather than the objects they create. I have included an example of a static method in Listing 6-14.

The `new` keyword is used to create an object from a class, like this:

```
...
let myData = new MyClass("Adam", "sunny");
...
```

This statement creates a new object using the `MyClass` class as its template. `MyClass` is used as a function in this situation, and the arguments passed to it will be received by the `constructor` function defined by the class. The result of this expression is a new object that is assigned to a variable called `myData`. Once you have created an object, you can access its properties and methods through the variable to which it has been assigned, like this:

```
...
myData.printMessages();
...
```

This example produces the following results in the browser's JavaScript console:

```
Hello Adam.
Today is sunny.
```

JAVASCRIPT CLASSES VS PROTOTYPES

The class feature doesn't change the underlying way that JavaScript handles types. Instead, it simply provides a way to use them that is more familiar to the majority of programmers. Behind the scenes, JavaScript still uses its traditional type system, which is based on prototypes. As an example, the code in Listing 6-4 can also be written like this:

```
var MyClass = function MyClass(name, weather) {
    this.name = name;
    this.weather = weather;
}

MyClass.prototype.printMessages = function () {
    console.log("Hello " + this.name + ". ");
    console.log("Today is " + this.weather + ".");
};

var myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

Angular development is easier when using classes, which is the approach that I have taken throughout this book. A lot of the features introduced in ES6 are classified as syntactic sugar, which means they make aspects of JavaScript easier to understand and use. The term *syntactic sugar* may seem pejorative, but JavaScript has some odd quirks, and many of these features help developers avoid common pitfalls.

Defining Class Getter and Setter Properties

JavaScript classes can define properties in their constructor, resulting in a variable that can be read and modified elsewhere in the application. Getters and setters appear as regular properties outside of the class, but they allow the introduction of additional logic, which is useful for validating or transforming new values or generating values programmatically, as shown in Listing 6-5.

Listing 6-5. Using Getters and Setters in the main.ts File in the src Folder

```
class MyClass {
    constructor(name, weather) {
        this.name = name;
        this._weather = weather;
    }

    set weather(value) {
        this._weather = value;
    }

    get weather() {
        return `Today is ${this._weather}`;
    }

    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log(this.weather);
    }
}

let myData = new MyClass("Adam", "sunny");
myData.printMessages();
```

The getter and setter are implemented as functions preceded by the `get` or `set` keyword. There is no notion of access control in JavaScript classes, and the convention is to prefix the names of internal properties with an underscore (the _ character). In the listing, the `weather` property is implemented with a setter that updates a property called `_weather` and a getter that incorporates the `_weather` value in a template string. This example produces the following results in the browser's JavaScript console:

```
Hello Adam.
Today is sunny
```

Using Class Inheritance

Classes can inherit behavior from other classes using the `extends` keyword, as shown in Listing 6-6.

Listing 6-6. Using Class Inheritance in the main.ts File in the src Folder

```
class MyClass {
    constructor(name, weather) {
        this.name = name;
        this._weather = weather;
    }

    set weather(value) {
        this._weather = value;
    }
    get weather() {
        return `Today is ${this._weather}`;
    }

    printMessages() {
        console.log("Hello " + this.name + ". ");
        console.log(this.weather);
    }
}

class MySubClass extends MyClass {

    constructor(name, weather, city) {
        super(name, weather);
        this.city = city;
    }

    printMessages() {
        super.printMessages();
        console.log(`You are in ${this.city}`);
    }
}

let myData = new MySubClass("Adam", "sunny", "London");
myData.printMessages();
```

The extends keyword is used to declare the class that will be inherited from, known as the *superclass* or *base class*. In the listing, the MySubClass inherits from MyClass. The super keyword is used to invoke the superclass's constructor and methods. The MySubClass builds on the MyClass functionality to add support for a city, producing the following results in the browser's JavaScript console:

```
Hello Adam.
Today is sunny
You are in London
```

Working with JavaScript Modules

JavaScript modules are used to manage the dependencies in a web application, which means you don't need to manage a large set of individual code files to ensure that the browser downloads all the code for the application. Instead, during the compilation process, all of the JavaScript files that the application requires are combined into a larger file, known as a *bundle*, and it is this that is downloaded by the browser.

■ **Note** Older versions of Angular relied on a module loader, which would send separate HTTP requests for the JavaScript files required by an application. Changes to the development tools have simplified this process and switch to using bundles created during the build process.

Creating and Using Modules

Each TypeScript or JavaScript file that you add to a project is treated as a module. To demonstrate, I created a folder called `modules` in the `src` folder, added to it a file called `NameAndWeather.ts`, and added the code shown in Listing 6-7.

Listing 6-7. The Contents of the NameAndWeather.ts File in the src/modules Folder

```
export class Name {
    constructor(first, second) {
        this.first = first;
        this.second = second;
    }

    get nameMessage() {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    constructor(weather, city) {
        this.weather = weather;
        this.city = city;
    }

    get weatherMessage() {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

The classes, functions, and variables defined in a JavaScript or TypeScript file can be accessed only within that file by default. The `export` keyword is used to make features accessible outside of the file so that they can be used by other parts of the application. In the example, I have applied the `export` keyword to the `Name` and `WeatherLocation` classes, which means they are available to be used outside of the module.

■ **Tip** I have defined two classes in the `NameAndWeather.ts` file, which has the effect of creating a module that contains two classes. The convention in Angular applications is to put each class into its own file, which means that each class is defined in its own module and that you will see the `export` keyword is the listings throughout this book.

The `import` keyword is used to declare a dependency on the features that a module provides. In Listing 6-8, I have used `Name` and `WeatherLocation` classes in the `main.ts` file, and that means I have to use the `import` keyword to declare a dependency on them and the module they come from.

Listing 6-8. Importing Specific Types in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
```

```
console.log(name.nameMessage);
console.log(loc.weatherMessage);
```

This is the way that I use the `import` keyword in most of the examples in this book. The keyword is followed by curly braces that contain a comma-separated list of the features that the code in the current files depends on, followed by the `from` keyword, followed by the module name. In this case, I have imported the `Name` and `WeatherLocation` classes from the `NameAndWeather` module in the `modules` folder. Notice that the file extension is not included when specifying the module.

When the changes to the `main.ts` file are saved, the Angular development tools build the project and see that the code in the `main.ts` file depends on the code in the `NameAndWeather.ts` file. This dependency ensures that the `Name` and `WeatherLocation` classes are included in the JavaScript bundle file, and you will see the following output in the browser's JavaScript console, showing that code in the module was used to produce the result:

```
Hello Adam Freeman
It is raining in London
```

Notice that I didn't have to include the `NaneAndWeather.ts` file in a list of files to be sent to the browser. Just using the `import` keyword is enough to declare the dependency and ensure that the code required by the application is included in the JavaScript file sent to the browser.

(You will see errors warning you that properties have not been defined. Ignore those warnings for the moment; I explain how they are resolved later in the chapter.)

UNDERSTANDING MODULE RESOLUTION

You will see two different ways of specifying modules in the `import` statements in this book. The first is a relative module, in which the name of the module is prefixed with `./`, like this example from Listing 6-8:

```
...
import { Name, WeatherLocation } from "./modules/NameAndWeather";
...
```

This statement specifies a module located relative to the file that contains the `import` statement. In this case, the `NameAndWeather.ts` file is in the `modules` directory, which is in the same directory as the `main.ts` file. The other type of import is nonrelative. Here is an example of a nonrelative import from Chapter 2 and one that you will see throughout this book:

```
...
import { Component } from "@angular/core";
...
```

The module in this `import` statement doesn't start with `./`, and the build tools resolve the dependency by looking for a package in the `node_modules` folder. In this case, the dependency is on a feature provided by the `@angular/core` package, which is added to the project when it is created by the `ng new` command.

Renaming Imports

In complex projects that have lots of dependencies, it is possible that you will need to use two classes with the same name from different modules. To re-create this situation, I created a file called `DuplicateName.ts` in the `src/modules` folder and defined the class shown in Listing 6-9.

Listing 6-9. The Contents of the DuplicateName.ts File in the src/modules Folder

```
export class Name {

    get message() {
        return "Other Name";
    }
}
```

This class doesn't do anything useful, but it is called `Name`, which means that importing it using the approach in Listing 6-8 will cause a conflict because the compiler won't be able to differentiate between the two classes with that name. The solution is to use the `as` keyword, which allows an alias to be created for a class when it is imported from a module, as shown in Listing 6-10.

Listing 6-10. Using a Module Alias in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```

The `Name` class in the `DupliateName` module is imported as `OtherName`, which allows it to be used without conflicting with the `Name` class in the `NameAndWeather` module. This example produces the following output:

```
Hello Adam Freeman
It is raining in London
Other Name
```

Importing All of the Types in a Module

An alternative approach is to import the module as an object that has properties for each of the types it contains, as shown in Listing 6-11.

Listing 6-11. Importing a Module as an Object in the main.ts File in the src Folder

```
import * as NameAndWeatherLocation from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
```

```
let name = new NameAndWeatherLocation.Name("Adam", "Freeman");
let loc = new NameAndWeatherLocation.WeatherLocation("raining", "London");
let other = new OtherName();

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
```

The `import` statement in this example imports the contents of the `NameAndWeather` module and creates an object called `NameAndWeatherLocation`. This object has `Name` and `Weather` properties that correspond to the classes defined in the module. This example produces the same output as Listing 6-10.

Useful TypeScript Features

TypeScript is a superset of JavaScript, providing language features that build on those that are provided by the JavaScript specification. In the sections that follow, I demonstrate the most useful TypeScript features for Angular development, many of which I have used in the examples in this book.

■ **Tip** TypeScript supports more features than I describe in this chapter. I introduce some additional features as I use them in later chapters, but for a full reference, see the TypeScript home page at `www.typescriptlang.org`.

Using Type Annotations

The headline TypeScript feature is support for type annotations, which can help reduce common JavaScript errors by applying type checking when the code is compiled, in a way that is reminiscent of languages like C# or Java. If you have struggled to come to terms with the JavaScript type system (or didn't even realize that there was one), then type annotations can go a long way to preventing the most common errors. (On the other hand, if you like the freedom of regular JavaScript types, you may find TypeScript type annotations restrictive and annoying.)

To show the kind of problem that type annotations solve, I created a file called `tempConverter.ts` in the `JavaScriptPrimer` folder and added the code in Listing 6-12.

Listing 6-12. The Contents of the tempConverter.ts in the src Folder

```
export class TempConverter {

    static convertFtoC(temp) {
        return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

The `TempConverter` class contains a simple static method called `convertFtoC` that accepts a temperature value expressed in degrees Fahrenheit and returns the same temperature expressed in degrees Celsius.

There are assumptions in this code that are not explicit. The `convertFtoC` method expects to receive a `number` value, on which the `toPrecision` method is called to set the number of floating-point digits. The method returns a `string`, although that is difficult to tell without inspecting the code carefully (the result of the `toFixed` method is a `string`).

These implicit assumptions lead to problems, especially when one developer is using JavaScript code written by another. In Listing 6-13, I have deliberately created an error by passing the temperature as a `string` value, instead of the `number` that the method expects.

Listing 6-13. Using the Wrong Type in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC("38");

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(`The temp is ${cTemp}C`);
```

When the code is executed by the browser, you will see the following message in the browser's JavaScript console (the exact working may differ based on the browser you are using):

```
temp.toPrecision is not a function
```

This kind of issue can be fixed without using TypeScript, of course, but it does mean that a substantial amount of the code in any JavaScript application is given over to checking the types that are being used. The TypeScript solution is to make type enforcement the job of the compiler, using type annotations that are added to the JavaScript code. In Listing 6-14, I have added type annotations to the `TempConverter` class.

Listing 6-14. Adding Type Annotations in the tempConverter.ts File in the src Folder

```
export class TempConverter {

    static convertFtoC(temp: number) : string {
        return ((parseFloat(temp.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Type annotations are expressed using a colon (the `:` character) followed by the type. There are two annotations in the example. The first specifies that the parameter to the `convertFtoC` method should be a `number`.

```
...
static convertFtoC(temp: number) : string {
...
```

The other annotation specifies that the result of the method is a string.

```
...
static convertFtoC(temp: number) : string {
...
```

When you save the changes to the file, the TypeScript compiler will run. Among the errors that are reported will be this one:

```
Argument of type 'string' is not assignable to parameter of type 'number'.
```

The TypeScript compiler has examined that the type of the value passed to the `convertFtoC` method in the `main.ts` file doesn't match the type annotation and has reported an error. This is the core of the TypeScript type system; it means you don't have to write additional code in your classes to check that you have received the expected types, and it also makes it easy to determine the type of a method result. To resolve the error reported to the compiler, Listing 6-15 updates the statement that invokes the `convertFtoC` method so that it uses a `number`.

Listing 6-15. Using a Number Argument in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC(38);

console.log(name.nameMessage);
console.log(loc.weatherMessage);
console.log(other.message);
console.log(`The temp is ${cTemp}C`);
```

When you save the changes, you will see the following messages displayed in the browser's JavaScript console:

```
Hello Adam Freeman
It is raining in London
Other Name
The temp is 3.3C
```

Type Annotating Properties and Variables

Type annotations can also be applied to properties and variables, ensuring that all of the types used in an application can be verified by the compiler. In Listing 6-16, I have added type annotations to the classes in the NameAndWeather module.

Listing 6-16. Adding Annotations in the NameAndWeather.ts File in the src/modules Folder

```
export class Name {
    first: string;
    second: string;

    constructor(first: string, second: string) {
        this.first = first;
        this.second = second;
    }

    get nameMessage() : string {
        return `Hello ${this.first} ${this.second}`;
    }
}

export class WeatherLocation {
    weather: string;
    city: string;

    constructor(weather: string, city: string) {
        this.weather = weather;
        this.city = city;
    }

    get weatherMessage() : string {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

Properties are declared with a type annotation, following the same pattern as for parameter and result annotations. The changes in Listing 6-17 resolve the remaining errors reported by the TypeScript compiler, which was complaining because it didn't know what the types were for the properties created in the constructors.

The pattern of receiving constructor parameters and assigning their values to variables is so common that TypeScript includes an optimization, as shown in Listing 6-17.

Listing 6-17. Using Parameters in the NameAndWeather.ts File in the src/modules Folder

```
export class Name {

    constructor(private first: string, private second: string) {}

    get nameMessage() : string {
        return `Hello ${this.first} ${this.second}`;
    }
}
```

```
export class WeatherLocation {

    constructor(private weather: string, private city: string) {}

    get weatherMessage() : string {
        return `It is ${this.weather} in ${this.city}`;
    }
}
```

The keyword `private` is an example of an access control modifier, which I describe in the "Using Access Modifiers" section. Applying the keyword to the constructor parameter has the effect of automatically defining the class property and assigning it the parameter value. The code in Listing 6-17 is a more concise version of Listing 6-16.

Specifying Multiple Types or Any Type

TypeScript allows multiple types to be specified, separated using a bar (the | character). This can be useful when a method can accept or return multiple types or when a variable can be assigned values of different types. Listing 6-18 modifies the `convertFtoC` method so that it will accept `number` or `string` values.

Listing 6-18. Accepting Multiple Values in the tempConverter.ts File in the src Folder

```
export class TempConverter {

    static convertFtoC(temp: number | string): string {
        let value: number = (<number>temp).toPrecision
            ? <number>temp : parseFloat(<string>temp);
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

The type declaration for the `temp` parameter has changes to `number | string`, which means that the method can accept either value. This is called a *union type*. Within the method, a type assertion is used to work out which type has been received. This is a slightly awkward process, but the parameter value is cast to a number value to check whether there is a `toPrecision` method defined on the result, like this:

```
...
(<number>temp).toPrecision
...
```

The angle brackets (the < and > characters) are to declare a type assertion, which will attempt to convert an object to the specified type. You can also achieve the same result using the `as` keyword, as shown in Listing 6-19.

Listing 6-19. Using the as Keyword in the tempConverter.ts File in the src Folder

```
export class TempConverter {

    static convertFtoC(temp: number | string): string {
        let value: number = (temp as number).toPrecision
            ? temp as number : parseFloat(<string>temp);
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

An alternative to specifying a union type is to use the any keyword, which allows any type to be assigned to a variable, used as an argument, or returned from a method. Listing 6-20 replaces the union type in the convertFtoC method with the any keyword.

■ **Tip** The TypeScript compiler will implicitly apply the any keyword when you omit a type annotation.

Listing 6-20. Specifying Any Type in the tempConverter.ts File in the src Folder

```
export class TempConverter {

    static convertFtoC(temp: any): string {
        let value: number;
        if ((temp as number).toPrecision) {
            value = temp;
        } else if ((temp as string).indexOf) {
            value = parseFloat(<string>temp);
        } else {
            value = 0;
        }
        return ((parseFloat(value.toPrecision(2)) - 32) / 1.8).toFixed(1);
    }
}
```

Using Tuples

Tuples are fixed-length arrays, where each item in the array is of a specified type. This is a vague-sounding description because tuples are so flexible. As an example, Listing 6-21 uses a tuple to represent a city and its current weather and temperature.

Listing 6-21. Using a Tuple in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let name = new Name("Adam", "Freeman");
let loc = new WeatherLocation("raining", "London");
let other = new OtherName();

let cTemp = TempConverter.convertFtoC("38");

let tuple: [string, string, string];
tuple = ["London", "raining", TempConverter.convertFtoC("38")]

console.log(`It is ${tuple[2]} degrees C and ${tuple[1]} in ${tuple[0]}`);
```

Tuples are defined as an array of types, and individual elements are accessed using array indexers. This example produces the following message in the browser's JavaScript console:

```
It is 3.3 degrees C and raining in London
```

Using Indexable Types

Indexable types associate a key with a value, creating a map-like collection that can be used to gather related data items together. In Listing 6-22, I have used an indexable type to collect together information about multiple cities.

Listing 6-22. Using Indexable Types in the main.ts File in the src Folder

```
import { Name, WeatherLocation } from "./modules/NameAndWeather";
import { Name as OtherName } from "./modules/DuplicateName";
import { TempConverter } from "./tempConverter";

let cities: { [index: string]: [string, string] } = {};

cities["London"] = ["raining", TempConverter.convertFtoC("38")];
cities["Paris"] = ["sunny", TempConverter.convertFtoC("52")];
cities["Berlin"] = ["snowing", TempConverter.convertFtoC("23")];

for (let key in cities) {
    console.log(`${key}: ${cities[key][0]}, ${cities[key][1]}`);
}
```

The `cities` variable is defined as an indexable type, with the key as a string and the data value as a `[string, string]` tuple. Values are assigned and read using array-style indexers, such as `cities["London"]`. The collection of keys in an indexable type can be accessed using a `for...in` loop, as shown in the example, which produces the following output in the browser's JavaScript console:

```
London: raining, 3.3
Paris: sunny, 11.1
Berlin: snowing, -5.0
```

Only `number` and `string` values can be used as the keys for indexable types, but this is a helpful feature that I use in examples in later chapters.

Using Access Modifiers

JavaScript doesn't support access protection, which means that classes, their properties, and their methods can all be accessed from any part of the application. There is a convention of prefixing the name of implementation members with an underscore (the _ character), but this is just a warning to other developers and is not enforced.

TypeScript provides three keywords that are used to manage access and that are enforced by the compiler. Table 6-2 describes the keywords.

■ **Caution** During development, these keywords have limited effect in Angular applications because a lot of functionality is delivered through properties and methods that are specified in fragments of code embedded in data binding expressions. These expressions are evaluated at runtime in the browser, where there is no enforcement of TypeScript features. They become more important when you come to deploy the application, and it is important to ensure that any property or method that is accessed in a data binding expression is marked as `public` or has no access modifier (which has the same effect as using the `public` keyword).

Table 6-2. *The TypeScript Access Modifier Keywords*

Keyword	Description
`public`	This keyword is used to denote a property or method that can be accessed anywhere. This is the default access protection if no keyword is used.
`private`	This keyword is used to denote a property or method that can be accessed only within the class that defines it.
`protected`	This keyword is used to denote a property or method that can be accessed only within the class that defines it or by classes that extend that class.

Listing 6-23 adds a `private` method to the `TempConverter` class.

Listing 6-23. Using an Access Modifier in the tempConverter.ts File in the src Folder

```
export class TempConverter {

    static convertFtoC(temp: any): string {
        let value: number;
        if ((temp as number).toPrecision) {
            value = temp;
        } else if ((temp as string).indexOf) {
            value = parseFloat(<string>temp);
        } else {
            value = 0;
        }
        return TempConverter.performCalculation(value).toFixed(1);
    }

    private static performCalculation(value: number): number {
        return (parseFloat(value.toPrecision(2)) - 32) / 1.8;
    }
}
```

The `performCalculation` method is marked as `private`, which means the TypeScript compiler will report an error code if any other part of the application tries to invoke the method.

Summary

In this chapter, I described the way that JavaScript supports working with objects and classes, explained how JavaScript modules work, and introduced the TypeScript features that are useful for Angular development. In the next chapter, I start the process of creating a realistic project that provides an overview of how different Angular features work together to create applications before digging into individual details in Part 2 of this book.