CHAPTER 12

Using Data Bindings

The example application in the previous chapter contains a simple template that was displayed to the user and that contained a data binding that showed how many objects were in the data model. In this chapter, I describe the basic data bindings that Angular provides and demonstrate how they can be used to produce dynamic content. In later chapters, I describe more advanced data bindings and explain how to extend the Angular binding system with custom features. Table 12-1 puts data bindings in context.

Table 12-1. *Putting Data Bindings in Context*

Question	Answer
What are they?	Data bindings are expressions embedded into templates and are evaluated to produce dynamic content in the HTML document.
Why are they useful?	Data bindings provide the link between the HTML elements in the HTML document and in template files with the data and code in the application.
How are they used?	Data bindings are applied as attributes on HTML elements or as special sequences of characters in strings.
Are there any pitfalls or limitations?	Data bindings contain simple JavaScript expressions that are evaluated to generate content. The main pitfall is including too much logic in a binding because such logic cannot be properly tested or used elsewhere in the application. Data binding expressions should be as simple as possible and rely on components (and other Angular features such pipes) to provide complex application logic.
Are there any alternatives?	No. Data bindings are an essential part of Angular development.

Table 12-2 summarizes the chapter.

Table 12-2. *Chapter Summary*

Problem	Solution	Listing
Display data dynamically in the HTML document	Define a data binding	1–4
Configure an HTML element	Use a standard property or attribute binding	5, 8
Set the contents of an element	Use a string interpolation binding	6, 7
Configure the classes to which an element is assigned	Use a class binding	9–13
Configure the individual styles applied to an element	Use a style binding	14–17
Manually trigger a data model update	Use the browser's JavaScript console	18, 19

A. Freeman, *Pro Angular 6*, https://doi.org/10.1007/978-1-4842-3649-9_12

Preparing the Example Project

For this chapter, I continue using the example project from Chapter 11. To prepare for this chapter, I added a method to the component class, as shown in Listing 12-1.

■ **Tip** You can download the example project for this chapter—and for all the other chapters in this book—from `https://github.com/Apress/pro-angular-6`.

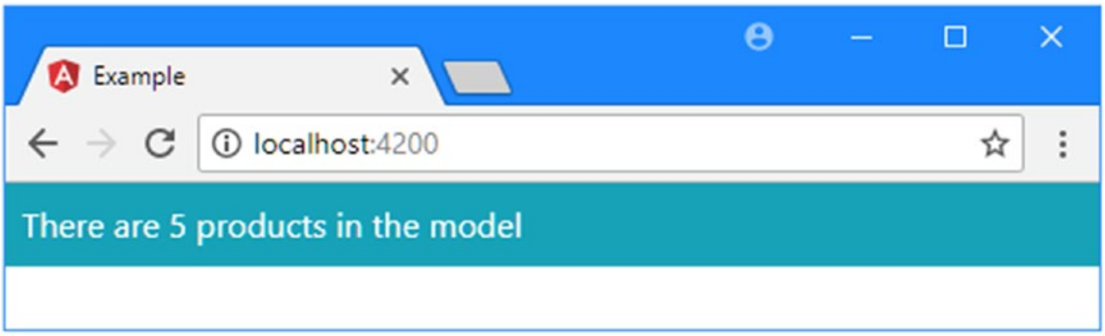
Listing 12-1. Adding a Method in the component.ts File in the src/app Folder

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

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(): string {
        return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
    }
}
```

Run the following command in the `example` folder to start the Angular development tools:

```
ng serve
```

Open a new browser and navigate to `http://localhost:4200` to see the content shown in Figure 12-1 will be displayed.



There are 5 products in the model

Figure 12-1. *Running the example application*

Understanding One-Way Data Bindings

One-way data bindings are used to generate content for the user and are the basic building block for Angular templates. The term *one-way* refers to the fact that the data flows in one direction, which means that data flows from the component to the data binding so that it can be displayed in a template.

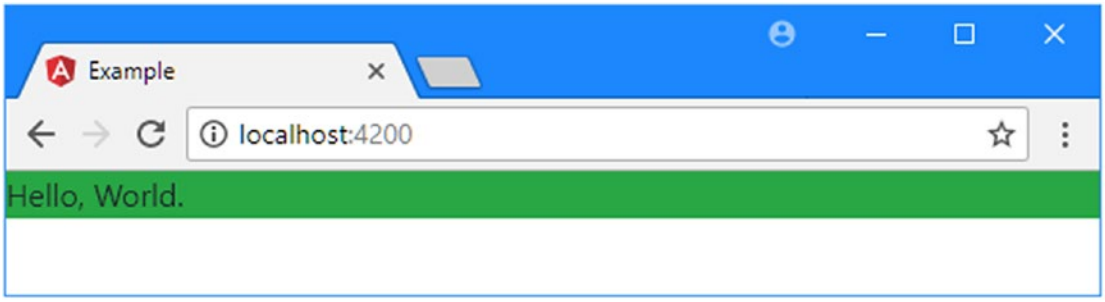
■ **Tip** There are other types of Angular data binding, which I describe in later chapters. Event bindings flow in the other direction, from the elements in the template into the rest of the application, and allow user interaction. *Two-way bindings* allow data to flow in both directions and are most commonly used in forms. See Chapters 13 and 14 for details of other bindings.

To get started with one-way data bindings, I have replaced the content of the template, as shown in Listing 12-2.

Listing 12-2. The Contents of the template.html File in the src/app Folder

```
<div [ngClass]="getClasses()" >
    Hello, World.
</div>
```

When you save the changes to the template, the development server will trigger a browser reload and display the output shown in Figure 12-2.



Hello, World.

Figure 12-2. *Using a one-way data binding*

This is a simple example, but it shows the basic structure of a data binding, which is illustrated in Figure 12-3.

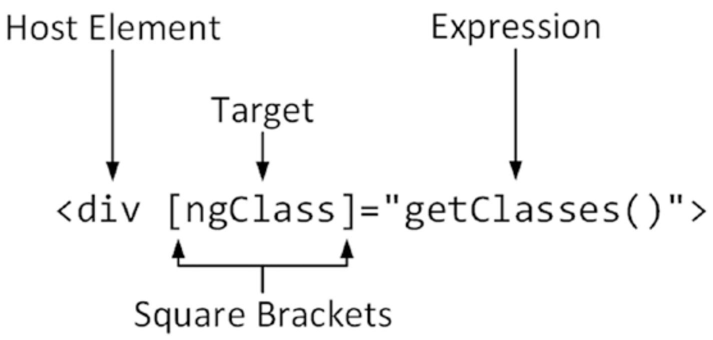


Figure 12-3. *The anatomy of a data binding*

A data binding has these four parts:

- The *host element* is the HTML element that the binding will affect, by changing its appearance, content, or behavior.
- The *square brackets* tell Angular that this is a one-way data binding. When Angular sees square brackets in a data binding, it will evaluate the expression and pass the result to the binding's *target* so that it can modify the host element.
- The *target* specifies what the binding will do. There are two different types of target: a *directive* or a *property binding*.
- The *expression* is a fragment of JavaScript that is evaluated using the template's component to provide context, meaning that the component's property and methods can be included in the expression, like the `getClasses` method in the example binding.

Looking at the binding in Listing 12-2, you can see that the host element is a `div` element, meaning that's the element that the binding is intended to modify. The expression invokes the component's `getClasses` method, which was defined at the start of the chapter. This method returns a string containing a Bootstrap CSS class based on the number of objects in the data model.

```
...
getClasses(): string {
    return this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
}
...
```

If there are five objects in the data model, then the method returns `bg-success`, which is a Bootstrap class that applies a green background. Otherwise, the method returns `bg-warning`, which is a Bootstrap class that applies an amber background.

The target for the data binding in Listing 12-2 is a directive, which is a class that is specifically written to support a data binding. Angular comes with some useful built-in directives, and you can create your own to provide custom functionality. The names of the built-in directives start with `ng`, which tells you that the `ngClass` target is one of the built-in directives. The target usually gives an indication of what the directive does, and, as its name suggests, the ngClass directive will add or remove the host element from the class or classes whose names are returned when the expression is evaluated.

Putting it all together, the data binding in Listing 12-2 will add the div element to the bg-success or bg-warning classes based on the number of items in the data model.

Since there are five objects in the model when the application starts (because the initial data is hard-coded into the SimpleDataSource class created in Chapter 12), the getClasses method returns bg-success and produces the result shown in Figure 12-3, adding a green background to the div element.

Understanding the Binding Target

When Angular processes the target of a data binding, it starts by checking to see whether it matches a directive. Most applications will rely on a mix of the built-in directives provided by Angular and custom directives that provide application-specific features. You can usually tell when a directive is the target of a data binding because the name will be distinctive and give some indication of what the directive is for. The built-in directives can be recognized by the ng prefix. The binding in Listing 12-2 gives you a hint that the target is a built-in directive that is related to the class membership of the host element. For quick reference, Table 12-3 describes the basic built-in Angular directives and where they are described in this book. (There are other directives described in later chapters, but these are the simplest and most commonly used.)

Table 12-3. *The Basic Built-in Angular Directives*

Name	Description
ngClass	This directive is used to assign host elements to classes, as described in the "Setting Classes and Styles" section.
ngStyle	This directive is used to set individual styles, as described in the "Setting Classes and Styles" section.
ngIf	This directive is used to insert content in the HTML document when its expression evaluates as true, as described in Chapter 13.
ngFor	This directive inserts the same content into the HTML document for each item in a data source, as described in Chapter 13.
ngSwitchngSwitchCasengSwitchDefault	These directives are used to choose between blocks of content to insert into the HTML document based on the value of the expression, as described in Chapter 13.
ngTemplateOutlet	This directive is used to repeat a block of content, as described in Chapter 13.

Understanding Property Bindings

If the binding target doesn't correspond to a directive, then Angular checks to see whether the target can be used to create a property binding. There are five different types of property binding, which are listed in Table 12-4, along with the details of where they are described in detail.

Table 12-4. *The Angular Property Bindings*

Name	Description
`[property]`	This is the standard property binding, which is used to set a property on the JavaScript object that represents the host element in the Document Object Model (DOM), as described in the "Using the Standard Property and Attribute Bindings" section.
`[attr.name]`	This is the attribute binding, which is used to set the value of attributes on the host HTML element for which there are no DOM properties, as described in the "Using the Attribute Binding" section.
`[class.name]`	This is the special class property binding, which is used to configure class membership of the host element, as described in the "Using the Class Bindings" section.
`[style.name]`	This is the special style property binding, which is used to configure style settings of the host element, as described in the "Using the Style Bindings" section.

Understanding the Expression

The expression in a data binding is a fragment of JavaScript code that is evaluated to provide a value for the target. The expression has access to the properties and methods defined by the component, which is how the binding in Listing 12-2 is able to invoke the `getClasses` method to provide the `ngClass` directive with the name of the class that the host element should be added to.

Expressions are not restricted to just calling methods or reading properties from the component; they can also perform most standard JavaScript operations. As an example, Listing 12-3 shows an expression that has a literal string value being concatenated with the result of the `getClasses` method.

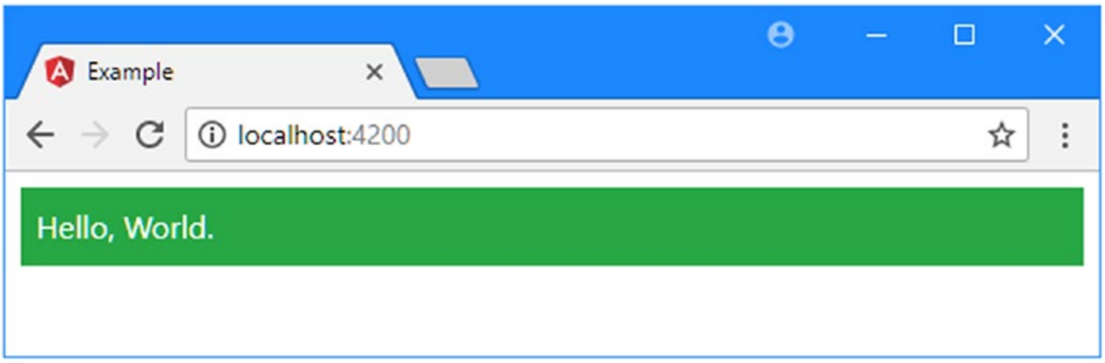
Listing 12-3. Performing an Operation in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white m-2 p-2 ' + getClasses()" >
    Hello, World.
</div>
```

The expression is enclosed in double quotes, which means that the string literal has to be defined using single quotes. The JavaScript concatenation operator is the + character, and the result from the expression will be the combination of both strings, like this:

```
text-white m-2 p-2 bg-success
```

The effect is that the `ngClass` directive will add the host element to four classes: `text-white`, `m-2`, and `p-2`, which Bootstrap uses to set the text color and add margin and padding around an element's content; and `bg-success`, which sets the background color. Figure 12-4 shows the combination of these two classes.



Hello, World.

Figure 12-4. *Combining classes in a JavaScript expression*

It is easy to get carried away when writing expressions and include complex logic in the template. This can cause problems because the expressions are not checked by the TypeScript compiler nor can they be easily unit tested, which means that bugs are more likely to remain undetected until the application has been deployed. To avoid this issue, expressions should be as simple as possible and, ideally, used only to retrieve data from the component and format it for display. All the complex retrieval and processing logic should be defined in the component or the model, where it can be compiled and tested.

Understanding the Brackets

The square brackets (the [and] characters) tell Angular that this is a one-way data binding that has an expression that should be evaluated. Angular will still process the binding if you omit the brackets and the target is a directive, but the expression won't be evaluated, and the content between the quote characters will be passed to the directive as a literal value. Listing 12-4 adds an element to the template with a binding that doesn't have square brackets.

Listing 12-4. Omitting the Brackets in a Data Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white m-2 p-2 ' + getClasses()">
  Hello, World.
</div>
<div ngClass="'text-white m-2 p-2 ' + getClasses()">
  Hello, World.
</div>
```

If you examine the HTML element in the browser's DOM viewer (by right-clicking in the browser window and selecting Inspect or Inspect Element from the pop-up menu), you will see that its `class` attribute has been set to the literal string, like this:

```
class="'text-white m-2 p-2 ' + getClasses()"
```

The browser will try to process the classes to which the host element has been assigned, but the element's appearance won't be as expected since they don't correspond to the class names used by Bootstrap. This is a common mistake to make, so it is the first thing to check if a binding doesn't have the effect you expected.

The square brackets are not the only ones that Angular uses in data bindings. For quick reference, Table 12-5 provides the complete set of brackets, the meaning of each, and where they are described in detail.

Table 12-5. *The Angular Brackets*

Name	Description
`[target]="expr"`	The square brackets indicate a one-way data binding where data flows from the expression to the target. The different forms of this type of binding are the topic of this chapter.
`{{expression}}`	This is the string interpolation binding, which is described in the "Using the String Interpolation Binding" section.
`(target) ="expr"`	The round brackets indicate a one-way binding where the data flows from the target to the destination specified by the expression. This is the binding used to handle events, as described in Chapter 14.
`[(target)] ="expr"`	This combination of brackets—known as the bana*na-in-a-box*—indicates a two-way binding, where data flows in both directions between the target and the destination specified by the expression, as described in Chapter 14.

Understanding the Host Element

The host element is the simplest part of a data binding. Data bindings can be applied to any HTML element in a template, and an element can have multiple bindings, each of which can manage a different aspect of the element's appearance or behavior. You will see elements with multiple bindings in later examples.

Using the Standard Property and Attribute Bindings

If the target of a binding doesn't match a directive, Angular will try to apply a property binding. The sections that follow describe the most common property bindings: the standard property binding and the attribute binding.

Using the Standard Property Binding

The browser uses the Document Object Model (DOM) to represent the HTML document. Each element in the HTML document, including the host element, is represented using a JavaScript object in the DOM. Like all JavaScript objects, the ones used to represent HTML elements have properties. These properties are used to manage the state of the element so that the `value` property, for example, is used to set the contents of an `input` element. When the browser parses an HTML document, it encounters each new HTML element, creates an object in the DOM to represent it, and uses the element's attributes to set the initial values for the object's properties.

The standard property binding lets you set the value of a property for the object that represents the host element, using the result of an expression. For example, setting the target of a binding to `value` will set the content of an `input` element, as shown in Listing 12-5.

Listing 12-5. Using the Standard Property Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white m-2 p-2 ' + getClasses()">
  Hello, World.
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
```

The new binding in this example specifies that the `value` property should be bound to the result of an expression that calls a method on the data model to retrieve a data object from the repository by specifying a key. It is possible that there is no data object with that key, in which case the repository method will return `null`.

To guard against using `null` for the host element's value property, the binding uses the template null conditional operator (the ? character) to safely navigate the result returned by the method, like this:

```
...
<input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
...
```

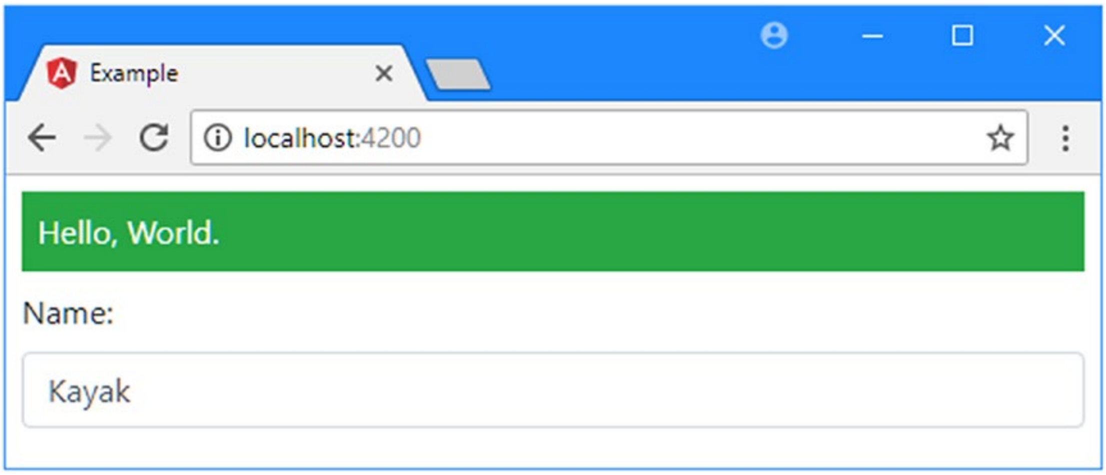
If the result from the `getProduct` method isn't `null`, then the expression will read the value of the `name` property and use it as the result. But if the result from the method is `null`, then the `name` property won't be read, and the null coalescing operator (the `||` characters) will set the result to `None` instead.

GETTING TO KNOW THE HTML ELEMENT PROPERTIES

Using property bindings can require some work figuring out which property you need to set. There is some inconsistency in the HTML specification. The name of most properties matches the name of the attribute that sets their initial value so that if you are used to setting the `value` attribute on an `input` element, for example, then you can achieve the same effect by setting the `value` property. Some property names don't match their attribute names, and some properties are not configured by attributes at all.

The Mozilla Foundation provides a useful reference for all the objects that are used to represent HTML elements in the DOM at `developer.mozilla.org/en-US/docs/Web/API`. For each element, Mozilla provides a summary of the properties that are available and what each is used for. Start with `HTMLElement` (`developer.mozilla.org/en-US/docs/Web/API/HTMLElement`), which provides the functionality common to all elements. You can then branch out into the objects that are for specific elements, such as `HTMLInputElement`, which is used to represent `input` elements.

When you save the changes to the template, the browser will reload and display an `input` element whose content is the `name` property of the data object with the key of `1` in the model repository, as shown in Figure 12-5.



Hello, World.

Name:

Kayak

Figure 12-5. *Using the standard property binding*

Using the String Interpolation Binding

Angular provides a special version of the standard property binding, known as the *string interpolation binding*, that is used to include expression results in the text content of host elements. To understand why this special binding is useful, it helps to look at what binding is required when the standard property binding is used.

The `textContent` property is used to set the content of HTML elements, which means that the content of an element can be set using a data binding like the one shown in Listing 12-6.

Listing 12-6. Setting an Element's Content in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white m-2 p-2 ' + getClasses()"
          [textContent]="'Name: ' + (model.getProduct(1)?.name || 'None')">
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
```

The expression in the new binding concatenates a literal string with the results of a method call in order to set the content of the `div` element.

The expression in this example is awkward to write, requiring careful attention to quotes, spaces, and brackets to ensure that the expected result is displayed in the output. The problem becomes worse for more complex bindings, where multiple dynamic values are interspersed among blocks of static content.

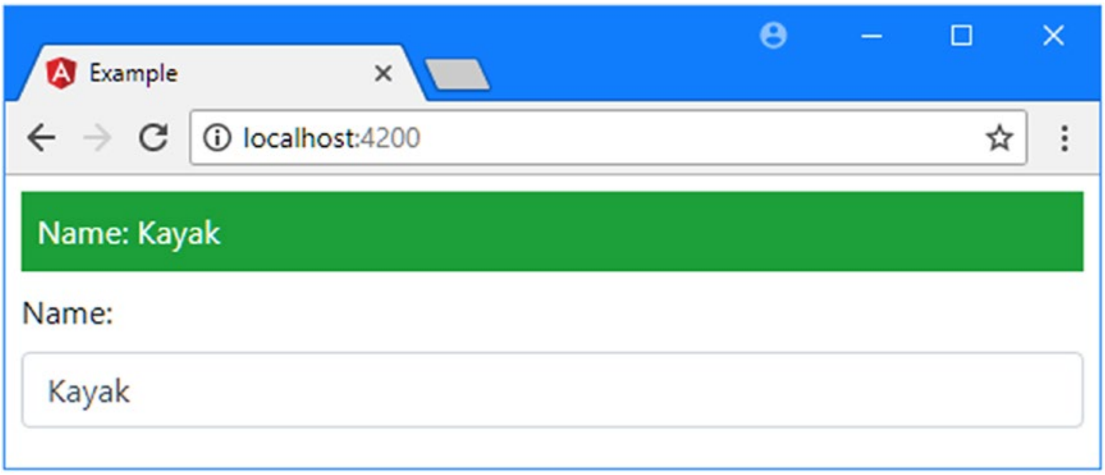
The string interpolation binding simplified this process by allowing fragments of expressions to be defined within the content of an element, as shown in Listing 12-7.

Listing 12-7. Using the String Interpolation Binding in the template.html File in the src/app Folder

```
<div [ngClass]="'text-white m-2 p-2 ' + getClasses()">
  Name: {{ model.getProduct(1)?.name || 'None' }}
</div>
<div class="form-group m-2">
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
```

The string interpolation binding is denoted using pairs of curly brackets ({{ and }}). A single element can contain multiple string interpolation bindings.

Angular combines the content of the HTML element with the contents of the brackets in order to create a binding for the `textContent` property. The result is the same as Listing 12-6, which is shown in Figure 12-6, but the process of writing the binding is simpler and less error-prone.



Name: Kayak

Name:

Kayak

Figure 12-6. *Using the string interpolation binding*

Using the Attribute Binding

There are some oddities in the HTML and DOM specifications that mean that not all HTML element attributes have equivalent properties in the DOM API. For these situations, Angular provides the *attribute binding*, which is used to set an attribute on the host element rather than setting the value of the JavaScript object that represents it in the DOM.

The most commonly encountered attribute without a corresponding property is `colspan`, which is used to set the number of columns that a `td` element will occupy in a table. Listing 12-8 shows using the attribute binding to set the `colspan` element based on the number of objects in the data model.

Listing 12-8. Using an Attribute Binding in the template.html File in the src/app Folder

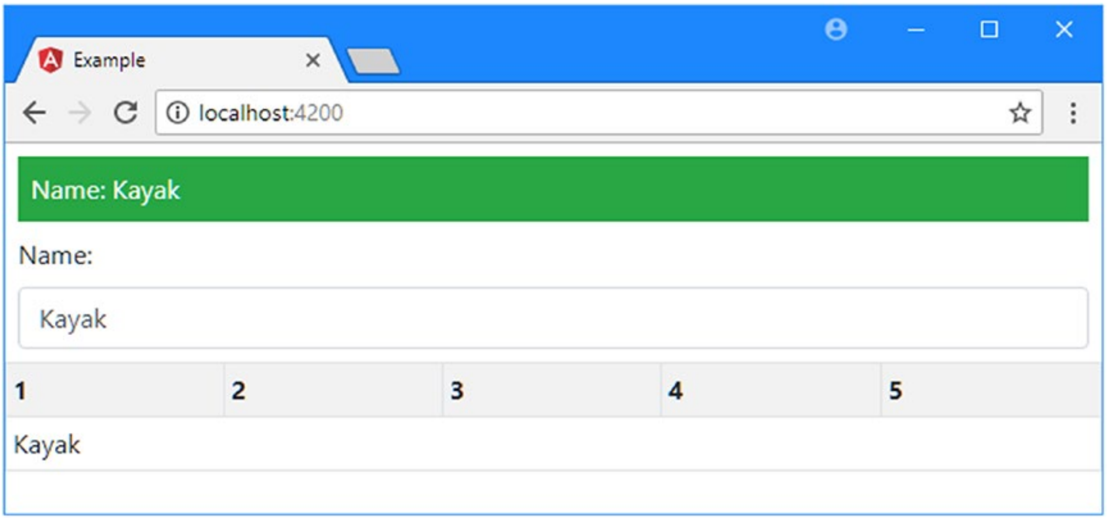
```
<div [ngClass]="'text-white m-2 p-2 ' + getClasses()">
  Name: {{model.getProduct(1)?.name || 'None'}}
</div>
<div class="form-group m-2">
```

```
  <label>Name:</label>
  <input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
</div>
<table class="table table-sm table-bordered table-striped mt-2">
    <tr>
        <th>1</th><th>2</th><th>3</th><th>4</th><th>5</th>
    </tr>
    <tr>
        <td [attr.colspan]="model.getProducts().length">
            {{model.getProduct(1)?.name || 'None'}}
        </td>
    </tr>
</table>
```

The attribute binding is applied by defining a target that prefixes the name of the attribute with `attr.` (the term `attr`, followed by a period). In the listing, I have used the attribute binding to set the value of the `colspan` element on one of the `td` elements in the table, like this:

```
...
<td [attr.colspan]="model.getProducts().length">
...
```

Angular will evaluate the expression and set the value of the `colspan` attribute to the result. Since the data model is hardwired to start with five data objects, the effect is that the `colspan` attribute creates a table cell that spans five columns, as shown in Figure 12-7.



Name: Kayak

Name:

Kayak

1	2	3	4	5
Kayak				

Figure 12-7. *Using an attribute binding*

Setting Classes and Styles

Angular provides special support in property bindings for assigning the host element to classes and for configuring individual style properties. I describe these bindings in the sections that follow, along with details of the ngClass and ngStyle directives, which provide closely related features.

Using the Class Bindings

There are three different ways in which you can use data bindings to manage the class memberships of an element: the standard property binding, the special class binding, and the ngClass directive. All three are described in Table 12-6, and each works in a slightly different way and is useful in different circumstances, as described in the sections that follow.

Table 12-6. *The Angular Class Bindings*

Example	Description
`<div [class]="expr"></div>`	This binding evaluates the expression and uses the result to replace any existing class memberships.
`<div [class.myClass]="expr"></div>`	This binding evaluates the expression and uses the result to set the element's membership of myClass.
`<div [ngClass]="map"></div>`	This binding sets class membership of multiple classes using the data in a map object.

Setting All of an Element's Classes with the Standard Binding

The standard property binding can be used to set all of an element's classes in a single step, which is useful when you have a method or property in the component that returns all of the classes that an element should belong to in a single string, with the names separated by spaces. Listing 12-9 shows the revision of the getClasses method in the component that returns a different string of class names based on the price property of a Product object.

Listing 12-9. Providing All Classes in a Single String in the component.ts File in the src/app Folder

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

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-2 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }
}
```

The result from the `getClasses` method will include the `p-2` class, which adds padding around the host element's content, for all `Product` objects. If the value of the `price` property is less than 50, the `bg-info` class will be included in the result, and if the value is 50 or more, the `bg-warning` class will be included (these classes set different background colors).

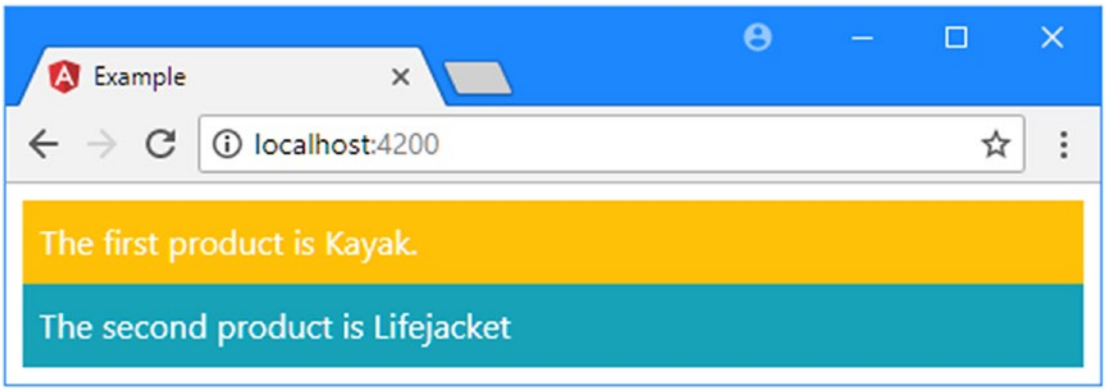
■ **Tip** You must ensure that the names of the classes are separated by spaces.

Listing 12-10 shows the standard property binding being used in the template to set the `class` property of host elements using the component's `getClasses` method.

Listing 12-10. Setting Class Memberships in the template.html File in the src/app Folder

```
<div class="text-white m-2">
  <div [class]="getClasses(1)">
    The first product is {{model.getProduct(1).name}}.
  </div>
  <div [class]="getClasses(2)">
    The second product is {{model.getProduct(2).name}}
  </div>
</div>
```

When the standard property binding is used to set the `class` property, the result of the expression replaces any previous classes that an element belonged to, which means that it can be used only when the binding expression returns all the classes that are required, as in this example, producing the result shown in Figure 12-8.



The first product is Kayak.

The second product is Lifejacket

Figure 12-8. *Setting class memberships*

Setting Individual Classes Using the Special Class Binding

The special class binding provides finer-grained control than the standard property binding and allows membership of a single class to be managed using an expression. This is useful if you want to build on the existing class memberships of an element, rather than replace them entirely. Listing 12-11 shows the use of the special class binding.

Listing 12-11. Using the Special Class Binding in the template.html File in the src/app Folder

```
<div class="text-white m-2">
  <div [class]="getClasses(1)">
    The first product is {{model.getProduct(1).name}}.
  </div>
  <div class="p-2"
       [class.bg-success]="model.getProduct(2).price < 50"
       [class.bg-info]="model.getProduct(2).price >= 50">
    The second product is {{model.getProduct(2).name}}
  </div>
</div>
```

The special class binding is specified with a target that combines the term `class`, followed by a period, followed by the name of the class whose membership is being managed. In the listing, there are two special class bindings, which manage the membership of the `bg-success` and `bg-info` classes.

The special class binding will add the host element to the specified class if the result of the expression is *truthy* (as described in the "Understanding Truthy and Falsy" sidebar). In this case, the host element will be a member of the `bg-success` class if the `price` property is less than 50 and a member of the `bg-info` class if the price property is 50 or more.

These bindings act independently from one another and do not interfere with any existing classes that an element belongs to, such as the `p-2` class, which Bootstrap uses to add padding around an element's content.

UNDERSTANDING TRUTHY AND FALSY

JavaScript has an odd feature, where the result of an expression can be truthy or falsy, providing a pitfall for the unwary. The following results are always falsy:

- The false (boolean) value
- The 0 (number) value
- The empty string ("")
- null
- undefined
- NaN (a special number value)

All other values are truthy, which can be confusing. For example, `"false"` (a string whose content is the word `false`) is truthy. The best way to avoid confusion is to only use expressions that evaluate to the `boolean` values `true` and `false`.

Setting Classes Using the ngClass Directive

The `ngClass` directive is a more flexible alternative to the standard and special property bindings and behaves differently based on the type of data that is returned by the expression, as described in Table 12-7.

Table 12-7. *The Expression Result Types Supported by the ngClass Directive*

Name	Description
String	The host element is added to the classes specified by the string. Multiple classes are separated by spaces.
Array	Each object in the array is the name of a class that the host element will be added to.
Object	Each property on the object is the name of one or more classes, separated by spaces. The host element will be added to the class if the value of the property is truthy.

The string and array features are useful, but it is the ability to use an object (known as a *map*) to create complex class membership policies that makes the `ngClass` directive especially useful. Listing 12-12 shows the addition of a component method that returns a map object.

Listing 12-12. Returning a Class Map Object in the component.ts File in the src/app Folder

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

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-2 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }

    getClassMap(key: number): Object {
        let product = this.model.getProduct(key);
        return {
            "text-center bg-danger": product.name == "Kayak",
            "bg-info": product.price < 50
        };
    }
}
```

The getClassMap method returns an object with properties whose values are one or more class names, with values based on the property values of the Product object whose key is specified as the method argument. As an example, when the key is 1, the method returns this object:

```
...
{
  "text-center bg-danger":true,
  "bg-info":false
}
...
```

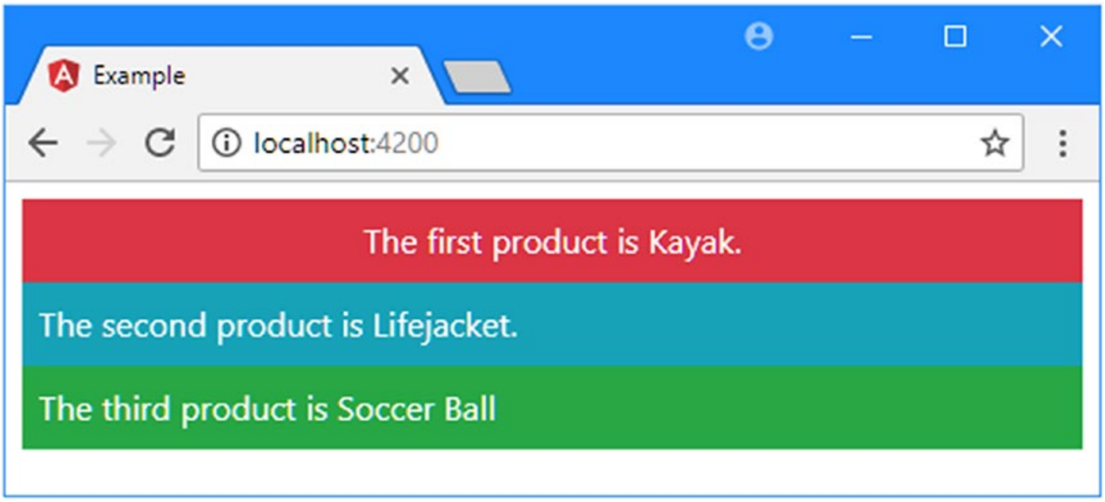
The first property will assign the host element to the text-center class (which Bootstrap uses to center the text horizontally) and the bg-danger class (which sets the element's background color). The second property evaluates to false, which means that the host element will not be added to the bg-info class. It may seem odd to specify a property that doesn't result in an element being added to a class, but, as you will see shortly, the value of expressions is automatically updated to reflect changes in the application, and being able to define a map object that specifies memberships this way can be useful.

Listing 12-13 shows the getClassMap and the map objects it returns used as the expression for data bindings that target the ngClass directive.

Listing 12-13. Using the ngClass Directive in the template.html File in the src/app Folder

```
<div class="text-white m-2">
  <div class="p-2" [ngClass]="getClassMap(1)">
    The first product is {{model.getProduct(1).name}}.
  </div>
  <div class="p-2" [ngClass]="getClassMap(2)">
    The second product is {{model.getProduct(2).name}}.
  </div>
  <div class="p-2" [ngClass]="{'bg-success': model.getProduct(3).price < 50,
                                'bg-info': model.getProduct(3).price >= 50}">
        The third product is {{model.getProduct(3).name}}
  </div>
</div>
```

The first two div elements have bindings that use the getClassMap method. The third div element shows an alternative approach, which is to define the map in the template. For this element, membership of the bg-info and bg-warning classes is tied to the value of the price property of a Product object, as shown in Figure 12-9. Care should be taken with this technique because the expression contains JavaScript logic that cannot be readily tested.



The first product is Kayak.

The second product is Lifejacket.

The third product is Soccer Ball

Figure 12-9. *Using the ngClass directive*

Using the Style Bindings

There are three different ways in which you can use data bindings to set style properties of the host element: the standard property binding, the special style binding, and the `ngStyle` directive. All three are described in Table 12-8 and demonstrated in the sections that follow.

Table 12-8. *The Angular Style Bindings*

Example	Description
`<div [style.myStyle]="expr"></div>`	This is the standard property binding, which is used to set a single style property to the result of the expression.
`<div [style.myStyle.units]="expr"></div>`	This is the special style binding, which allows the units for the style value to be specified as part of the target.
`<div [ngStyle]="map"></div>`	This binding sets multiple style properties using the data in a map object.

Setting a Single Style Property

The standard property binding and the special style bindings are used to set the value of a single style property. The difference between these bindings is that the standard property binding must include the units required for the style, while the special binding allows for the units to be included in the binding target.

To demonstrate the difference, Listing 12-14 adds two new properties to the component.

Listing 12-14. Adding Properties in the component.ts File in the src/app Folder

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

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-2 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }

    getClassMap(key: number): Object {
        let product = this.model.getProduct(key);
        return {
            "text-center bg-danger": product.name == "Kayak",
            "bg-info": product.price < 50
        };
    }

    fontSizeWithUnits: string = "30px";
    fontSizeWithoutUnits: string= "30";
}
```

The fontSizeWithUnits property returns a value that includes a quantity and the units that quantity is expressed in: 30 pixels. The fontSizeWithoutUnits property returns just the quantity, without any unit information. Listing 12-15 shows how these properties can be used with the standard and special bindings.

■ **Caution** Do not try to use the standard property binding to target the style property to set multiple style values. The object returned by the style property of the JavaScript object that represents the host element in the DOM is read-only. Some browsers will ignore this and allow changes to be made, but the results are unpredictable and cannot be relied on. If you want to set multiple style properties, then create a binding for each of them or use the ngStyle directive.

Listing 12-15. Using Style Bindings in the template.html File in the src/app Folder

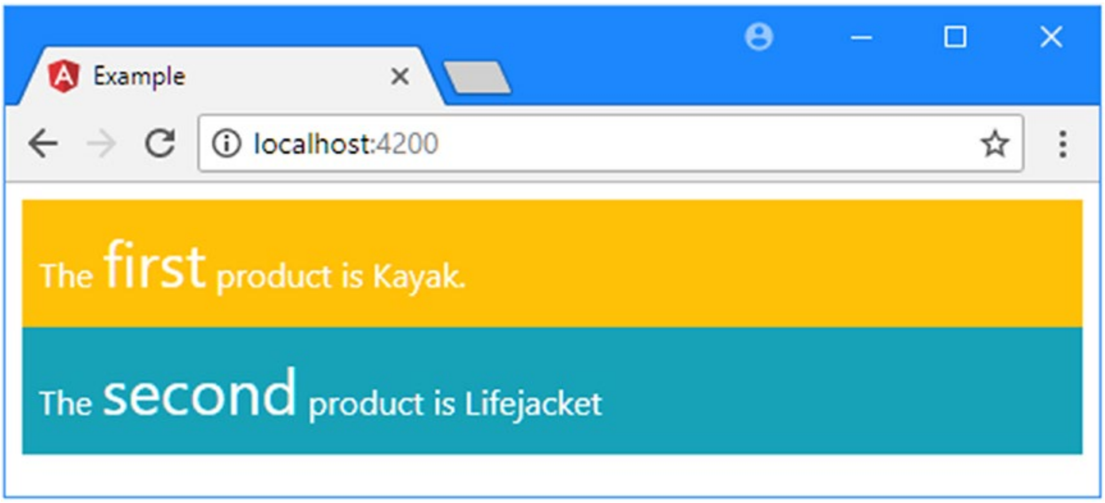
```
<div class="text-white m-2">
  <div class="p-2 bg-warning">
    The <span [style.fontSize]="fontSizeWithUnits">first</span>
    product is {{model.getProduct(1).name}}.
  </div>
  <div class="p-2 bg-info">
    The <span [style.fontSize.px]="fontSizeWithoutUnits">second</span>
    product is {{model.getProduct(2).name}}
  </div>
</div>
```

The target for the binding is `style.fontSize`, which sets the size of the font used for the host element's content. The expression for this binding uses the `fontSizeWithUnits` property, whose value includes the units, `px` for pixels, required to set the font size.

The target for the special binding is `style.fontSize.px`, which tells Angular that the value of the expression specifies the number in pixels. This allows the binding to use the component's `fontSizeWithoutUnits` property, which doesn't include units.

■ **Tip** You can specify style properties using the JavaScript property name format (`[style.fontSize]`) or using the CSS property name format (`[style.font-size]`).

The result of both bindings is the same, which is to set the font size of the `span` elements to 30 pixels, producing the result shown in Figure 12-10.



The first product is Kayak.

The second product is Lifejacket

Figure 12-10. *Setting individual style properties*

Setting Styles Using the ngStyle Directive

The `ngStyle` directive allows multiple style properties to be set using a map object, similar to the way that the `ngClass` directive works. Listing 12-16 shows the addition of a component method that returns a map containing style settings.

Listing 12-16. Creating a Style Map Object in the component.ts File in the src/app Folder

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

@Component({
    selector: "app",
    templateUrl: "template.html"
})
```

```
export class ProductComponent {
    model: Model = new Model();

    getClasses(key: number): string {
        let product = this.model.getProduct(key);
        return "p-2 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }

    getStyles(key: number) {
        let product = this.model.getProduct(key);
        return {
            fontSize: "30px",
            "margin.px": 100,
            color: product.price > 50 ? "red" : "green"
        };
    }
}
```

The map object returned by the getStyle method shows that the ngStyle directive is able to support both of the formats that can be used with property bindings, including either the units in the value or the property name. Here is the map object that the getStyles method produces when the value of the key argument is 1:

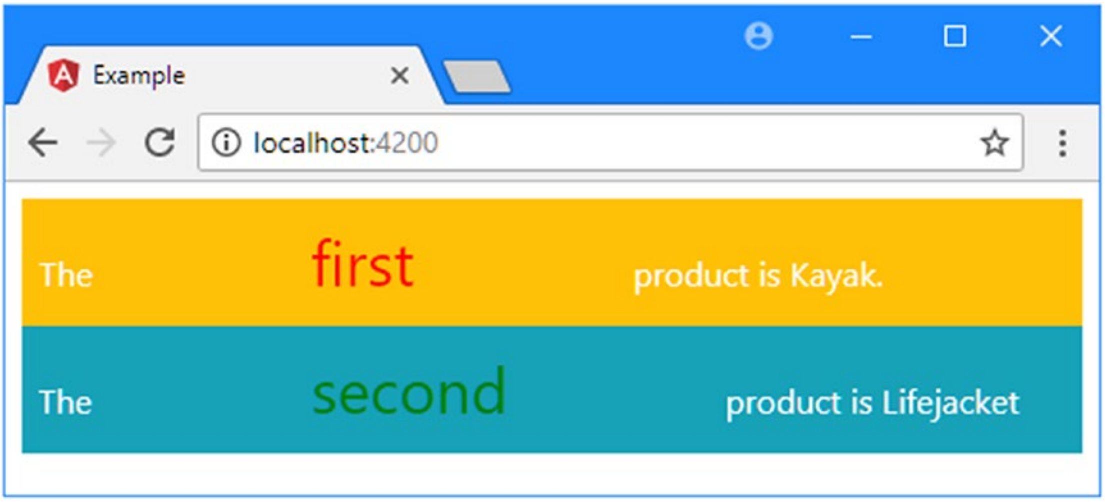
```
...
{
  "fontSize":"30px",
  "margin.px":100,
  "color":"red"
}
...
```

Listing 12-17 shows data bindings in the template that use the ngStyle directive and whose expressions call the getStyles method.

Listing 12-17. Using the ngStyle Directive in the template.html File in the src/app Folder

```
<div class="text-white m-2">
  <div class="p-2 bg-warning">
    The <span [ngStyle]="getStyles(1)">first</span>
    product is {{model.getProduct(1).name}}.
  </div>
  <div class="p-2 bg-info">
    The <span [ngStyle]="getStyles(2)">second</span>
    product is {{model.getProduct(2).name}}
  </div>
</div>
```

The result is that each span element receives a tailored set of styles, based on the argument passed to the getStyles method, as shown in Figure 12-11.



The first product is Kayak.
The second product is Lifejacket

Figure 12-11. *Using the ngStyle directive*

Updating the Data in the Application

When you start out with Angular, it can seem like a lot of effort to deal with the data bindings, remembering which binding is required in different situations. You might be wondering if it is worth the effort.

Bindings are worth understanding because their expressions are re-evaluated when the data they depend on changes. As an example, if you are using a string interpolation binding to display the value of a property, then the binding will automatically update when the value of the property is changed.

To provide a demonstration, I am going to jump ahead and show you how to take manual control of the updating process. This is not a technique that is required in normal Angular development, but it provides a solid demonstration of why bindings are so important. Listing 12-18 shows some changes to the component that enable the demonstration.

Listing 12-18. Preparing the Component in the component.ts File in the src/app Folder

```
import { ApplicationRef, Component } from "@angular/core";
import { Model } from "./repository.model";
import { Product } from "./product.model";

@Component({
    selector: "app",
    templateUrl: "template.html"
})
export class ProductComponent {
    model: Model = new Model();

    constructor(ref: ApplicationRef) {
        (<any>window).appRef = ref;
        (<any>window).model = this.model;
    }
    getProductByPosition(position: number): Product {
        return this.model.getProducts()[position];
    }

    getClassesByPosition(position: number): string {
        let product = this.getProductByPosition(position);
        return "p-2 " + (product.price < 50 ? "bg-info" : "bg-warning");
    }
}
```

I have imported the ApplicationRef type from the @angular/core module. When Angular performs the bootstrapping process, it creates an ApplicationRef object to represent the application. Listing 12-18 adds a constructor to the component that receives an ApplicationRef object as an argument, using the Angular dependency injection feature, which I describe in Chapter 19. Without going into detail now, declaring a constructor argument like this tells Angular that the component wants to receive the ApplicationRef object when a new instance is created.

Within the constructor, there are two statements that make a demonstration possible but would undermine many of the benefits of using TypeScript and Angular if used in a real project.

```
...
(<any>window).appRef = ref;
(<any>window).model = this.model;
...
```

These statements define variables in the global namespace and assign the ApplicationRef and Model objects to them. It is good practice to keep the global namespace as clear as possible, but exposing these objects allows them to be manipulated through the browser's JavaScript console, which is important for this example.

The other methods added to the constructor allow a Product object to be retrieved from the repository based on its position, rather than by its key, and to generate a class map that differs based on the value of the price property.

Listing 12-19 shows the corresponding changes to the template, which uses the ngClass directive to set class memberships and the string interpolation binding to display the value of the Product.name property.

Listing 12-19. Preparing for Changes in the template.html File in the src/app Folder

```
<div class="text-white m-2">
  <div [ngClass]="getClassesByPosition(0)">
    The first product is {{getProductByPosition(0).name}}.
  </div>
  <div [ngClass]="getClassesByPosition(1)">
    The second product is {{getProductByPosition(1).name}}
  </div>
</div>
```

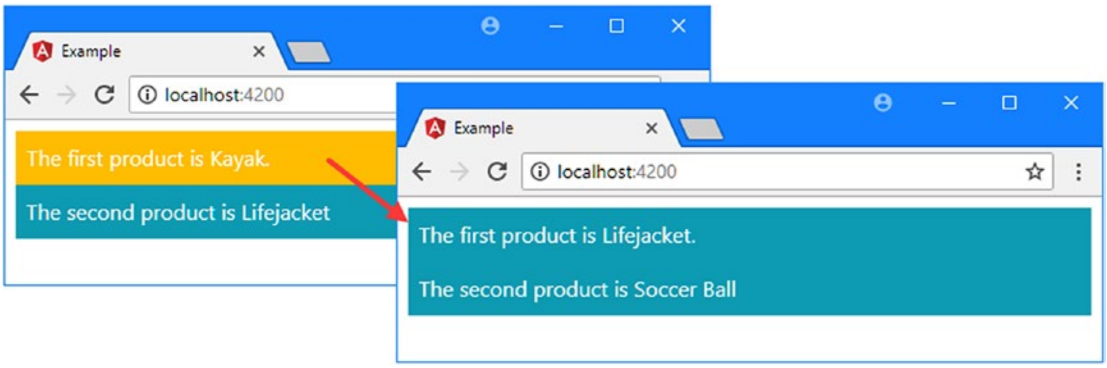
Save the changes to the component and template. Once the browser has reloaded the page, enter the following statement into the browser's JavaScript console and press Return:

```
model.products.shift()
```

This statement calls the `shift` method on the array of `Product` objects in the model, which removes the first item from the array and returns it. You won't see any changes yet because Angular doesn't know that the model has been modified. To tell Angular to check for changes, enter the following statement into the browser's JavaScript console and press Return:

```
appRef.tick()
```

The `tick` method starts the Angular change detection process, where Angular looks at the data in the application and the expressions in the data binding and processes any changes. The data bindings in the template use specific array indexes to display data, and now that an object has been removed from the model, the bindings will be updated to display new values, as shown in Figure 12-12.



The first product is Kayak.

The second product is Lifejacket

The first product is Lifejacket.

The second product is Soccer Ball

Figure 12-12. *Manually updating the application model*

It is worth taking a moment to think about what happened when the change detection process ran. Angular reevaluated the expressions on the bindings in the template and updated their values. In turn, the `ngClass` directive and the string interpolation binding reconfigured their host elements by changing their class memberships and displaying new content.

The happens because Angular data bindings are *live*, meaning that the relationship between the expression, the target, and the host element continues to exist after the initial content is displayed to the user and dynamically reflects changes to the application state. This effect is, I admit, much more impressive when you don't have to make changes using the JavaScript console. I explain how Angular allows the user to trigger changes using events and forms in Chapter 14.

Summary

In this chapter, I described the structure of Angular data bindings and showed you how they are used to create relationships between the data in the application and the HTML elements that are displayed to the user. I introduced the property bindings and described how two of the built-in directives—`ngClass` and `ngStyle`—are used. In the next chapter, I explain how more of the built-in directives work.