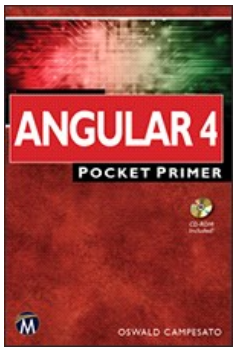

Angular 4 Pocket Primer

by Oswald Campesato

Mercury Learning. (c) 2018. Copying Prohibited.

Reprinted for Krishna Ananthi T, Unisys

Krishna.Ananthi@in.unisys.com

Reprinted with permission as a subscription benefit of **Skillport**,
http://skillport.books24x7.com/

All rights reserved. Reproduction and/or distribution in whole or in part in electronic,paper or other forms without written permission is prohibited.

Chapter 2: UI Controls and User Input

Overview

This chapter contains Angular applications with an assortment of user interface (UI) Controls and examples for handling user interaction, such as user input and mouse events. Keep in mind that the code samples in this chapter render UI Controls using standard HTML syntax instead of using functionality that is specific to Angular. In addition, the last section in this chapter contains links to toolkits that provide Angular UI components.

The first part of this chapter briefly discusses basic debugging techniques that you can use in any Angular application. This section uses the button-related code sample in Chapter 1 to illustrate how to use the debugger in Angular.

The second part of this chapter shows you how to manage lists of items, which includes displaying, adding, and deleting items from a list. Note that forms in Angular are deferred until Chapter 4, where you will also learn about `Controls` and `ControlGroup`.

The third section of this chapter contains two examples of displaying a list of user names: the first retrieves user names that are stored as strings in a JavaScript array, and the second retrieves user names that are stored in object literals in a JavaScript array. The third section goes a step further: You will learn how to define a custom user component that contains user-related information (also contained in a JavaScript array).

> **Note** DVD When you copy a project directory from the companion disc, if the node_modules directory is not present, then copy the top-level node_modules directory that has been soft-linked inside that project directory (which is true for most of the sample applications).

Now let's learn how to perform some basic debugging in Angular.

Debugging Angular Code in the Console

This section shows you how to use `ng.probe()` to "step through" the execution of an Angular application, and to find (or update) the values of variables. The information in this section will help you detect simple errors in your applications. In case you're interested, a much more powerful debugger is Augury (a Chrome extension), which is discussed in Chapter 10.

Now launch the Angular application `ButtonClick` in Chapter 1 (Listing 1.8) in Chrome and perform the following steps:

1. Open Chrome Web Inspector.
2. Click the `Elements` tab.
3. Click the `<app-root>` element.

Now click the `Console` tab and enter the following snippet in the console:

```
ng.probe($0)
```

You will see a `DebugElement` that looks something like the following:

```
DebugElement {nativeNode: app-root, parent: null, listeners:
Array[0], providerTokens: Array[1], properties: Map...}
```

Expand the preceding object and peruse its elements. Click the button (at the top of the screen) several times, and then obtain a reference to the instance of the component class with the following code snippet:

```
ng.probe($0).componentInstance
```

The preceding snippet displays the following element:

```
UserInput {clickCount: 3}
```

Modify the value of `clickCount` with the following code snippet:

```
ng.probe($0).componentInstance.clickCount = 7
```

The preceding code snippet changes the value of `clickCount` to 7, which admittedly does not have any practical purpose in this code sample. However, in other Angular applications that contain various input fields and widgets, this functionality could be useful for testing purposes.

The following code snippet provides component-related information:

```
ng.probe($0).injector._depProvider.componentView
```

The preceding snippet displays the following output:

```
AppView {proto: AppProtoView, renderer: DebugDomRenderer,
         viewManager: AppViewManager_, projectableNodes: null,
                              containerAppElement: AppElement...}
ng.probe($0).injector._depProvider.componentView.
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
                                        changeDetector
```

The next code snippet provides additional information:

```
ChangeDetector_UserInput_0 {id: "UserInput_0"",
numberOfPropertyProtoRecords: 2, bindingTargets: Array[1],
directiveIndices: Array[0], strategy: 5...}
```

Experiment with the available elements by expanding them and inspecting their contents. In addition, the following link contains useful information about Chrome development tools:

https://developers.google.com/web/tools/chrome-devtools/ javascript

Now let's create a simple Angular application that shows you how to display a list of strings via the `ngFor` directive, as discussed in the next section.

The `ngFor` Directive in Angular

The code sample in this section displays a hard-coded list of strings via the `*ngFor` directive. This very simple code sample is a starting point from which you can create more complex—and more interesting—Angular applications.

DVD Copy the directory `SimpleList` from the companion disc into a convenient location. Listing 2.1 displays the contents of `app.component.ts` that illustrates how to display a list of items using the `*ngFor` directive in Angular.

Listing 2.1: app.component.ts

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-root',
  template: `<div *ngFor="let item of items">
              {{item}}
            </div>`
})
export class AppComponent {
  items = [];

  constructor() {
    this.items = ['one','two','three','four'];
  }
}
```

Listing 2.1 contains a `Component` annotation that in turn contains the standard `selector` property. The `template` property consists of a `<div>` element. This element contains the `ngFor` directive, which iterates through the `items` array and displays each item in that array. Notice that the `items` array is initialized as an empty array in the `AppComponent` class, and then its value is set to an array of four strings in the `constructor` method.

Launch the application in this section and you will see the following output in a browser session:

one

two

three

four

Now that you understand how to display items in an array, let's take a short digression to learn about the type of Angular code that can keep track of the radio button that users have clicked ("checked"). After that we'll see how to use a `<button>` element to add new user names to a list of users.

Angular and Radio Buttons

DVD Copy the directory `RadioButtons` from the companion disc into a convenient location. Listing 2.2 displays the contents of `app.component.ts` that illustrates how to render a set of radio buttons and keep track of which button users have clicked.

Listing 2.2: app.component.ts

```
import {Component} from '@angular/core';
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
@Component({
  selector: 'app-root',
  template: `
   <h2>{{radioTitle}}</h2>
   <label *ngFor="let item of radioItems">
      <input type="radio" name="options"
             (click)="model.options = item"
             [checked]="item === model.options">
      {{item}}–
   </label>
   <p><button (click)="model.options='option1'">Set Option
                                                  #1</button>
  `
})
export class AppComponent {
  radioTitle = "Radio Buttons in Angular";
  radioItems = ['option1','option2','option3','option4'];
  model = { options: 'option3' };
}
```

Listing 2.2 defines the `AppComponent` component whose `template` property contains three parts: a `<label>` element, an `<input>` element, and a `<button>` element. The `<label>` element contains an `ngFor` directive that displays a set of radio buttons by iterating through the `radioItems` array that is defined in the `AppComponent` class.

By default, the first radio button is highlighted. However, when users click the `<button>` element, the `(click)` attribute of the `<input>` element sets the *current* item to the value of `model.options`, and then the `[checked]` attribute of the `<input>` element sets the *checked* item to the current value of `model.options`. As you can see, the `<input>` element in Listing 2.2 contains functionality that is more compact than using JavaScript to achieve the same results.

Adding Items to a List in Angular

DVD Copy the directory `AddListButton` from the companion disc into a convenient location. Listing 2.3 displays the contents of `app.component.ts` that illustrates how to append strings to an array of items when users click a button.

Listing 2.3: app.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'app-root',
   template: `
       <div>
         <input #fname>
         <button (click)="clickMe(fname.value)">ClickMe
                                                 </button>
         <ul>
           <li *ngFor="let user of users">
             {{user}}
           </li>
         </ul>
       </div>`
})
export class AppComponent {
   users = ["Jane", "Dave", "Tom"];

   clickMe(user) {
      console.log("new user = "+user);
      this.users.push(user);
/*
      // prevent empty user or duplicates
      if(user is non-null) {
        if(user is duplicate) {
          // display alert message
        } else {
          // display alert message
        }
      } else {
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
        // display alert message
      }
*/
   }
}
```

Listing 2.3 contains code that is similar to Listing 2.1 (which displays a list of strings). In addition, the `template` property in Listing 2.3 contains an `<input>` element so that users can enter text. When users click the `<button>` element, the `clickMe()` method is invoked with `fname.value` as a parameter, which is a reference to the text in the `<input>` element.

Notice the use of the `#fname` syntax as an identifier for an element, which in this case is an `<input>` element. Thus, the text that users enter in the `<input>` element is referenced via `fname.value`. The following code snippet provides this functionality:

```
<input #fname>
<button (click)="clickMe(fname.value)">ClickMe</button>
```

The `clickMe()` method in the `AppComponent` component contains a `console.log()` statement to display the user-entered text (which is optional) and then appends that text to the array `user`. The final section in Listing 2.3 consists of a commented out block of pseudocode that prevents users from entering an empty string or a duplicate string. This code block involves "pure" JavaScript, and the actual code is left as an exercise for you.

Deleting Items from a List in Angular

This section enhances the code in the previous section by adding a new `<button>` element next to each list item.

DVD Copy the directory `DelListButton` from the companion disc into a convenient location. Listing 2.4 displays the contents of `app.component.ts` that illustrates how to delete individual elements from an array of items when users click a button that is adjacent to each array item.

Listing 2.4: app.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'app-root',
   template: `
     <div>
       <input #fname>
       <button (click)="clickMe(fname.value)">ClickMe</button>
       <ul>
         <li *ngFor="let user of users">
           <button (click)="deleteMe(user)">Delete</button>
           {{user}}
         </li>
       </ul>
     </div>`
})
export class AppComponent {
  users = ["Jane", "Dave", "Tom"];

  deleteMe(user) {
     console.log("delete user = "+user);
     var index = this.users.indexOf(user);

     if(index >=0 ) {
        this.users.splice(index, 1);
     }
  }
  clickMe(user) {
     console.log("new user = "+user);
     this.users.push(user);
/*
     // prevent empty user or duplicates
     if(user is non-null) {
       if(user is duplicate) {
         // display alert message
       } else {
         // display alert message
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
        }
      } else {
        // display alert message
      }
*/
  }
}
```

Listing 2.4 contains an `ngFor` directive that displays a list of "pairs" of items, where each "pair" consists of a `<button>` element followed by a user in the `users` array.

When users click any `<button>` element, the "associated" user is passed as a parameter to the `deleteMe()` method, which simply deletes that user from the `users` array. The contents of `deleteMe()` is standard JavaScript code for removing an item from an array. You can replace the block of pseudocode in Listing 2.4 with the same code that you added in Listing 2.3 to prevent users from entering an empty string or a duplicate string.

Angular Directives and Child Components

In this section you will see how to create a child component in Angular that you can reference in an Angular application.

DVD Copy the directory `ChildComponent` from the companion disc into a convenient location. Listing 2.5 displays the contents of `app.component.ts` that illustrates how to import a custom component (written by you) in an Angular application.

Listing 2.5: app.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'app-root',
   template: `<div>Goodbye<child-comp></child-comp>World!
                                                    </div>`
})
export class AppComponent {}
```

Listing 2.5 contains a `template` property that consists of a `<div>` element that contains a nested `<child-comp>` element, where the latter is the value of the `selector` property in the child component `ChildComponent`.

Notice that Listing 2.5 does *not* import the `ChildComponent` class: this class is imported in `app.module.ts` in Listing 2.7.

Listing 2.6 displays the contents of `child.component.ts` in the `app` subdirectory.

Listing 2.6: child.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'child-comp',
   template: `<div>Hello World from ChildComponent!</div>`
})
export class ChildComponent{}
```

Listing 2.6 is straightforward: The `template` property specifies a text string that will appear inside the `<child-comp>` element that is nested inside the `<div>` element in Listing 2.5.

Note This is the first code sample in this chapter that involves modifying the default contents of the file `app.module.ts`.

Listing 2.7 displays the modified contents of `app.module.ts`, which *must* import the class `ChildComponent` from `child.component.ts` and also specify `ChildComponent` in the `declarations` property. These additions are shown in bold.

Listing 2.7: app.module.ts

```
import { NgModule }       from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { AppComponent }   from './app.component';
import { ChildComponent } from './child.component';
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, ChildComponent ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

The first detail to notice in Listing 2.7 is the new `import` statement (shown in bold) that imports the `ChildComponent` component from the TypeScript file `child.component.ts`. The second detail is the inclusion of `ChildComponent` (shown in bold) in the `declarations` array.

As you can see, these are fairly straightforward steps for including a child component in an Angular application. With practice you will become familiar with the sequence of steps that are illustrated in this section.

The Constructor and Storing State in Angular

This section contains a code sample that illustrates how to initialize a variable in a constructor and then reference the value of that variable via interpolation in the `template` property.

DVD Copy the directory `StateComponent` from the companion disc into a convenient location. Listing 2.8 displays the contents of `app.component.ts`.

Listing 2.8: app.component.ts

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-root',
  template: '<h3>My name is {{emp.fname}} {{emp.lname}}</h3>'
})
export class AppComponent {
  public emp = {fname:'John',lname:'Smith',city:'San
                                                Francisco'};
  public name = 'John Smith'

  constructor() {
    this.name = 'Jane Edwards'
    this.emp = {fname:'Sarah',lname:'Smith',city:'San
                                               Francisco'};
  }
}
```

Listing 2.8 is almost the same as Listing 2.5: The current code involves the addition of a `constructor()` method that initializes the variable `name` as well as the literal object `emp`. The `emp` variable is shown in bold in the template property and also in two other places inside the `AppComponent` class.

Question: Which name will be displayed when you launch the application?

Answer: The value that is assigned to the `emp` variable in the `constructor`. This behavior is the same as OO-oriented languages such as Java.

Here is the output from launching this application:

My name is Sarah Smith

Keen-eyed readers will notice how we "slipped in" the TypeScript keyword `public` in the declaration of the `emp` and `name` variables. Other possible keywords include `private` and `protected`, which (again) behave the same way that they do in Java. If you are unfamiliar with these keywords, you can find online tutorials that will explain their purpose.

TypeScript supports another handy syntax (for variables), which is discussed in the next section.

Private Arguments in the Constructor: A Shortcut

TypeScript provides a short-hand notation for initializing private variables via a constructor. Consider the following TypeScript code block:

```
class MyStuff {
   private firstName: string;
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
   constructor(firstName: string) {
      this.firstName = firstName;
   }
}
```

A simpler and equivalent TypeScript code block is shown here:

```
class MyStuff {
   constructor(private firstName: string) {
   }
}
```

TypeScript support for the `private` keyword in a constructor is a nice feature: it reduces some boilerplate code and also eliminates a potential source of error (i.e., misspelled variable names).

As another example, the `constructor()` method in the following code snippet populates an `employees` object with data retrieved from an `EmpService` component (defined elsewhere and not important here):

```
constructor(private empService: EmpService) {
   this.employees = this.empService.getEmployees();
}
```

The next section shows you how to use the `*ngIf` directive for conditional logic in Angular applications.

Conditional Logic in Angular

Although previous examples contain a `template` property with a single line of text, Angular enables you to specify multiple lines of text. If you place interpolated variables inside a pair of matching backticks, Angular will replace ("interpolate") the variables with their values.

DVD Copy the directory `IfLogic` from the companion disc into a convenient location. Listing 2.9 displays the contents of `app.component.ts` that illustrates how to use the `*ngIf` directive.

Listing 2.9: app.component.ts

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
     <h3>Hello everyone!</h3>
     <h3>My name is {{emp.fname}} {{emp.lname}}</h3>
     <button (click)="moreInfo()">More Details</button>
     <div *ngIf="showMore === true">
       <h3>I live in {{emp.city}}</h3>
     </div>
    <div (click)="showDiv = !showDiv">Toggle Me</div>
    <div *ngIf="showDiv"
         style="color:white;background-color:blue;
                                    width:25%">Content1</div>
    <div *ngIf="showDiv"
         style="background-color:red; width:25%;">Content2
                                                    </div>
  `
})
export class AppComponent {
  public emp = {fname:'John',lname:'Smith',city:'San
                                              Francisco'};
  public showMore = false;

  moreInfo() {
    this.showMore = true;
  }
}
```

Listing 2.9 contains some new code in the `template` property: a `<button>` element that invokes the method `moreInfo()` when users click the button. After the click event, a `<div>` element with city-related information inside an `<h3>` element is displayed. Notice that this `<div>` element is only displayed when `showMore` is `true`, which is controlled via the `ngIf` directive that checks for the value of `showMore`. The initial value of `showMore` is `false`, but as soon as users click the `<button>` element, the value is set to `true`, and at that point the `<div>` element is displayed.

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

The new code in AppComponent involves a Boolean variable showMore (initially false) and the method moreInfo(), which initializes show-More to true.

Detecting Mouse Positions in Angular Applications

Angular provides support for detecting various mouse events, some of which you have already seen in previous sections (such as click events). The code sample in this section shows you how to detect a mouse position inside a Scalable Vector Graphics (SVG) <svg> element (SVG graphics effects are discussed in Chapter 3).

DVD Copy the directory SVGMouseMove from the companion disc into a convenient location. Listing 2.10 displays the contents of app.component.ts that illustrates how to detect a mousemove event and to display the coordinates of the current mouse position.

Listing 2.10: app.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'app-root',
   template: `<div><mouse-move></mouse-move></div>`
})
class AppComponent {}
```

Listing 2.10 contains a template property that consists of a element that contains a nested <mouse-move> element, where the latter is the value of the selector property in the custom component MouseMove, which is defined in mousemove.ts. In essence, the component AppComponent "delegates" the handling of mousemove events to the MouseMove component, which defines the mouseMove() function in order to handle such events.

Listing 2.11 displays the contents of mousemove.ts that illustrates how to detect a mousemove event and to display the coordinates of the current mouse position.

Listing 2.11: mousemove.ts

```
import {Component} from '@angular/core';

@Component({
 selector: 'mouse-move',
 template: `<svg id="svg" width="600px" height="400px"
             (mousemove)="mouseMove($event)">
           </svg>
          `
})
export class MouseMove{
   mouseMove(event) {
     console.log("Position x: "+event.clientX+" y: "+event.
                                                  clientY);
   }
}
```

Listing 2.11 contains the mouseMove() method whose lone argument event is an object that contains information (such as its location) about the mouse event. The mouseMove() method contains a console.log() statement that simply displays the x-coordinate and the y-coordinate of the location of the mouse click event.

Remember to update the contents of app.module.ts to include the MouseMove class, as shown in Listing 2.12.

Listing 2.12: app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';
import { MouseMove }     from './mousemove';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, MouseMove ],
  bootstrap:    [ AppComponent ]
})
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
export class AppModule { }
```

Listing 2.12 imports the `MouseMove` class and adds this class to the `declarations` property (both of which are shown in bold).

Mouse Events and User Input in Angular

Angular provides support for mouse events, and automatically recognizes the events `click`, `mousedown`, `mousemove`, `mouseover`, `mouseup`, and `mousewheel`.

The following code snippet shows you how to specify a `<button>` element with an event handler in standard HTML:

```
<button onclick="action()">Action</button>
```

The following code snippet shows you how to specify a `<button>` element with an event handler in Angular:

```
<button (click)="action($event)">Action</button>
```

Listing 2.13 displays the contents of `mouseevents.ts` that illustrates how to handle a `mouseover` event in Angular.

Listing 2.13: mouseevents.ts

```
import {Component} from '@angular/core';

@Component({
 selector: 'mouse-events',
 template: `<div>
             <input type="text" #myInput>
             <button (mouseover)="mouseEvent($event,myInput.
                                 value)">Mouse Over</button>
             </div>`
})
export class MouseEvents {
   mouseCount = 0;

   mouseEvent(event, value) {
      ++this.mouseCount;
      console.log("mouse count: "+this.mouseCount);
      console.log(event, value);
   }
}
```

Listing 2.13 contains a `template` property that comprises a `<div>` element, an `<input>` element where users can enter text, and a `<button>` element with a mouse-related event handler called `mouseEvent`. The expression `$event.myInput.value` references the text in the `<input>` element, and this value is passed to the `mouseEvent()` method.

The next portion of Listing 2.13 is the exported class `MouseEvents` that starts with the variable `mouseCount` whose initial value is `0`. The remainder of `MouseEvents` is the `mouseEvent()` method, which increments and displays the value of `mouseCount` during each `mouseover` event and displays the text in the `<input>` element.

Listing 2.14 displays the contents of `app.component.ts` that involves a "generic" `<mouse-events>` element for mouse-related events in Angular. Keep in mind that this element is a custom element (i.e., it's not an Angular element).

Listing 2.14: app.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'app-root',
   template: `<div><mouse-events></mouse-events></div>`
})
export class AppComponent {}
```

Listing 2.14 contains a `template` property that consists of a `<div>` element that contains a nested `<mouse-events>` element, where the latter is the value of the `selector` property in the custom component `MouseEvents`, which is defined in `mouseevents.ts`. In essence, the component `AppComponent` "delegates" the handling of `mouseover` events to the `MouseEvents` component, which defines the `mouseEvent()` function in order to handle such events.

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

To capture user input via a `mouseclick`, replace `(mouseover)` with `(click)` in the `<button>` element (and also the displayed text), as shown here:

```
<button (click)="mouseEvent($event,myInput.value)">Add
                                                   </button>
```

Remember to update the contents of `app.module.ts` to include the `MouseEvents` class, as shown in Listing 2.15.

Listing 2.15: app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';
import { MouseEvents }   from './mouseevents';

@NgModule({
  imports:      [ BrowserModule ],
  declarations: [ AppComponent, MouseEvents ],
  bootstrap:    [ AppComponent ]
})

export class AppModule { }
```

Listing 2.15 is straightforward: It imports the `MouseEvents` class and adds this class to the `declarations` property (shown in bold).

Handling User Input

The code sample in this section shows you how to handle user input and introduces the notion of a *service* in Angular, which is discussed in greater detail in Chapter 5.

As you have already seen, Angular enables you to create a reference to an HTML element, as shown here:

```
<input type="text" #user>
```

The `#user` syntax creates a reference to the `<input>` element that enables you to reference `{{user.value}}` to see its value, or `{{user.type}}` to see the type of the input. Moreover, you can use this reference in the following code block:

```
<p (click)="user.focus()">
  Get the input focus
</p>
<input type="text" #user (keyup)>
{{user.value}}
```

When users click the `<input>` element, the `focus()` method is invoked, and the `(keyup)` property updates the value in the input during the occurrence of a `keyup` event.

DVD Copy the directory `TodoInput` from the companion disc into a convenient location. Listing 2.16 displays the contents of `app.component.ts` that illustrates how to reference a component that appends user input to an array in Angular.

Listing 2.16: app.component.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'app-root',
   template: `<div>
               <todo-input></todo-input>
               <todo-list></todo-list>
             </div>`
})
export class AppComponent {}
```

Listing 2.16 contains a standard `import` statement. The `template` property specifies a `<div>` element that contains placeholders for the `TodoInput` and `TodoList` components.

Listing 2.17 displays the contents of `todoinput.ts` that illustrates how to display an `<input>` field and a `<button>` element to capture user input in Angular.

Listing 2.17: todoinput.ts

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
import {Component}    from '@angular/core';
import {TodoService } from './todoservice';

@Component({
 selector: 'todo-input',
 template: `
   <div>
     <input type="text" #myInput>
     <button (click)="mouseEvent(myInput.value)">Add Name
                                                  </button>
   </div>`
})
export class TodoInput{
   constructor(public todoService:TodoService) {}

   mouseEvent(value) {
      if((value != null) && (value.length > 0)) {
        this.todoService.todos.push(value);
        console.log("todos: "+this.todoService.todos);
      } else {
         console.log("value must be non-null");
      }
   }
}
```

Listing 2.17 contains a `template` property that consists of a `<div>` element that contains an `<input>` element for user input, followed by a `<button>` element for handling mouse click events.

The `TodoInput` class defines an empty constructor that also initializes an instance of the custom `TodoService` that is imported at the top of the file. This instance contains an array `todos` that is updated with new to-do items when users click the `<button>` element, provided that the new to-do item is not the empty string.

Listing 2.18 displays the contents of `todolist.ts` that keeps track of the items in a to-do list.

Listing 2.18: todolist.ts

```
import {Component}   from '@angular/core';
import {TodoService} from './todoservice';

@Component({
 selector: 'todo-list',
 template: `<div>
             <ul>
               <li *ngFor="let todo of todoService.todos">
                 {{todo}}
               </li>
             </ul>
            </div>`
})
export class TodoList {
   constructor(public todoService:TodoService) {}
}
```

Listing 2.18 contains a `template` property whose contents are similar to the contents of the `template` property in Listing 2.3, along with an empty constructor that initializes an instance of the `TodoService` custom component. This instance is used in the `template` property to iterate through the elements in the `todos` array.

Listing 2.19 displays the contents of `todoservice.ts` that keeps track of the to-do list.

Listing 2.19: todoservice.ts

```
export class TodoService {
   todos = [];
}
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

Listing 2.19 contains a `todos` array that is updated with new to-do items when users click the `<button>` element in the root component.

Finally, update the contents of `app.module.ts` to include the class shown in bold in Listing 2.20.

Listing 2.20: app.module.ts

```
import { NgModule }      from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent }  from './app.component';
import { TodoInput }     from './todoinput';
import { TodoList }      from './todolist';
import { TodoService }   from './todoservice';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ TodoService ],
  declarations: [ AppComponent, TodoInput, TodoList ],
  bootstrap:    [ AppComponent ]
})
export class AppModule { }
```

Listing 2.20 imports three to-do-related classes and adds them to the `providers` property and the `declarations` property (shown in bold).

The moduleId and templateUrl Properties in Angular

An earlier section showed you how to use the `ngFor` directive in a `template` property to iterate through a list of items. Angular also supports the `templateUrl` property, which means that you can move the code in the `template` property to a separate file, and then reference the name of that file as the value of the `templateUrl` property.

For example, suppose that the code in the `template` property is placed in the file `itemdetails.html`. If this file is in the same directory as `itemsapp.ts` (shown below), you must include the property `moduleId` to indicate the subdirectory where the file `itemdetails.html` is located. If you do not specify the `moduleId` property, then Angular assumes that `itemdetails.html` is in the same directory as the top-level Web page that launches the top-level Angular component.

Note Use the moduleId property to specify relative paths for files that are specified in the templateUrl property.

Listing 2.21 displays the contents of `itemsapp.ts` that references the file `itemdetails.html`, which contains the code for iterating through a list of items. The `moduleId` property indicates that this file is in the same directory as `itemsapp.ts`.

Listing 2.21: itemsapp.ts

```
import {Component} from '@angular/core';

@Component({
  selector: 'app-root',
  moduleId: 'app/itemdetails',
  templateUrl: 'itemdetails.html'
})
export class AppComponent {
  items = [];

  constructor() {
    this.items = ['one','two','three','four'];
  }
}
```

Listing 2.21 contains a `templateUrl` property that references the file `itemdetails.html` with Angular code for displaying the contents of the items array.

Listing 2.22 displays the contents of `itemdetails.html` that contains the code for displaying the contents of the `items` array, which is initialized in the constructor in Listing 2.21.

Listing 2.22: itemdetails.html

```
<div *ngFor="let item of items">
  {{item}}
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
</div>
```

You could easily move the contents of Listing 2.22 into a `template` property in Listing 2.21; the only reason for including this file is to show you how to use the `templateUrl` property.

Working with Custom Classes in Angular

You have already seen an example of a custom TypeScript class that represents a user. This section shows you how to work with an array of instances of a custom TypeScript class.

Listing 2.23 displays the contents of `newuser.ts` that illustrates how to create a custom TypeScript class that represents a user.

Listing 2.23: newuser.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'new-user',
   template: '',
})
export class User {
   fname:string;

   constructor(fname:string) {
      this.fname = fname;
   }
}
```

For ease of illustration, Listing 2.23 defines a very simple `User` custom component that only keeps track of the `fname` property for a single user.

Listing 2.24 displays the contents of `app.component.ts` that uses the `User` custom component to populate an array with a set of users represented by instances of the `User` class and then display user-related information in a list. The `User` class is obviously more useful when you include other user-related properties in addition to the first name property.

Listing 2.24: app.component.ts

```
import {Component} from '@angular/core';
import {User}      from './newuser';
@Component({
   selector: 'app-root',
   template: `
      <div>
        <input #fname>
        <button (click)="clickMe(fname.value)">Add User</
                                                    button>
        <ul>
          <li *ngFor="let user of users"
                                (click)="onSelect(user)">
            {{user.fname}}
          </li>
        </ul>
      </div>`
})
export class AppComponent {
   newUser:User;

   users = [
             new User("Jane"),
             new User("Dave"),
             new User("Tom")
           ];

   clickMe(user) {
      console.log("creating new user: "+user);
      this.newUser = new User(user);
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
      this.users.push(this.newUser);
   }

   onSelect(user) {
      console.log("Selected user: "+JSON.stringify(user));
      var index = this.users.indexOf(user);
      this.users.splice(index,1);
   }
}
```

Listing 2.24 contains a `template` property that iterates through the list of `User` instances in the `users` array and displays the name contained in each instance. Notice how the `users` array is initialized in the `AppComponent` component: Three `User` instances are created from the `User` custom component that is defined in `newuser.ts`.

Listing 2.24 also contains a `clickMe()` method that is invoked when users click the `<button>` element, after which a new user is appended to the `users` array. Finally, Listing 2.24 contains an `onSelect()` method that is invoked when users select a different item in the list of users.

Click Events in Multiple Components

An Angular application can contain multiple components, each of which can declare event handlers with the same name. This section contains an example that shows you the order in which click events are handled in an Angular application.

DVD Copy the directory `ClickItems` from the companion disc into a convenient location. Listing 2.25 displays the contents of `app.component.ts` that declares an `onClick()` event handler for each item in a list of items.

Listing 2.25: app.component.ts

```
import {Component} from '@angular/core';
import {ClickItem} from './clickitem';

@Component({
   selector: 'app-root',
   styles: [`li { display: inline; }`],
   template: `
    <div>
      <ul>
       <li><img (click)="onClick()"
             height="100" src="src/sample1.png">
                                                  </li>
       <li><img (click)="onClick()"
             width="100" height="100" src="src/sample2.png">
                                                  </li>
       <li><img (click)="onClick()"
             width="100" height="100" src="src/sample3.png">
                                                  </li>
       </ul>
     </div>
     `
})
export class AppComponent {
  onClick() {
    console.log("app.component.ts: you clicked me");
  }
}
```

The `template` property in Listing 2.25 displays an unordered list in which each item is a clickable `PNG`-based image. When users click one of the images, the `onClick()` method is invoked that simply displays a message via `console.log()`.

Listing 2.26 displays the contents of `clickitem.ts` that declares an `onClick()` event handler for each item in a list of items.

Listing 2.26: clickitem.ts

```
import {Component} from '@angular/core';

@Component({
   selector: 'cclick',
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
    styleUrl: [` li { inline: block } `],
    template: `
      <div>
       <ul>
        <li><img (click)="onClick(100)"
                 width="100" height="100" src="src/sample1.
                                                  png"></li>
        <li><img (click)="onClick(200)"
                 width="100" height="100" src="src/sample2.
                                                  png"></li>
        <li><img (click)="onClick(300)"
                 width="100" height="100" src="src/sample3.
                                                  png"></li>
       </ul>
      </div>
    `
})
export class ClickItem {
  onClick(id) {
    console.log("clickitem.ts clicked: "+id));
  }
}
```

Listing 2.26 is similar to Listing 2.25 in terms of functionality. Launch the application and click in various locations in your browser, and observe the different messages that are displayed in Chrome Web Inspector.

Working with `@Input`, `@Output`, and `EventEmitter`

Angular supports the `@Input` and `@Output` annotations to pass values between components. The `@Input` annotation is for variables that receive values from a parent component, whereas the `@Output` annotation sends (or "emits") data from a component to its parent component when the value of the given variable is modified.

The output from this code sample is anticlimactic. However, the purpose of this code sample is to draw your attention to some of the nonintuitive code snippets (especially in `app.module.ts`). Moreover, this code sample works correctly for version 2.1.5 of the TypeScript compiler, but it's possible that future versions will require modifications to the code (so keep this point in mind).

DVD Now copy the directory `ParentChildEmitters` from the companion disc into a convenient location. Listing 2.27 displays the contents of `app.component.ts` that shows you how to update the value of a property of a child component from a parent component.

Listing 2.27: app.component.ts

```
import {Component}      from '@angular/core';
import {EventEmitter}   from '@angular/core';
import {ChildComponent} from './childcomponent';

@Component({
  selector: 'app-root',
  providers: [ChildComponent],
  template: `
     <div>
      <child-comp [childValue]="parentValue"
        (childValueChange)="reportValueChange($event)">
      </child-comp>
     </div>
    `
})
export class AppComponent {
  public parentValue:number = 77;

  constructor() {
    console.log("constructor parentValue = "+this.
                                              parentValue);
  }

  reportValueChange(event) {
    console.log(event);
  }
```


Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

```
}
```

The `template` property in Listing 2.27 has a top-level `<div>` element that contains a `<child-comp>` element that has two attributes, as shown here:

```
<child-comp [childValue]="parentValue"
            (childValueChange)="reportValueChange($event)">
</child-comp>
```

The `[childValue]` attribute assigns the value of `parentValue` to the value of `childValue`. Notice that the variable `parentValue` is defined in `AppComponent`, whereas the variable `childValue` is defined in `ChildComponent`. *This is how to pass a value from a parent component to a child component*.

Next, the `childValueChange` attribute is assigned the value that is returned from `ChildComponent` to the current ("parent") component. Keep in mind that the attribute `childValueChange` is updated only when the value of `childValue` (in the child component) is modified. *This is how to pass a value from a child component to a parent component*.

Keep in mind the following point: The child component *must* define a variable of type `EventEmitter` (such as `childValueChange`) to "emit" a modified value from the child component to the parent component.

The next portion of Listing 2.27 is a simple constructor, followed by the method `reportValueChange,` which contains a `console.log()` statement.

Listing 2.28 displays the contents of `childcomponent.ts` that shows you how to update the value of a property of a child component from a parent component.

Listing 2.28: childcomponent.ts

```
import {Component}    from '@angular/core';
import {Input}        from '@angular/core';
import {Output}       from '@angular/core';
import {EventEmitter} from '@angular/core';

@Component({
  selector: 'child-comp',
  template: `
     <button (click)="decrement();">Subtract</button>
     <input type="text" [value]="childValue">
     <button (click)="increment();">Add</button>
   `
})
export class ChildComponent {
  @Input() childValue:number = 3;
  @Output() childValueChange = new EventEmitter();

  constructor() {
    console.log("constructor childValue = "+this.childValue);
  }
  increment() {
    this.childValue++;

    this.childValueChange.emit({
      value: this.childValue
    })
  }
  decrement() {
    this.childValue--;

    this.childValueChange.emit({
      value: this.childValue
    })
  }
}
```

Listing 2.28 contains a `template` property that has a "decrement" `<button>` element, an `<input>` field where users can enter a number, and also an "increment" `<button>` element. The first `<button>` element increments the value `<input>` field, whereas the second `<button>` element decrements the value.

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

The exported class `ChildComponent` contains the numeric variable `childValue`, which is decorated via `@Input()`, and whose value is set by the parent.

As you can see, the methods `increment()` and `decrement()` increase and decrease the value of `childValue`, respectively. In both cases, the modified value of `childValue` is then "emitted" back to the parent with this code block:

```
this.childValueChange.emit({
   value: this.childValue
})
```

Update the contents of `app.module.ts` as shown in Listing 2.29, *which is different from the code in previous examples in this chapter*.

Listing 2.29: app.module.ts

```
import { NgModule }       from '@angular/core';
import {CUSTOM_ELEMENTS_SCHEMA} from '@angular/core';
import { BrowserModule }  from '@angular/platform-browser';
import { AppComponent }   from './app.component';
import { ChildComponent } from './childcomponent';

@NgModule({
  imports:      [ BrowserModule ],
  providers:    [ ChildComponent ],
  declarations: [ AppComponent ],
  bootstrap:    [ AppComponent ],
  schemas:      [CUSTOM_ELEMENTS_SCHEMA]
})
export class AppModule { }
```

If you specify `ChildComponent` in the `declarations` property instead of the `providers` property, you will probably see this error message:

```
"Can't bind to <child-comp> since it isn't a known native
property"
```

When you launch the Angular application in this section, the value that is displayed in the `<input>` element is `77`, which is the value in the parent component, and *not* the value that is assigned in the child component (which is `3`).

Presentational Components

Presentational components receive data as input and generate views as outputs (so they do not maintain the application state). Consider the following component:

```
@Component({
  selector: 'student-info',
  template: `<h2>{{studentDetails?.status}}</h2>
    <div class="container">
      <table class="table">
        <tbody>
        <tr *ngFor="let student of students">
            <td>{{student.fname}}</td>
            <td>{{student.lname}}</td>
        </tr>
        </tbody>
    </table>
</div>`
})
export class StudentDetailsComponent {
   @Input()
   studentDetails:StudentDetails;
}
```

The `StudentDetailsComponent` component has primarily presentational responsibilities. The component receives input data and displays that on the screen. As a result, this component is reusable.

By contrast, application-specific components (also called "smart" components) are tightly coupled to a specific Angular application. Thus, a smart component would have a presentation component (but not the converse).

Because data is passed to this component synchronously (not via an `Observable`), the data might not be present initially, which is the reason for including the so-called Elvis operator (the "?" in the template).

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

Styling Elements and Shadow DOM in Angular

Angular supports Cascading Style Sheets (CSS) encapsulation, which means that CSS selectors will only match elements that are defined in the same custom component. This encapsulation is available because of `ShadowDOM` emulation in Angular. In particular, this involves an `import` statement to import `ViewEncapsulation`, and also specifying one of the following values:

- ViewEncapsulation.Emulated
- ViewEncapsulations.Native
- ViewEncapsulation.None

The default for components is `ViewEncapsulation.Emulated`, which outputs namespaces of our class next to our styles and inherits global styles. By contrast, `ViewEncapsulations.Native` uses the Native `ShadowDOM` (which is not supported in all browsers), and loses global styles. Finally, `ViewEncapsulation.None` removes all style encapsulation in a component.

```
styles: [`
    #mydiv {
      font-size: 1rem;
      line-height: 1.25;
      color: #999;
      background-color: #ffcccc;
    }
    :global(body) {
      color: #666;
      background-color: #ccccff;
    }
  `],
```

DVD Copy the directory `ViewEncapsulation` from the companion disc into a convenient location. Listing 2.30 displays the contents of `app.component.ts` that contains a CSS selector that matches the `<button>` element in `app.component.ts` but not the `<button>` element in `index.html`.

Listing 2.30: app.component.ts

```
import {Component} from '@angular/core';
import {ViewEncapsulation} from '@angular/core';

@Component({
  selector: 'app-root',
  encapsulation: ViewEncapsulation.Native,
  styles: [`
    .button { background-color: red; }
  `],
  template: `
    <button class="button"<Click in Component</button>
  `,
})
class AppComponent {}
```

Listing 2.30 contains two `import` statements and an encapsulation property whose value is set to `ViewEncapsulation.Native`. Next, the `styles` property specifies the color `red` for the `<button>` element that is declared in the template section.

Now modify the `<body>` element in `index.html` by adding a `<button>` element as shown here:

```
<body>
  <app-root>Loading...</app-root>
  <button class="button">Click in index.html</button>
</body>
```

Launch the Angular application and you will see a red `<button>` element in the component and a dark gray `<button>` element in the HTML page.

Angular UI Components

The code samples in this chapter show you how to use HTML widgets in Angular applications. However, you can also use Angular UI components that are "wrappers" for HTML widgets. The official Angular website contains an extensive collection of UI components:

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited

https://angular.io/resources/#!#UI%20Components

The preceding website contains various links to other toolkits, such as `ng-bootstrap` (native Angular 2 directives for Bootstrap) and Angular Material 2 (Material Design components for Angular 2).

The following website contains an extensive collection of Angular UI components (e.g., table, tree, menu, and chart):

https://github.com/brillout/awesome-angular-components

Keep in mind that the UI components in the preceding link are for Angular 2, so it's a good idea to test them in the latest version of Angular. In addition, Chapter 5 contains form-related UI components that do work in version 4 of Angular.

You can also perform an Internet search for other open source (or commercial) alternatives to determine which one suits your needs.

New Features in Angular

Some new features in Angular include support for if/else logic in the `ngIf` directive and the `NgComponentOutlet` directive.

The `ngIf` directive conditionally includes a template based on the value of an expression. Next, `ngIf` evaluates the expression and then renders the `then` or `else` template in its place when expression is `truthy` or `falsy` respectively. Typically the `then` template is the inline template of `ngIf` unless bound to a different value, and the `else` template is blank unless it is bound. More information is located here:

https://angular.io/docs/ts/latest/api/common/index/NgIfdirective. html

The `NgComponentOutlet` directive is an experimental directive that provides a declarative approach for creating dynamic components, an example of which is shown here:

```
@Component({
  selector: 'app-root',
  template: `
    <h1>Angular ngComponentOutlet</h1>
    <ng-container *ngComponentOutlet="myComponent">
                                              </ng-container>
    <button (click)="doSomething()">Toggle Component</button>
  `,
})
```

As you can probably surmise, the `ng-container` directive is a logical container for "grouping" nodes. In the earlier code block, the `ng-container` directive has an `NgComponentOutlet` whose value is of type `Input`, which in turn references a custom component. Next, make sure you add dynamic components to the `entryComponents` section of `ngModule`, as shown here:

```
@NgModule({
  ...,
  entryComponents: [MyComponent1, MyComponent2],
  ...
})
```

The `NgComponentOutlet` directive supports additional options that are described here:

https://angular.io/docs/ts/latest/api/common/index/NgComponentOutlet- directive.html

Summary

This chapter showed you how to use UI Controls in Angular applications. You saw how to render buttons, how to render lists of names, and how to add and delete names from those lists. You also learned about conditional logic and how to create child components.

Then you saw how to handle mouse-related events, such as `mousemove` events. Next you learned about communicating between parent and child components, followed by a discussion of presentational components. You also learned how to specify different types of CSS encapsulation in an Angular application. Finally, you were briefly introduced to some new UI-related features in Angular.

Reprinted for UNISYS/298520, Unisys Mercury Learning, Mercury Learning and Information LLC (c) 2018, Copying Prohibited