DISPLAYING DATA

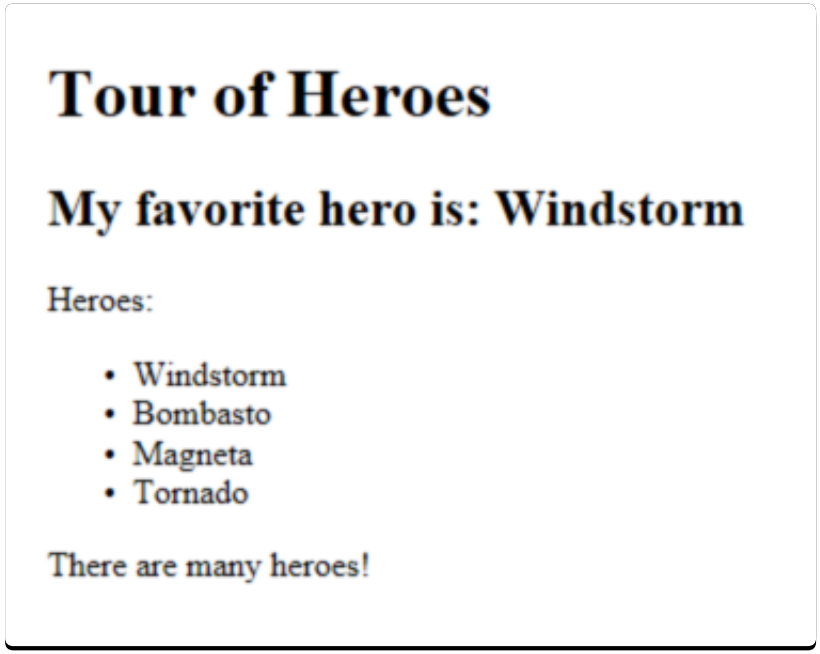
Developer Preview Only - some details may change

Interpolation and other forms of property binding help us show app data in the UI.

We typically display data in Angular by binding controls in an HTML template to properties of an Angular component.

In this chapter, we'll create a component with a list of heroes. Each hero has a name. We'll display the list of hero names and conditionally show a selected hero in a detail area below the list.

The final UI looks like this:



Run the live example

Showing component properties with interpolation

The easiest way to display a component property is to bind the property name through interpolation. With interpolation, we put the property name in the view template, enclosed in double curly braces: `{{myHero}}`.

Let's build a small illustrative example together.

Create a new project folder (`displaying-data`) and follow the steps in the QuickStart.

Then modify the `app.component.ts` file by changing the template and the body of the component. When we're done, it

should look like this:

app/app.component.ts

```
1. import {Component} from 'angular2/core';
2.
3. @Component({
4.   selector: 'my-app',
5.   template: `
6.     <h1>{{title}}</h1>
7.     <h2>My favorite hero is: {{myHero}}</h2>
8.     `
9. })
10. export class AppComponent {
11.   title = 'Tour of Heroes';
12.   myHero = 'Windstorm';
13. }
```

We added two properties to the formerly empty component: `title` and `myHero`.

Our revised template displays the two component properties using double curly brace interpolation:

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero}}</h2>
  `
```

> The template is a multi-line string within ECMAScript 2015 backticks (`). The backtick (`) — which is *not* the same character as a single quote (') — has many nice features. The feature we're exploiting here is the ability to compose the string over several lines, which makes for much more readable HTML.

Angular automatically pulls the value of the `title` and `myHero` properties from the component and inserts those values into the browser. Angular updates the display when these properties change.

> More precisely, the redisplay occurs after some kind of asynchronous event related to the view such as a keystroke, a timer completion, or an async `XHR` response. We don't have those in this sample. But then the properties aren't changing on their own either. For the moment we must operate on faith.

Notice that we haven't called **new** to create an instance of the `AppComponent` class. Angular is creating an instance for us. How?

Notice the CSS `selector` in the `@Component` decorator that specifies an element named "my-app". Remember back in QuickStart that we added the `<my-app>` element to the body of our `index.html`

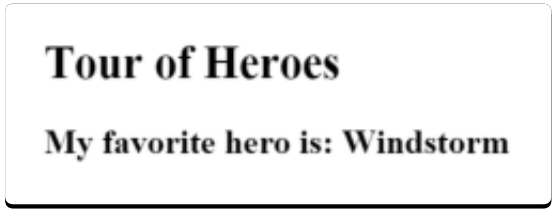
```
<body>
  <my-app>loading...</my-app>
</body>
```

When we bootstrap with the `AppComponent` class (see `main.ts`), Angular looks for a `<my-app>` in the `index.html`, finds it, instantiates an instance of `AppComponent`, and renders it inside the `<my-app>` tag.

We're ready to see changes in a running app by firing up the npm script that both compiles and serves our applications while watching for changes.

```
npm start
```

We should see the title and hero name:



Let's review some of the choices we made and consider alternatives.

Template inline or template file?

We can store our component's template in one of two places. We can define it *inline* using the `template` property, as we do here. Or we can define the template in a separate HTML file and link to it in the component metadata using the `@Component` decorator's `templateUrl` property.

The choice between inline and separate HTML is a matter of taste, circumstances, and organization policy. Here we're using inline HTML because the template is small, and the demo is simpler without the HTML file.

In either style, the template data bindings have the same access to the component's properties.

Constructor or variable initialization?

We initialized our component properties using variable assignment. This is a wonderfully concise and compact technique.

Some folks prefer to declare the properties and initialize them within a constructor like this:

```
export class AppCtorComponent {
  title: string;
  myHero: string;

  constructor() {
    this.title = 'Tour of Heroes';
    this.myHero = 'Windstorm';
  }
}
```

That's fine too. The choice is a matter of taste and organization policy. We'll adopt the more terse "variable assignment" style in this chapter simply because there will be less code to read.

Showing an array property with NgFor

We want to display a list of heroes. We begin by adding a mock heroes name array to the component, just above `myHero`,

and redefine `myHero` to be the first name in the array.

app/app.component.ts (class)

```
export class AppComponent {
  title = 'Tour of Heroes';
  heroes = ['Windstorm', 'Bombasto', 'Magneta', 'Tornado'];
  myHero = this.heroes[0];
}
```

Now we use the Angular `NgFor` "repeater" directive in the template to display each item in the `heroes` list.

app/app.component.ts (template)

```
  template: `
    <h1>{{title}}</h1>
    <h2>My favorite hero is: {{myHero}}</h2>
    <p>Heroes:</p>
    <ul>
      <li *ngFor="#hero of heroes">
        {{ hero }}
      </li>
    </ul>
  `
```

Our presentation is the familiar HTML unordered list with `<ul>` and `<li>` tags. Let's focus on the `<li>` tag.

```
<li *ngFor="#hero of heroes">
  {{ hero }}
</li>
```

We added a somewhat mysterious `*ngFor` to the `<li>` element. That's the Angular "repeater" directive. Its presence on the `<li>` tag marks that `<li>` element (and its children) as the "repeater template".

> Don't forget the leading asterisk (*) in `*ngFor`. It is an essential part of the syntax. Learn more about this and `NgFor` in the Template Syntax chapter.

Notice the `#hero` in the `NgFor` double-quoted instruction. The `#hero` is a local template variable declaration. The `#` prefix declares a local variable name named `hero`.

Angular duplicates the `<li>` for each item in the list, setting the `hero` variable to the item (the hero) in the current iteration. Angular uses that variable as the context for the interpolation in the double curly braces.

> We happened to give `NgFor` an array to display. In fact, `NgFor` can repeat items for any iterable object.

Assuming we're still running under the `npm start` command, we should see heroes appearing in an unordered list.

Creating a class for the data

We are defining our data directly inside our component. That's fine for a demo but certainly isn't a best practice. It's not even a good practice. Although we won't do anything about that in this chapter, we'll make a mental note to fix this down the road.

At the moment, we're binding to an array of strings. We do that occasionally in real applications, but most of the time we're displaying objects — potentially instances of classes.

Let's turn our array of hero names into an array of `Hero` objects. For that we'll need a `Hero` class.

Create a new file in the `app/` folder called `hero.ts` with the following short bit of code.

app/hero.ts

```
export class Hero {
  constructor(
    public id:number,
    public name:string) { }
}
```

We've defined a class with a constructor and two properties: `id` and `name`.

It might not look like we have properties, but we do. We're taking advantage of a TypeScript shortcut in our declaration of the constructor parameters.

Consider the first parameter:

```
public id:number,
```

That brief syntax does a lot:

- declares a constructor parameter and its type
- declares a public property of the same name
- initializes that property with the corresponding argument when we "new" an instance of the class

Using the Hero class

Let's redefine the `heroes` property in our component to return an array of these Hero objects and also set the `myHero` property with the first of these mock heroes.

app.component.ts (excerpt)

```
heroes = [
  new Hero(1, 'Windstorm'),
  new Hero(13, 'Bombasto'),
  new Hero(15, 'Magneta'),
  new Hero(20, 'Tornado')
];
myHero = this.heroes[0];
```

We'll have to update the template. At the moment it displays the entire `hero` object, which used to be a string value. Let's fix that so we interpolate the `hero.name` property.

app.component.ts (template)

```
template: `
  <h1>{{title}}</h1>
  <h2>My favorite hero is: {{myHero.name}}</h2>
  <p>Heroes:</p>
  <ul>
    <li *ngFor="#hero of heroes">
      {{ hero.name }}
```

```
      </li>
    </ul>
  `
```

Our display looks the same, but now we know much better what a hero really is.

Conditional display with NgIf

Sometimes the app should display a view or a portion of a view only under specific circumstances.

In our example, we'd like to display a message if we have a large number of heroes — say, more than 3.

The Angular `NgIf` directive inserts or removes an element based on a truthy/falsey condition. We can see it in action by adding the following paragraph at the bottom of the template:

```
<p *ngIf="heroes.length > 3">There are many heroes!</p>
```

> Don't forget the leading asterisk (*) in `*ngIf`. It is an essential part of the syntax. Learn more about this and `NgIf` in the Template Syntax chapter.

The template expression inside the double quotes looks much like JavaScript and it *is* much like JavaScript. When the component's list of heroes has more than 3 items, Angular adds the paragraph to the DOM and the message appears. If

there are 3 or fewer items, Angular omits the paragraph, so no message appears.

> Angular isn't showing and hiding the message. It is adding and removing the paragraph element from the DOM. That hardly matters here. It would matter a great deal from a performance perspective if we were conditionally including or excluding a big chunk of HTML with many data bindings.

Try it out. Because the array has four items, the message should appear. Go back into `app.component.ts` and delete or comment out one of the elements from the hero array. The browser should refresh automatically and the message should disappear.

Summary

Now we know how to use:

- **interpolation** with double curly braces to display a component property
- `NgFor` to display a list of items
- a TypeScript class to shape the **model data** for our component and display properties of that model
- `NgIf` to conditionally display a chunk of HTML based on a boolean expression

Here's our final code:

```
1. import {Component} from 'angular2/core';
2. import {Hero} from './hero'
```

```
3.
4. @Component({
5.   selector: 'my-app',
6.   template: `
7.   <h1>{{title}}</h1>
8.   <h2>My favorite hero is: {{myHero.name}}</h2>
9.   <p>Heroes:</p>
10.  <ul>
11.    <li *ngFor="#hero of heroes">
12.      {{ hero.name }}
13.      </li>
14.  </ul>
15.  <p *ngIf="heroes.length > 3">There are many heroes!</p>
16. `
17. })
18.
19. export class AppComponent {
20.   title = 'Tour of Heroes';
21.   heroes = [
22.     new Hero(1, 'Windstorm'),
23.     new Hero(13, 'Bombasto'),
24.     new Hero(15, 'Magneta'),
25.     new Hero(20, 'Tornado')
26.   ];
27.   myHero = this.heroes[0];
28. }
```

Next Step

User Input