Creating a custom directive

Directive decorator

We'll call our directive `appCard` and we'll attach it to the card block like so:

```
<div class="card card-block" appCard>...</div>
```

We create directives by annotating a class with the `@Directive` decorator.

Lets create a class called `CardHoverDirective` and use the `@Directive` decorator to associate this class with our attribute `appCard`, like so:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
  selector:"[appCard]"
})
class CardHoverDirective { }
```

Attribute selector

The above code is very similar to what we would write if this was a component, the first striking difference is that *the selector is wrapped with `[]`*.

To understand why we do this we first need to understand that the selector attribute uses *CSS matching rules* to match a component/directive to a HTML element.

In CSS to match to a specific element we would just type in the name of the element, so `input {…}` or `p {…}`.

This is why previously when we defined the selector in the `@Component` directive we just wrote the *name*of the element, which matches onto an element of the same name.

If we wrote the selector as `.appCard`, like so:

```
import { Directive } from '@angular/core';
.
.
.
@Directive({
```

```
  selector:".appCard"
})
class CardHoverDirective { }
```

Then this would associate the directive with any element that has a *class* of ccCardHover, like so:

```
<div class="card card-block appCard">...</div>
```

We want to associate the directive to an element which has a certain attribute.

To do that in CSS we wrap the name of the attribute with `[]`, and this is why the selector is called `[appCard]`.

Directive constructor

The next thing we do is add a constructor to our directive, like so:

```
import { ElementRef } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef) {
  }
}
```

When the directive gets created Angular can inject an instance of something called `ElementRef` into its constructor.

The `ElementRef` gives the directive *direct access* to the DOM element upon which it's attached.

Let's use it to change the background color of our card to gray.

`ElementRef` itself is a wrapper for the actual DOM element which we can access via the property `nativeElement`, like so:

```
el.nativeElement.style.backgroundColor = "gray";
```

This however assumes that our application will always be running in the environment of a browser.

Angular has been built from the ground up to work in a number of different environments, including server side via node and on a native mobile device. So the Angular team has provided a *platform independent* way of setting properties on our elements via something called a `Renderer`.

Listing 1. script.ts

```
import { Renderer } from '@angular/core';
.
.
.
class CardHoverDirective {
      constructor(private elem: ElementRef,
        private renderer: Renderer2) {
        const el = this.elem.nativeElement;
        el.style.background = 'blue';
        //renderer.setStyle(el, 'background', 'gray');
    }
 }
```

(1) We use *Dependency Injection* (DI) to inject the renderer into our directives constructor.

(2) Instead of setting the background color directly via the DOM element we do it by going through the renderer.

Running the application now show this: