DIRECTIVE DECORATOR

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

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

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

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

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

Therefore, 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 `.ccCardHover`, like so:

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

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

```
<div class="card card-block ccCardHover">...</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 `[ccCardHover]`.

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`.

```
import { Renderer } from '@angular/core';
.
.
.
class CardHoverDirective {
  constructor(private el: ElementRef,
              private renderer: Renderer) {
    renderer.setElementStyle(el.nativeElement, 'backgroundColor', 'gray');
  }
}
```

SUMMARY

We create a directive by decorating a class with the `@Directive` decorator.

The convention is to associate a directive to an element via an ATTRIBUTE SELECTOR, that is the name of the attribute wrapped in `[]`.

We can inject a reference to the element the directive is associated with to the constructor of the directive. Then via a `renderer` we can interact with and change certain properties of that element.

The above is a very basic example of a custom directive, in the next lecture we'll show you how you can detect when the user hovers over the card and a BETTER way of interacting with the host element