COMPONENT INPUTS

Inputs specify the parameters we expect our component to receive. To designate an input, we use the `@Input()` decoration on a component class property.

When we specify that a Component takes an input, it is expected that the definition class will have an instance variable that will receive the value. For example, say we have the following code:

```
import { Component, Input } from '@angular/core';
@Component({
selector: 'my-component',
})
class MyComponent {
        @Input() name: string;
        @Input() age: number;
}
```

The `name` and `age` inputs map to the `name` and `age` properties on instances of the `MyComponent` class.

If we need to use two different names for the attribute and the property, we could for example write `@Input('firstname') name: String;`. But the Angular Style Guide suggests avoiding this.

If we want to use MyComponent from another template, we write something like: <my-component [name]="myName" [age]="myAge"></my-component>.

Notice that the attribute name matches the input name, which in turn matches the MyComponent property name. However, these don't always have to match.

For instance, say we wanted our attribute key and instance property to differ. That is, we want to use our component like this:

```
<my-component [shortName]="myName" [oldAge]="myAge"></my-component>
```

To do this, we would change the format of the string in the `inputs` option:

```
@Component({
      selector: 'my-component'
})
class MyComponent {
      @Input('shortName') name: string;
      @Input('oldAge') age: number;
}
```

The property name (`name`, `age`) represent how that incoming property will be visible ("bound") in the controller.

The **`@Input`** argument (`shortName`, `oldAge`) configures how the property is visible to the "outside world".

COMPONENT OUTPUTS

When you want to send data from your component to the outside world, you use output bindings.

Let's say a component we're writing has a button and we need to do something when that button is clicked.

The way to do this is by binding the click output of the button to a method declared on our component's controller. You do that using the `(output)="action"` notation.

Here's an example where we keep a counter and increment (or decrement) based on which button is pressed:

```
@Component({
   selector: 'counter',
   template: `
   {{ value }}
   <button (click)="increase()">Increase</button>
   <button (click)="decrease()">Decrease</button>
   `
})
class Counter {
   value: number;

constructor() {
   this.value = 1;
}

increase() {
   this.value = this.value + 1;
   return false;
}
decrease() {
   this.value = this.value - 1;
   return false;
 }
}
```

In this example we're saying that every time the first button is clicked, we want the `increase()` method on our controller to be invoked. And, similarly, when the second button is clicked, we want to call the `decrease()` method.

The parentheses attribute syntax looks like this: `(output)="action"`. In this case, the output we're listening for is the `click` event on this button. There are many other built-in events we can listen to such as: `mousedown`, `mousemove`, `dbl-click`, etc.

In this example, the event is internal to the component. That is, calling `increase()` increments `this.value`, but there's no effect that leaves this component. When creating our own components we can also expose "public events" (component `outputs`) that allow the component to talk to the outside world.

The key thing to understand here is that in a view, we can listen to an event by using the `(output)="action"` syntax.

EMITTING CUSTOM EVENTS

Let's say we want to create a component that emits a custom event, like `click` or `mousedown` above. To create a custom output event we do three things:

1. Specify `outputs` in the `@Component` configuration
2. Attach an `EventEmitter` to the output property
3. Emit an event from the `EventEmitter`, at the right time

An `EventEmitter` is an object that helps you implement the Observer Pattern. That is, it's an object that will:

1. maintain a list of subscribers and
2. publish events to them.

Here's a short and sweet example of how you can use `EventEmitter`

```
let ee = new EventEmitter();
ee.subscribe((name: string) => console.log(`Hello ${name}`));
ee.emit("Nate");
```

When we assign an `EventEmitter` to an output Angular automatically subscribes for us. You don't need to do the subscription yourself (though in a special situation you could add your own subscriptions, if you want to).

Here's an example of how we write a component that has `outputs`:

```
@Component({
   selector: 'single-component',
   template: `
   <button (click)="liked()">Like it?</button>
   `
})
class SingleComponent {
   @Output() putRingOnIt: EventEmitter<string>;

   constructor() {
          this.putRingOnIt = new EventEmitter();
}
 liked(): void {
   this.putRingOnIt.emit("oh oh oh");
  }
}
```

Notice that we did all three steps: 1. specified `outputs`, 2. created an `EventEmitter` that we attached to the output property `putRingOnIt` and 3. Emitted an event when `liked` is called.

If we wanted to use this output in a parent component we could do something like this:

```
@Component({
      selector: 'club',
      template: `
      <div>
      <single-component
      (putRingOnIt)="ringWasPlaced($event)"
      ></single-component>
      </div>
      `
})
class ClubComponent {
      ringWasPlaced(message: string) {
            console.log(`Put your hands up: ${message}`);
      }
}
```

notice that:

- putRingOnIt comes from the outputs of SingleComponent
- ringWasPlaced is a function on the ClubComponent
- $event contains the thing that was emitted, in this case a string