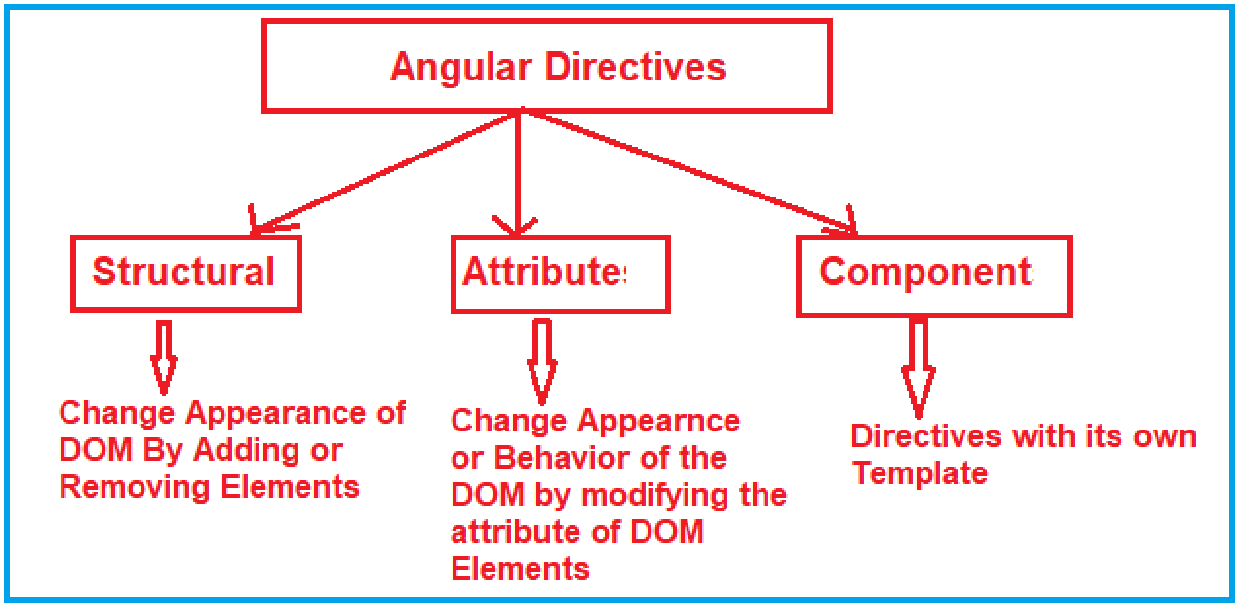

Angular Directives
Structural
Attributes
Component
Change Appearance of DOM By Adding or Removing Elements
Change Appearnce or Behavior of the DOM by modifying the attribute of DOM Elements
Directives with its own Template

Directives that use microtemplates are known as structural directives.

Attribute directives are classes that are able to modify the behavior or appearance of the element they are applied to.

Structural directives change the layout of the HTML document by adding and removing elements.
There are three built-in structural directives, NgIf, NgFor and NgSwitch.

*ngIf ="expression"

The ngIf directive is used to include an element and its content in the HTML document if the expression evaluates as true. The asterisk before the directive name indicates that this is a micro-template directive
<div *ngIf="true">This div will show if the expression evaluates to true</div>

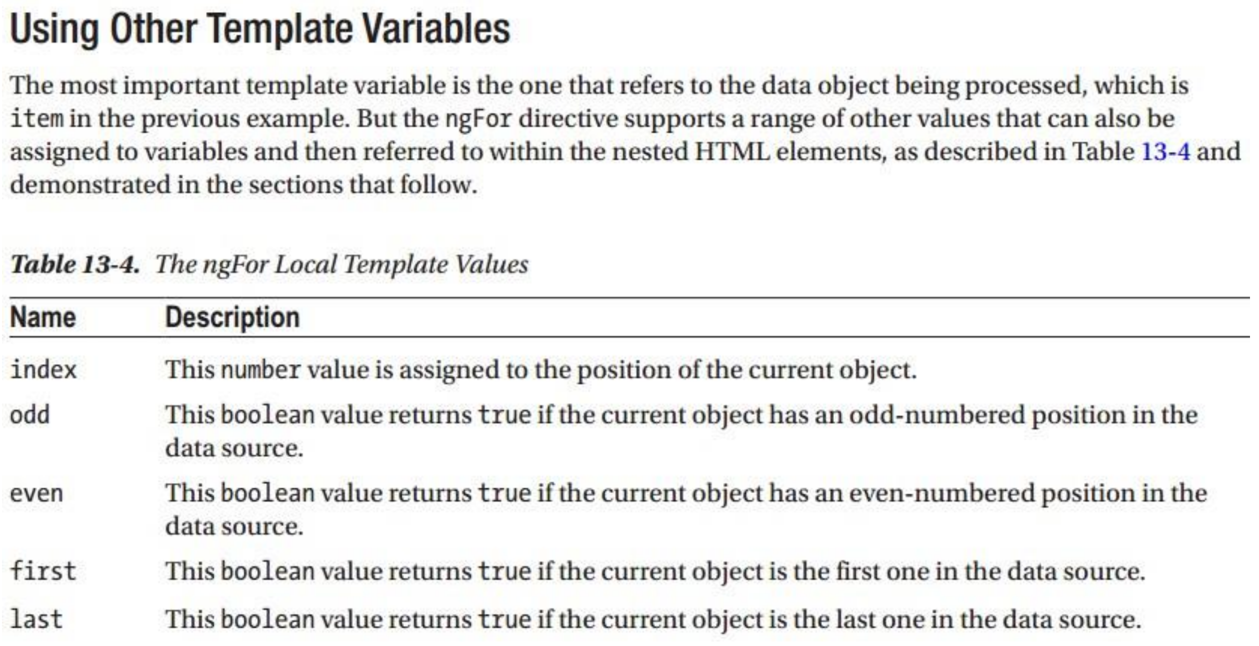
*ngFor ="let item of items; let i = index; let odd = odd; let first = first; let last = last"

The ngFor directive is used to generate the same set of elements for each object in an array. The asterisk before the directive name indicates that this is a micro-template directive

<div *ngFor="#item of expr"></div>

For the expression itself, there are two distinct parts, joined with the of keyword. The right-hand part of the expression provides the data source that will be enumerated

The left-hand side of the ngFor expression defines a *template variable*, denoted by the let keyword, which is how data is passed between elements within an Angular template.



Using Other Template Variables

The most important template variable is the one that refers to the data object being processed, which is `item` in the previous example. But the `ngFor` directive supports a range of other values that can also be assigned to variables and then referred to within the nested HTML elements, as described in Table 13-4 and demonstrated in the sections that follow.

Table 13-4. *The ngFor Local Template Values*

Name	Description
`index`	This `number` value is assigned to the position of the current object.
`odd`	This `boolean` value returns `true` if the current object has an odd-numbered position in the data source.
`even`	This `boolean` value returns `true` if the current object has an even-numbered position in the data source.
`first`	This `boolean` value returns `true` if the current object is the first one in the data source.
`last`	This `boolean` value returns `true` if the current object is the last one in the data source.

[ngSwitch] ="**expression** "

The ngSwitch directive is used to choose between multiple elements to include in the HTML document based on the result of an expression, which is then compared to the result of the individual expressions defined using ngSwitchCase directives. If none of the ngSwitchCase values matches, then the element to which the ngSwitchDefault directive has been applied will be used. The asterisks before the ngSwitchCase and ngSwitchDefault directives indicate they are micro-template directives.

```
<div [ngSwitch]="expr">
<span *ngSwitchCase=" epression "></span>
<span *ngSwitchDefault></span>
</div>
```

[ngTemplateOutlet] ="**titleTemplate**"

The ngTemplateOutlet directive is used to repeat a block of content at a specified location, which can be useful
when you need to generate the same content in different places and want to avoid duplication

```
<ng-template #titleTemplate>
<h4 class="p-2 bg-success text-white">Repeated Content</h4>
</ng-template>
<ng-template [ngTemplateOutlet]="titleTemplate"></ng-template>
some elements
<ng-template [ngTemplateOutlet]="titleTemplate"></ng-template>
```

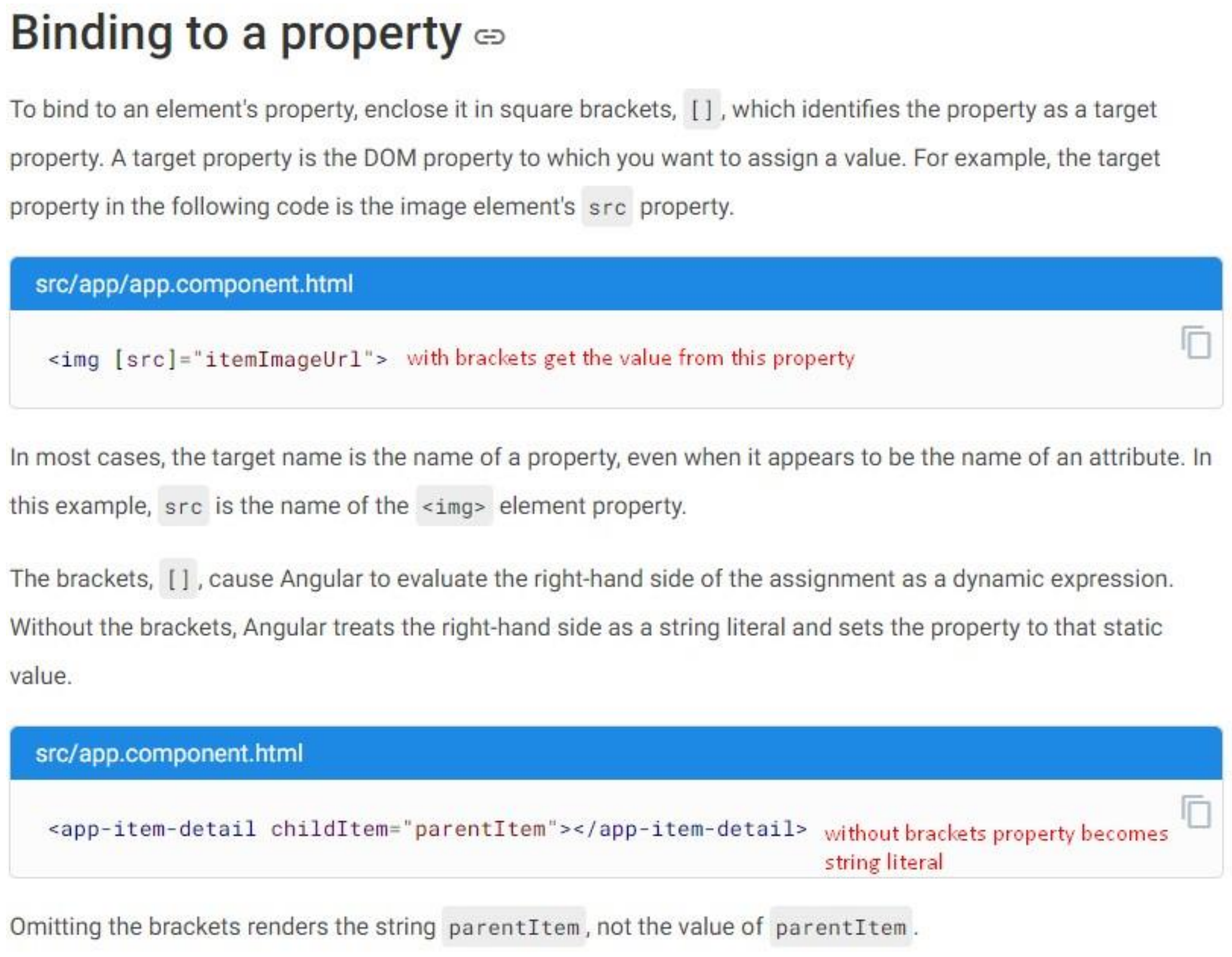
[(ngModel)] ="selectedProduct"
a two-way binding with [()] syntax (also known as 'banana-in-a-box syntax'), the value in the UI always syncs back to the domain model in your class. Behind the scenes, an event binding is applied to the input event, and a property binding is
applied to the value property

An attribute directive that updates styles for the containing HTML element.

[ngClass] =" 'someclass "

[ngStyle] ="{'font-style': styleExp}"

Property Binding



Binding to a property

To bind to an element's property, enclose it in square brackets, `[]`, which identifies the property as a target property. A target property is the DOM property to which you want to assign a value. For example, the target property in the following code is the image element's `src` property.

src/app/app.component.html

```
<img [src]="itemImageUrl">
```
with brackets get the value from this property

In most cases, the target name is the name of a property, even when it appears to be the name of an attribute. In this example, `src` is the name of the `<img>` element property.

The brackets, `[]`, cause Angular to evaluate the right-hand side of the assignment as a dynamic expression. Without the brackets, Angular treats the right-hand side as a string literal and sets the property to that static value.

src/app.component.html

```
<app-item-detail childItem="parentItem"></app-item-detail>
```
without brackets property becomes string literal

Omitting the brackets renders the string `parentItem`, not the value of `parentItem`.

Events Binding

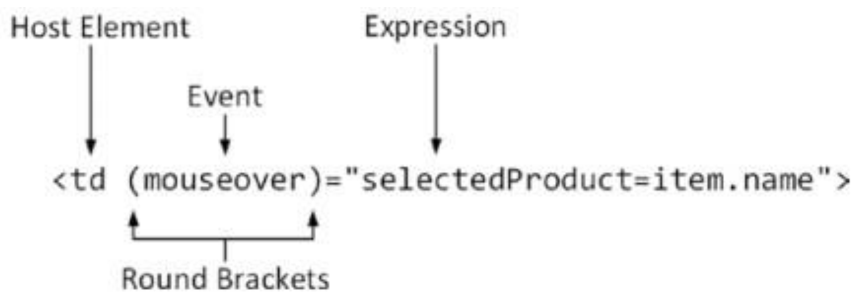


Figure 14-3. *The anatomy of an event binding*

An event binding has these four parts:

- The *host element* is the source of events for the binding.
- The *round brackets* tell Angular that this is an event binding, which is a form of one-way binding where data flows from the element to the rest of the application.
- The *event* specifies which event the binding is for.
- The *expression* is evaluated when the event is triggered.

Event bindings evaluate an expression when an event is triggered
Enabling forms support Add the @angular/forms module to
the application
Responding to an event Use an event binding
Getting details of an event Use the $event object
Events:
(mouseover)
(change)
(ngSubmit)
(click)
(input)
the input event is triggered every time the content in the input element is changed
(keyup)
the keyup event is a standard dom event, and the result is that application is updated as the user
releases each key while typing in the input element

Table 14-3. *The Properties Common to All DOM Event Objects*

Name	Description
`type`	This property returns a `string` that identifies the type of event that has been triggered.
`target`	This property returns the `object` that triggered the event, which will generally be the object that represents the HTML element in the DOM.
`timeStamp`	This property returns a `number` that contains the time that the event was triggered, expressed as milliseconds since January 1, 1970.

```
<div *ngIf="hero" class="name">{{hero.name}}</div>

<table class="table table-striped table-bordered table-sm">
    <thead>
        <tr>
            <th>#</th>
            <th>Description</th>
            <th>Done</th>
        </tr>
    </thead>
    <tbody>
        <tr *ngFor="let item of items; let i = index">
            <td>{{ i + 1 }}</td>
            <td>{{ item.task }}</td>
            <td><input type="checkbox" [(ngModel)]="item.complete" /></td>
            Before using the ngModel directive in a two-
way data binding, you must import the FormsModule and add it to the NgModule's im
ports list.
            <td [ngSwitch]="item.complete">
                <span *ngSwitchCase="true">Yes</span>
                <span *ngSwitchDefault>No</span>
            </td>
        </tr>
    </tbody>
</table>

<input class="form-control" placeholder="Enter task here" #todoText />
//template reference variable
<button class="btn btn-primary mt-1" (click)="addItem(todoText.value)"></button>
//get the value with help of /template reference variable
ili
<button class="btn btn-primary mt-
1" [value]="'some value'" (click)="addItem($event.target.value)"></button>

addItem(value)
value ke bide some value

ili

<tr *ngFor="let item of items; let i = index" (click)="addItem(item)>
```

```
 <select class="form-control" [value]="productsPerPage"
(change)="changePageSize($event.target.value)">
<option value="3">3 per Page</option>
<option value="4">4 per Page</option>
<option value="6">6 per Page</option>
<option value="8">8 per Page</option>
</select>

    <input type="number" class="form-control-sm"
    style="width:5em"
    [value]="line.quantity"
    (change)="cart.updateQuantity(line.product,
    $event.target.value)" />




<input class="form-control" [value]="model.getProduct(1)?.name || 'None'" />
...
If the result from the getProduct method isn’t null, then the expression will rea
d the value of the name
property and use it as the result. But if the result from the method is null, the
n the name property won’t be
read, and the null coalescing operator (the || characters) will set the result to
 None instead.
```

Boostrap and Jquery

1.

```
npm install --save bootstrap jquery
```

2. in angular.json add

```
  "styles": [
            "src/styles.css",

            "node_modules/bootstrap/dist/css/bootstrap.min.css"
          ],
          "scripts": [
            "node_modules/jquery/dist/jquery.min.js",
            "node_modules/bootstrap/dist/js/bootstrap.min.js"
          ]
```

ili

npm install bootstrap@4.4.1

vo style

"node_modules/bootstrap/dist/css/bootstrap.min.css" /vo style vo angular.json

Font Awsome

```
npm install font-awesome --save
"node_modules/font-awesome/css/font-awesome.css"  ///vo style vo angular.json
```

npm install -g @angular/cli

npm is the package manager for the Node JavaScript platform. It puts modules in place so that node can find them, and manages dependency conflicts intelligently. It is extremely configurable to support a wide variety of **use** cases. Most commonly, it is **used** to publish, discover, install, and develop node programs

ng v   //check angular cli version angular node

Create a workspace and initial application

ng new my-app

The Angular CLI includes a server, so that you can build and serve your app locally.
cd my-app ng serve --open

ng g c componentName

ng g c componentName -it -is //for inline template it and inline style is

Bindings are worth understanding because their expressions are re-evaluated when the data they depend on changes

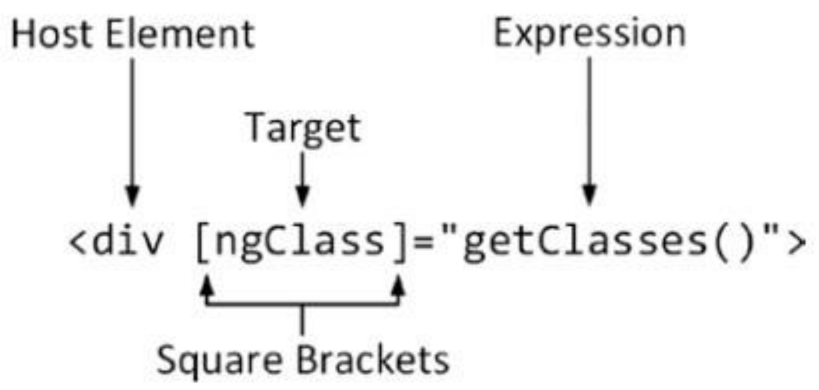


Figure 12-3. *The anatomy of a data binding*

A data binding has these four parts:

- The *host element* is the HTML element that the binding will affect, by changing its appearance, content, or behavior.
- The *square brackets* tell Angular that this is a one-way data binding. When Angular sees square brackets in a data binding, it will evaluate the expression and pass the result to the binding's *target* so that it can modify the host element.
- The *target* specifies what the binding will do. There are two different types of target: a *directive* or a *property binding*.
- The *expression* is a fragment of JavaScript that is evaluated using the template's component to provide context, meaning that the component's property and methods can be included in the expression, like the `getClasses` method in the example binding.

Types of data binding

Angular provides three categories of data binding according to the direction of data flow:

- From the source to view
- From view to source
- In a two way sequence of view to source to view

Type	Syntax	Category

Interpolation
Property
Attribute
Class
Style
	`{{expression}}`
	`[target]="expression"`
	`bind-target="expression"`
		One-way
		from data source
		to view target

Event	`(target)="statement"`
	`on-target="statement"`
		One-way
		from view target
		to data source

Two-way	`[(target)]="expression"`
	`bindon-target="expression"`
		Two-way

Binding types other than interpolation have a target name to the left of the equal sign. The target of a binding is a property or event, which you surround with square brackets, [ ], parentheses, ( ), or both, [( )].

The binding punctuation of [ ], ( ), [( )], and the prefix specify the direction of data flow.

- Use [ ] to bind from source to view.
- Use ( ) to bind from view to source.
- Use [( )] to bind in a two way sequence of view to source to view.

Place the expression or statement to the right of the equal sign within double quotes, " ". For more information see Interpolation and Template statements.

Binding types and targets

The target of a data binding can be a property, an event, or an attribute name. Every public member of a source directive is automatically available for binding in a template expression or statement. The following table summarizes the targets for the different binding types.

Type	**Target**	**Examples**

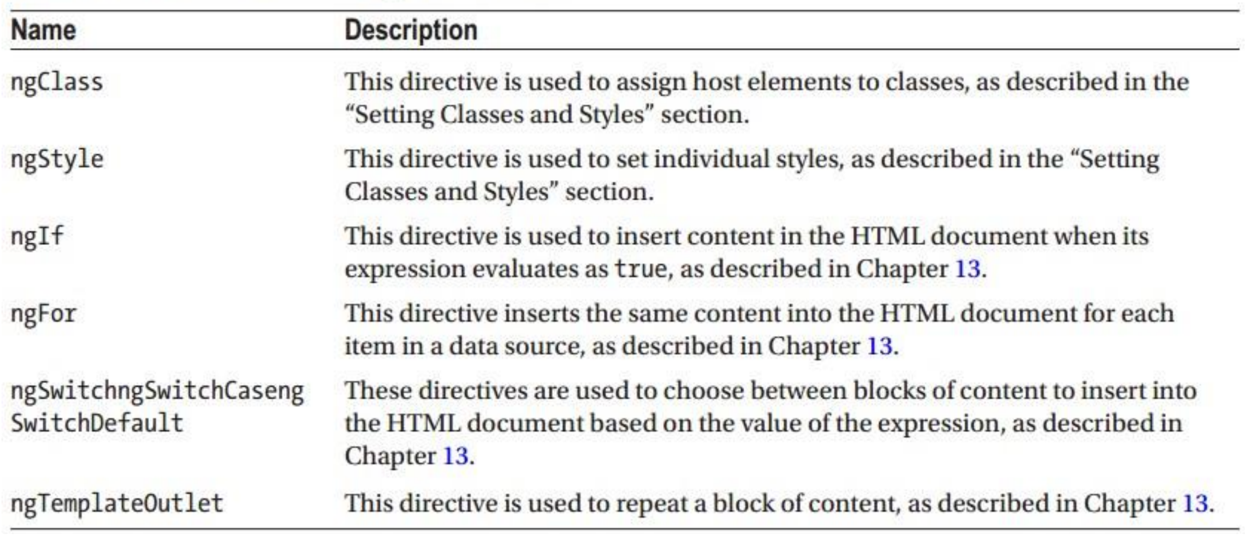
Property	Element property
	Component property
	Directive property
		`src`, `hero`, and `ngClass` in the following:
		`<img [src]="heroImageUrl">`
		`<app-hero-detail [hero]="currentHero"></app-hero-detail>`
		`<div [ngClass]="{'special': isSpecial}"></div>`

Event	Element event
	Component event
	Directive event
		`click`, `deleteRequest`, and `myClick` in the following:
		`<button (click)="onSave()">Save</button>`
		`<app-hero-detail (deleteRequest)="deleteHero()"></app-hero-detail>`
		`<div (myClick)="clicked=$event" clickable>click me</div>`

Two-way	Event and property	`<input [(ngModel)]="name">`
Attribute	Attribute (the exception)	`<button [attr.aria-label]="help">help</button>`
Class	`class` property	`<div [class.special]="isSpecial">Special</div>`
Style	`style` property	`<button [style.color]="isSpecial ? 'red' : 'green'">`

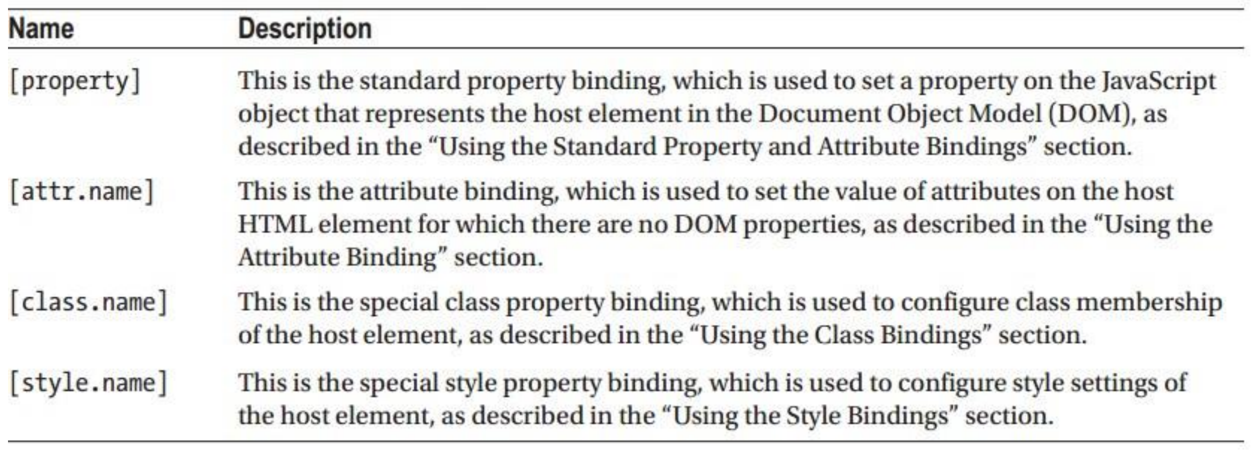
When Angular
sees square brackets in a data binding, it will evaluate the expression and pass the result to the binding's target so that it can modify the host element.
If the binding target doesn't correspond to a directive, then Angular checks to see whether the target can be used to create a property binding.
The target specifies what the binding will do. There are two different types of target: a directive or a property binding.

Table 12-3. The Basic Built-in Angular Directives



Name	Description
`ngClass`	This directive is used to assign host elements to classes, as described in the "Setting Classes and Styles" section.
`ngStyle`	This directive is used to set individual styles, as described in the "Setting Classes and Styles" section.
`ngIf`	This directive is used to insert content in the HTML document when its expression evaluates as `true`, as described in Chapter 13.
`ngFor`	This directive inserts the same content into the HTML document for each item in a data source, as described in Chapter 13.
`ngSwitchngSwitchCasengSwitchDefault`	These directives are used to choose between blocks of content to insert into the HTML document based on the value of the expression, as described in Chapter 13.
`ngTemplateOutlet`	This directive is used to repeat a block of content, as described in Chapter 13.

Table 12-4. The Angular Property Bindings



Name	Description
`[property]`	This is the standard property binding, which is used to set a property on the JavaScript object that represents the host element in the Document Object Model (DOM), as described in the "Using the Standard Property and Attribute Bindings" section.
`[attr.name]`	This is the attribute binding, which is used to set the value of attributes on the host HTML element for which there are no DOM properties, as described in the "Using the Attribute Binding" section.
`[class.name]`	This is the special class property binding, which is used to configure class membership of the host element, as described in the "Using the Class Bindings" section.
`[style.name]`	This is the special style property binding, which is used to configure style settings of the host element, as described in the "Using the Style Bindings" section.

The expression has access to the properties and methods defined by the component
Expressions are not restricted to calling methods or reading properties from the component; th ey can also perform most standard JavaScript operation

The expression is enclosed in double quotes, which means that the string literal has to be define d using single quotes

The square brackets (the [and] characters) tell Angular that this is a one-way data binding that has an expression that should be evaluated if you omit the brackets and the
target is a directive, the expression won't be evaluated, and the content between the quote characters will be passed to the directive as a literal value

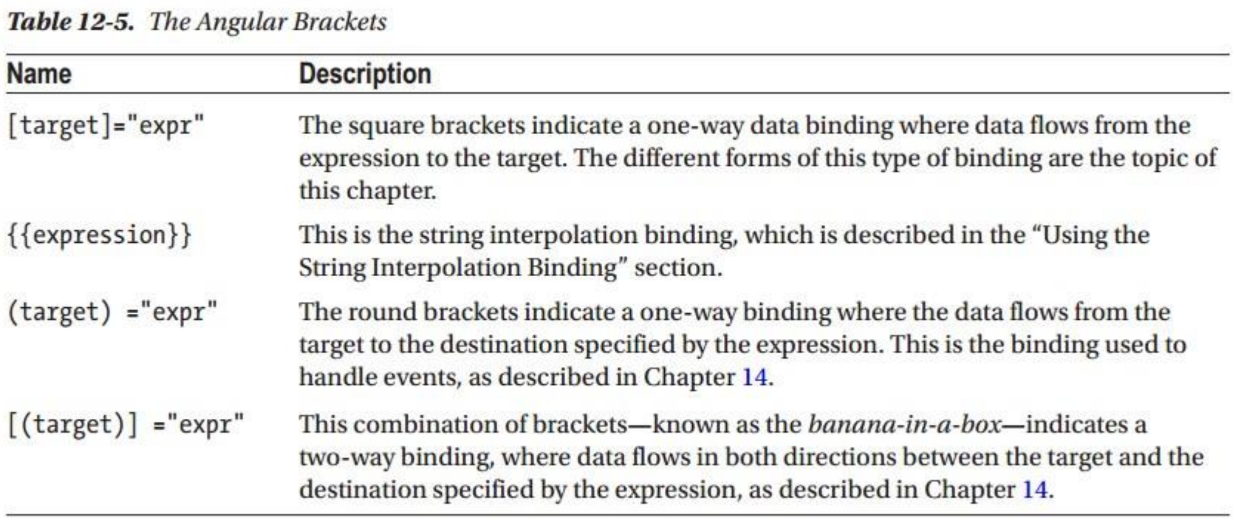


Table 12-5. *The Angular Brackets*

Name	Description
`[target]="expr"`	The square brackets indicate a one-way data binding where data flows from the expression to the target. The different forms of this type of binding are the topic of this chapter.
`{{expression}}`	This is the string interpolation binding, which is described in the "Using the String Interpolation Binding" section.
`(target) ="expr"`	The round brackets indicate a one-way binding where the data flows from the target to the destination specified by the expression. This is the binding used to handle events, as described in Chapter 14.
`[(target)] ="expr"`	This combination of brackets—known as the *banana-in-a-box*—indicates a two-way binding, where data flows in both directions between the target and the destination specified by the expression, as described in Chapter 14.

Class Binding

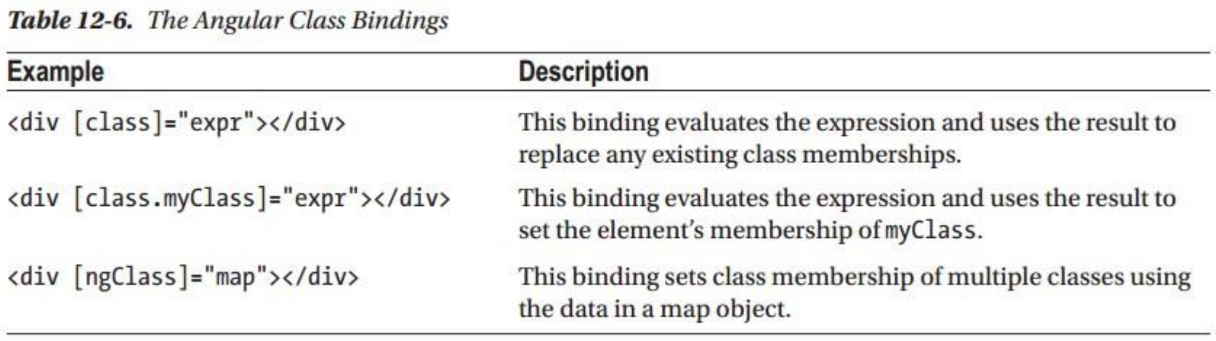
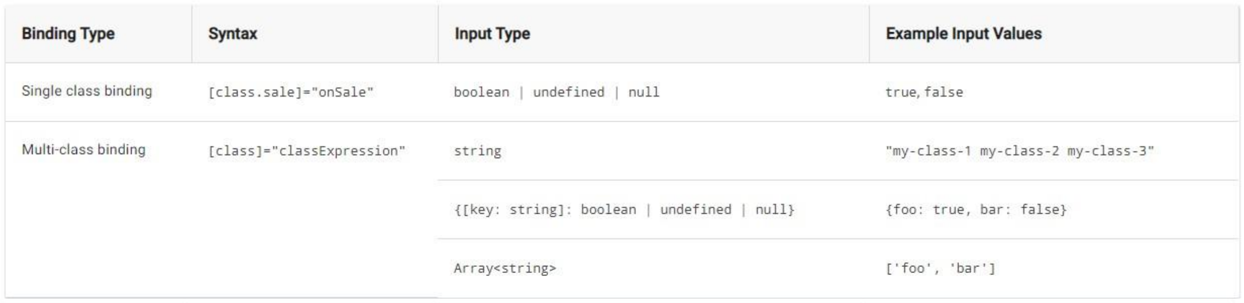


Table 12-6. *The Angular Class Bindings*

Example	Description
`<div [class]="expr"></div>`	This binding evaluates the expression and uses the result to replace any existing class memberships.
`<div [class.myClass]="expr"></div>`	This binding evaluates the expression and uses the result to set the element's membership of `myClass`.
`<div [ngClass]="map"></div>`	This binding sets class membership of multiple classes using the data in a map object.

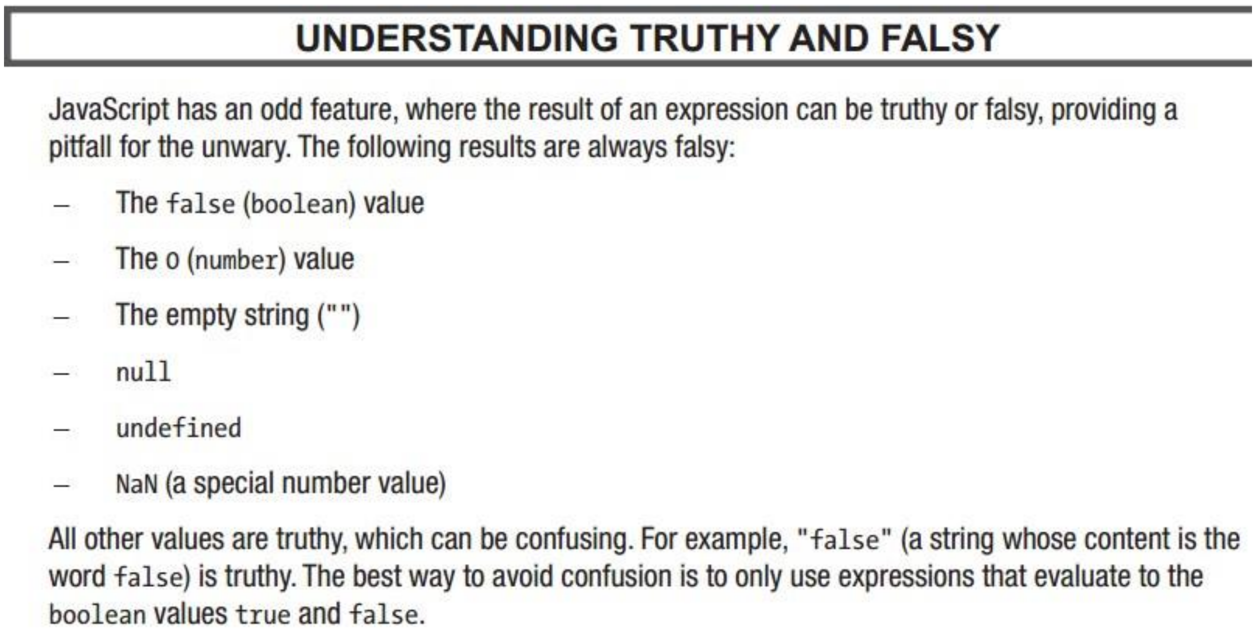


Binding Type	Syntax	Input Type	Example Input Values
Single class binding	`[class.sale]="onSale"`	`boolean | undefined | null`	`true, false`
Multi-class binding	`[class]="classExpression"`	`string`	`"my-class-1 my-class-2 my-class-3"`
		`{[key: string]: boolean | undefined | null}`	`{foo: true, bar: false}`
		`Array<string>`	`['foo', 'bar']`

Binding Type	Syntax	Input Type	Example Input Values
Single class binding	`[class.sale]="onSale"`	`boolean | undefined | null`	`true, false`
Multi-class binding	`[class]="classExpression"`	`string`	`"my-class-1 my-class-2 my-class-3"`
		`{[key: string]: boolean | undefined | null}`	`{foo: true, bar: false}`
		`Array<string>`	`['foo', 'bar']`

[class.bg -success]="model.getProduct(2).price < 50"

The special class binding will add the host element to the specified class if the result of the expression is truthy



UNDERSTANDING TRUTHY AND FALSY

JavaScript has an odd feature, where the result of an expression can be truthy or falsy, providing a pitfall for the unwary. The following results are always falsy:

- The `false` (`boolean`) value
- The `0` (`number`) value
- The empty string (`""`)
- `null`
- `undefined`
- `NaN` (a special number value)

All other values are truthy, which can be confusing. For example, `"false"` (a string whose content is the word `false`) is truthy. The best way to avoid confusion is to only use expressions that evaluate to the `boolean` values `true` and `false`.

ngClass

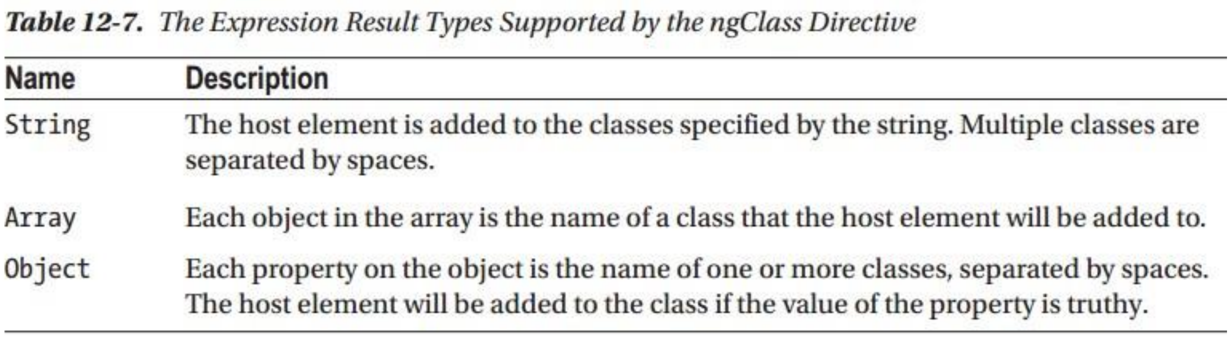


Table 12-7. The Expression Result Types Supported by the ngClass Directive

Name	Description
`String`	The host element is added to the classes specified by the string. Multiple classes are separated by spaces.
`Array`	Each object in the array is the name of a class that the host element will be added to.
`Object`	Each property on the object is the name of one or more classes, separated by spaces. The host element will be added to the class if the value of the property is truthy.

string

```
<some-element [ngClass]=" 'first second' " >...</some-element>
```

array

```
<some-element [ngClass]="['first', 'second']">...</some -element>
```

object

```
<some-element [ngClass]="{'first': true, 'second': true, 'third': false}">...</some -
element>
```

combination

```
<some-element [ngClass]="stringExp|arrayExp|objExp">...</some -element>
```

object

```
<some-element [ngClass]="{'class1 class2 class3' : true}">...</some -element>
```

Object

```
getClassMap(key: number): Object {
let product = this.model.getProduct(key);
return {
"text-center bg-danger": product.name == "Kayak",
"bg-info": product.price < 50 }; }
```

will evaluate to

```
{
"text-center bg-danger":true,
"bg-info":false
}
```

or

```
[ngClass]="{'bg-success': model.getProduct(3).price < 50,
'bg-info': model.getProduct(3).price >= 50}"
```

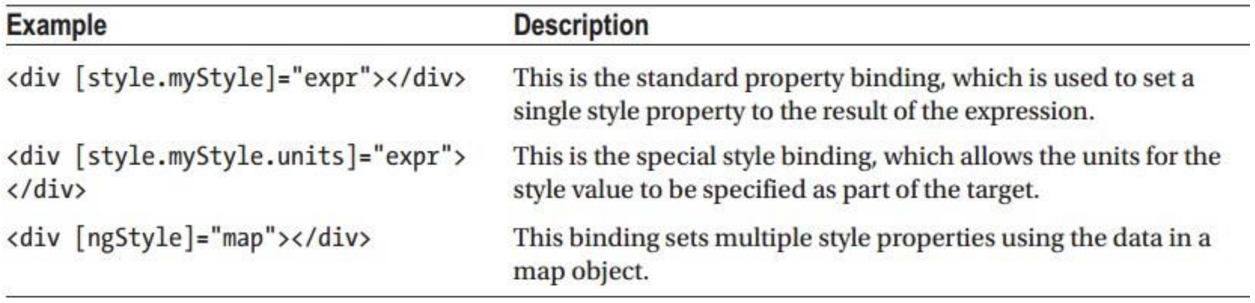
String

```
 [ngClass]="getClasses()"


   getClasses(): string {
      retur n this.model.getProducts().length == 5 ? "bg-success" : "bg-warning";
```

```
}
```

Style Binding

Table 12-8. *The Angular Style Bindings*



Example	Description
`<div [style.myStyle]="expr"></div>`	This is the standard property binding, which is used to set a single style property to the result of the expression.
`<div [style.myStyle.units]="expr"></div>`	This is the special style binding, which allows the units for the style value to be specified as part of the target.
`<div [ngStyle]="map"></div>`	This binding sets multiple style properties using the data in a map object.

Do not try to use the standard property binding to target the style property to set multiple style
values
if you want to set multiple style properties, then create a binding for
each of them or use the ngStyle directive.
[style.fontSize]="fontSizeWithUnits"
You can specify style properties using the Javascript property name format ([style.fontSize]) or
using the Css property name format ([style.font-size]).

The **ngStyle** directive allows multiple style properties to be set using a map object, similar to the way that
the ngClass directive works

```
getStyles(key: number) {
let product = this.model.getProduct(key);
return {
fontSize: "30px",
"margin.px": 100,
color: product.price > 50 ? "red" : "green"
};
}
```

Forms and Validation

the Angular validation features work only when there is a form element present, and Angular
will report an error if you add the ngControl directive to an element that is not contained in a form.
the novalidate attribute to the form element, which tells the browser not to use its native
validation features, which are inconsistently implemented by different browsers and generally get in the
way. Since Angular will be providing the validation, the browser’s own implementation of these features is
not required
the build in validation attribute show only colors

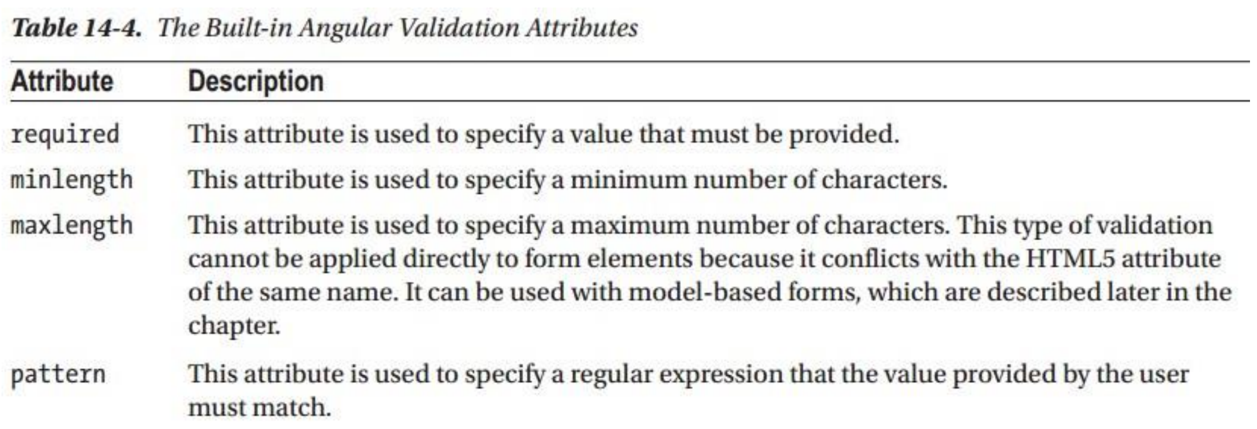
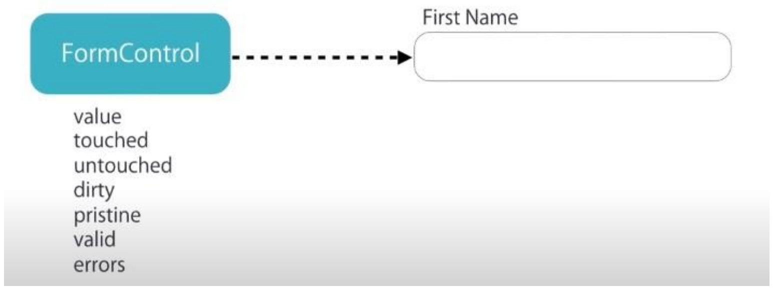


Table 14-4. The Built-in Angular Validation Attributes

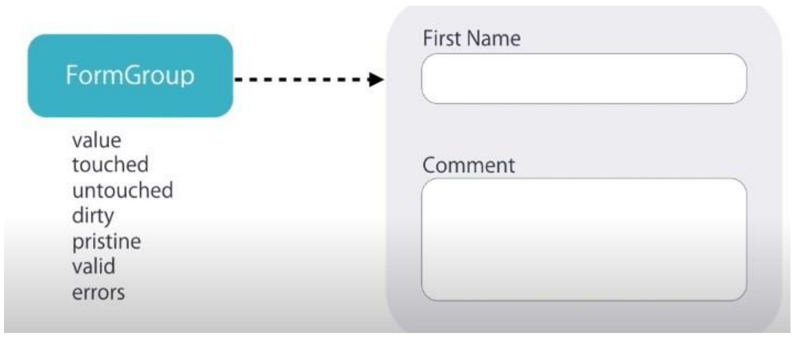
Attribute	Description
`required`	This attribute is used to specify a value that must be provided.
`minlength`	This attribute is used to specify a minimum number of characters.
`maxlength`	This attribute is used to specify a maximum number of characters. This type of validation cannot be applied directly to form elements because it conflicts with the HTML5 attribute of the same name. It can be used with model-based forms, which are described later in the chapter.
`pattern`	This attribute is used to specify a regular expression that the value provided by the user must match.

if we want text

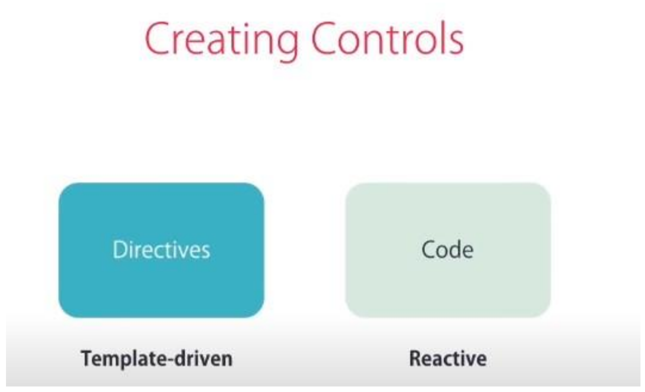
for each input we create FormControl class



for each form FormGroup object



the formControl and FormGroup object can be created with templete driven or reactive way



the FormsModule gives us template driven directives such as:

- ngModel and
- NgForm

Whereas ReactiveFormsModule gives us reactive driven directives like

- formControl and
- ngFormGroup … and several more

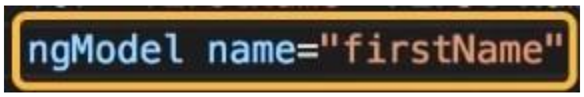
Templete Driven

if you import FormsModule, NgForm will get automatically attached to any
tags you have in your view. This is really useful but potentially confusing because it happens behind the scenes. There are two important pieces of functionality that NgForm gives us:

1. FormGroup named ngForm
2. (ngSubmit) output

```
<form #f="ngForm"  (ngSubmit)="onSubmit(f.value)">
```

form #f="ngForm" 4 (ngSubmit)="onSubmit(f.value)" First we have #f="ngForm". The #v="thing" syntax says that we want to create a local variable for this view. Here we're creating an alias to ngForm, for this view, bound to the variable #f. Where did ngForm come from in the first place? It came from the NgForm directive. And what type of object is ngForm? It is a FormGroup. That means we can use f as a FormGroup in our view. And that's exactly what we do in the (ngSubmit) output.



```
ngModel name="firstName"
```

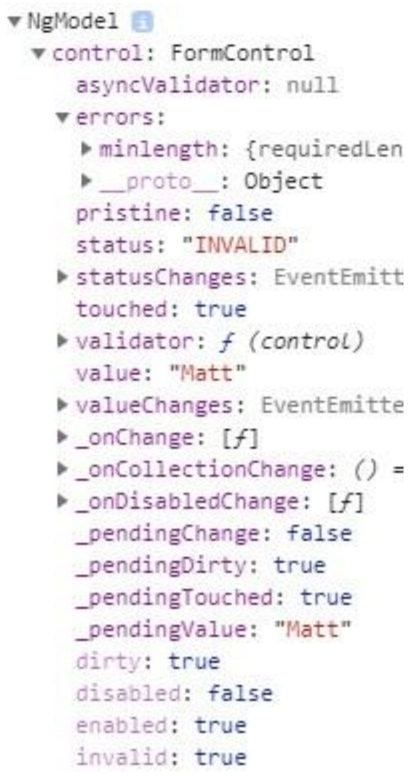
with this on an input angular creates formControl obj

```
<div>

<input class="form-control"
name="name"
[(ngModel)]="somename"
ngModel
```

```
#name="ngModel"
required
minlength="5"
pattern="^[A-Za-z ]+$"
(change)="ShowNgModel(name)" if we want to print the FormControl object
/>
<ul class="text-danger list-unstyled" *ngIf="name.dirty && name.invalid">
  <li *ngIf="name.errors.required">
    You must enter a product name
    </li>
</ul>

</div>
```

```
<form #f="ngForm"
(ngSubmit)="onSubmit(f.value)"
class="ui form">


<label for="skuInput">SKU</label>
<input type="text"
id="skuInput"
placeholder="SKU"
name="sku" ngModel>
```

NgModel creates a new FormControl that is automatically added to the parent FormGroup (in this case, on the form) and then binds a DOM element to that new FormControl. That is, it sets up an association between the input tag in our view and the FormControl and the association is matched by a name, in this case "sku"



```
▼NgModel
  ▼control: FormControl
     asyncValidator: null
    ▼errors:
      ▶minlength: {requiredLen
      ▶__proto__: Object
     pristine: false
     status: "INVALID"
    ▶statusChanges: EventEmitt
     touched: true
    ▶validator: ƒ (control)
     value: "Matt"
    ▶valueChanges: EventEmitte
    ▶_onChange: [ƒ]
    ▶_onCollectionChange: () =
    ▶_onDisabledChange: [ƒ]
     _pendingChange: false
     _pendingDirty: true
     _pendingTouched: true
     _pendingValue: "Matt"
     dirty: true
     disabled: false
     enabled: true
     invalid: true
```

Table 14-6. *The Validation Object Properties*

Name	Description
`path`	This property returns the name of the element.
`valid`	This property returns `true` if the element's contents are valid and `false` otherwise.
`invalid`	This property returns `true` if the element's contents are invalid and `false` otherwise.
`pristine`	This property returns `true` if the element's contents have not been changed.
`dirty`	This property returns `true` if the element's contents have been changed.
`touched`	This property returns `true` if the user has visited the element.
`untouched`	This property returns `true` if the user has not visited the element.
`errors`	This property returns an object whose properties correspond to each attribute for which there is a validation error.
`value`	This property returns the `value` of the element, which is used when defining custom validation rules, as described in the "Creating Custom Form Validators" section.

Table 14-7. *The Angular Form Validation Error Description Properties*

Name	Description
`required`	This property returns `true` if the `required` attribute has been applied to the input element. This is not especially useful because this can be deduced from the fact that the `required` property exists.
`minlength.requiredLength`	This property returns the number of characters required to satisfy the `minlength` attribute.
`minlength.actualLength`	This property returns the number of characters entered by the user.
`pattern.requiredPattern`	This property returns the regular expression that has been specified using the `pattern` attribute.
`pattern.actualValue`	This property returns the contents of the element.

import { NgForm } from "@angular/forms";

<form novalidate #form="ngForm" (ngSubmit)="submitForm(form)">

angular create FormGroup obj

```
submitForm(form: NgForm) {
this.formSubmitted = true;
if (form.valid) {
this.addProduct(this.newProduct);
this.newProduct = new Product();
form.reset();
this.formSubmitted = false;
}
}
```

NgForm provides the reset method, which resets the validation status of the form and returns it to its original and pristine state.

Reactive or Model-Based Forms

Using myForm in the view We want to change our <form>
to use myForm. If you recall, in the last section we said that ngForm is applied for us automatically when we use FormsModule. We also mentioned that ngForm creates its own FormGroup. Well, in this case, we don't want to use an outside FormGroup. Instead we want to use our instance variable myForm, which we created with our FormBuilder. How can we do that? Angular provides another directive that we use when we have an existing Form-Group: it's called formGroup and we use it like this:

```
<form [formGroup]="myForm"
```

Here we're telling Angular that we want to use myForm as the FormGroup for this form.

Remember how earlier we said that when using FormsModule that NgForm will be automatically applied to a
element? There is an exception: NgForm won't be applied to a
that has formGroup.

```
import {, ReactiveFormsModule } from "@angular/forms";
import { FormControl, FormGroup, Validators } from "@angular/forms";
```

1. create formControl

```
nekojFormControl:FormControl=new FormControl("value of the input field",Validators.required)
```

2.Add it to inoput element

```
<input class="form-control" name="name" [formControl]="nekojFormControl" />

 {{nekojFormControl.value}} this is the FormControl object
```

you can create custom class instead of FormControl that extend FormControl
you can create custom class instead of FormGroup that extend FormGroup

1.create formGroup obj

```
nekojaFormGroup:FormGroup=new FormGroup({

   nekojFormControl:new FormControl("sssdAAAAAAAAAAAaasx",Validators.required)

 });
```

2.Add it to form element

```
<form [formGroup]="nekojaFormGroup">

<input class="form-control" name="name" formControlName="nekojFormControl" />

</form>
```

formControlName must be used with a parent formGroup directive

```
  Example:
   <div [formGroup]="myGroup">
     <input formControlName="firstName">
   </div>
```

Nested Form Group

.ts

```
  nekojaFormGroup:FormGroup=new FormGroup({

    nekojFormControl:new FormControl("sssdAAAAAAAAAAAaasx",Validators.required),

    nestedformgroup:new FormGroup({

      nestedFormControl:new FormControl("BBBBBBBBBBBBBBBBBBB",Validators.required
)

    })

  });
```

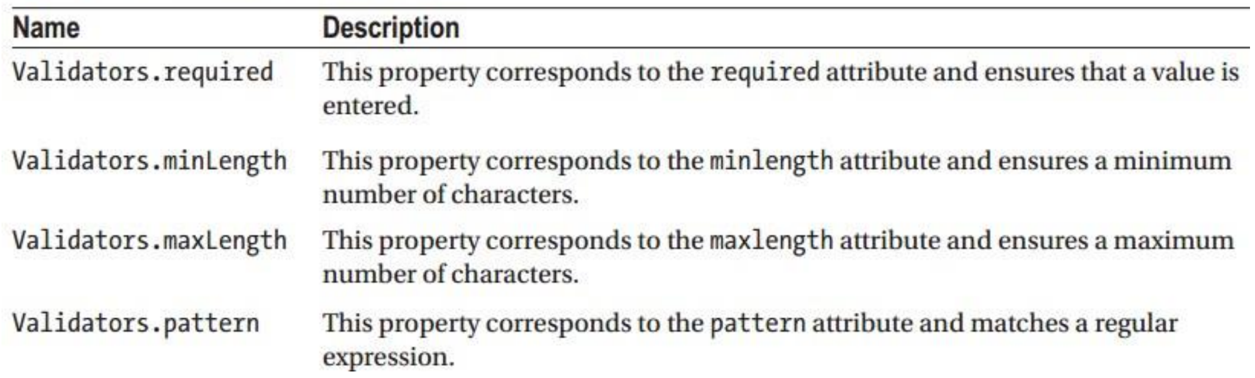
.html

```
<form [formGroup]="nekojaFormGroup">

    <input class="form-control" name="name" formControlName="nekojFormControl" />

    <section formGroupName="nestedformgroup">
    <input class="form-
control" name="name" formControlName="nestedFormControl" />
    </section>

    </form>
```

Validating

Angular defines a class called Validators in the @angular/forms
module that has properties for each of the built-in validation checks, as described in Table 14-8.

Table 14-8. *The Validator Properties*



Name	Description
`Validators.required`	This property corresponds to the `required` attribute and ensures that a value is entered.
`Validators.minLength`	This property corresponds to the `minlength` attribute and ensures a minimum number of characters.
`Validators.maxLength`	This property corresponds to the `maxlength` attribute and ensures a maximum number of characters.
`Validators.pattern`	This property corresponds to the `pattern` attribute and matches a regular expression.

`Validators.compose`

The Validators.compose method accepts an array of validators

```
  nestedFormControl:new FormControl("BBBBBBBBBBBBBBBBBBB",Validators.compose([Val
idators.required,Validators.minLength(5)]))
```

Validated
FormControl i FormGroup objects have error property

Custom Validator
1.Create Validator logic
limit.validator.ts

```
import { FormControl } from "@angular/forms";


export class LimitValidator {

     static Limit(limit: number) {

        return (control: FormControl) => {

            let val = Number(control.value);

            if (val != NaN && val > limit) {
                return { "limit": { "limit": limit, "actualValue": val } };

            } else {
                return null;
            }
        }
```

```
    }
} The limit property returns an object that has a limit property that is set to the validation limit and an
actualValue property that is set to the value entered by the user
```

2. adding validator to form control

```
 nekojFormControl:new FormControl("sssdAAAAAAAAAAAaasx",LimitValidator.Limit(10))
```

```
input.ng-dirty.ng-invalid { border: 2px solid #ff0000 }
input.ng-dirty.ng-valid { border: 2px solid #6bc502 }
```

Remember: To create a new FormGroup and FormControls implicitly use: • ngForm and • ngModel But to bind to an existing FormGroup and FormControls use: • formGroup and • formControl

Custom Directive

```
import { Directive, ElementRef,Attribute,Input } from "@angular/core";


@Directive({
    selector: "[pa]",
})
export class PaAttrDirective {


    constructor(element: ElementRef) {

        element.nativeElement.classList.add("btn-danger");

    }

}
```

```
import {PaAttrDirective} from './mydirectiv.directive'
and add it to declarations

in @NgModule
```

```
<button  pa >2222222222222222</button>
```

Configure Custom Directive with Attribute

```
import {PaAttrDirective} from './mydirectiv.directive'
and add it to declarations

in @NgModule
```

```
import { Directive, ElementRef,Attribute,Input } from "@angular/core";


@Directive({
    selector: "[pa]",
})
export class PaAttrDirective {


    constructor(element: ElementRef,@Attribute("paattr") bgClass: string) {

        element.nativeElement.classList.add(bgClass || "btn-danger");

    }
}
```

must be both the directive and attribute

```
<button pa paattr="btn-primary" >1111111111111111</button>
```

Configure Custom Directive with Input

```
import {PaAttrDirective} from './mydirectiv.directive'
and add it to declarations

in @NgModule
```

```
import { Directive, ElementRef ,Input } from "@angular/core";


@Directive({
    selector: "[pa]",
})
export class PaAttrDirective {

    @Input("pa")
     bgClass: string;//The input name needs to match the selector to be able to a
ssign this way

    constructor(private element: ElementRef) {

    }
    ngOnInit() //This method is called after Angular has set the initial value fo
r all the input properties that the directive has declared
    {
        this.element.nativeElement.classList.add(this.bgClass || "btn-danger");
    }
}
```

[pa] expect expression

```
<button  [pa]="'btn-primary'" >1111111111111111</button>
```

Structural Directive

```
import {PaStructureDirective} from './myStructDirective.sdirective'
```

and add it to declarations

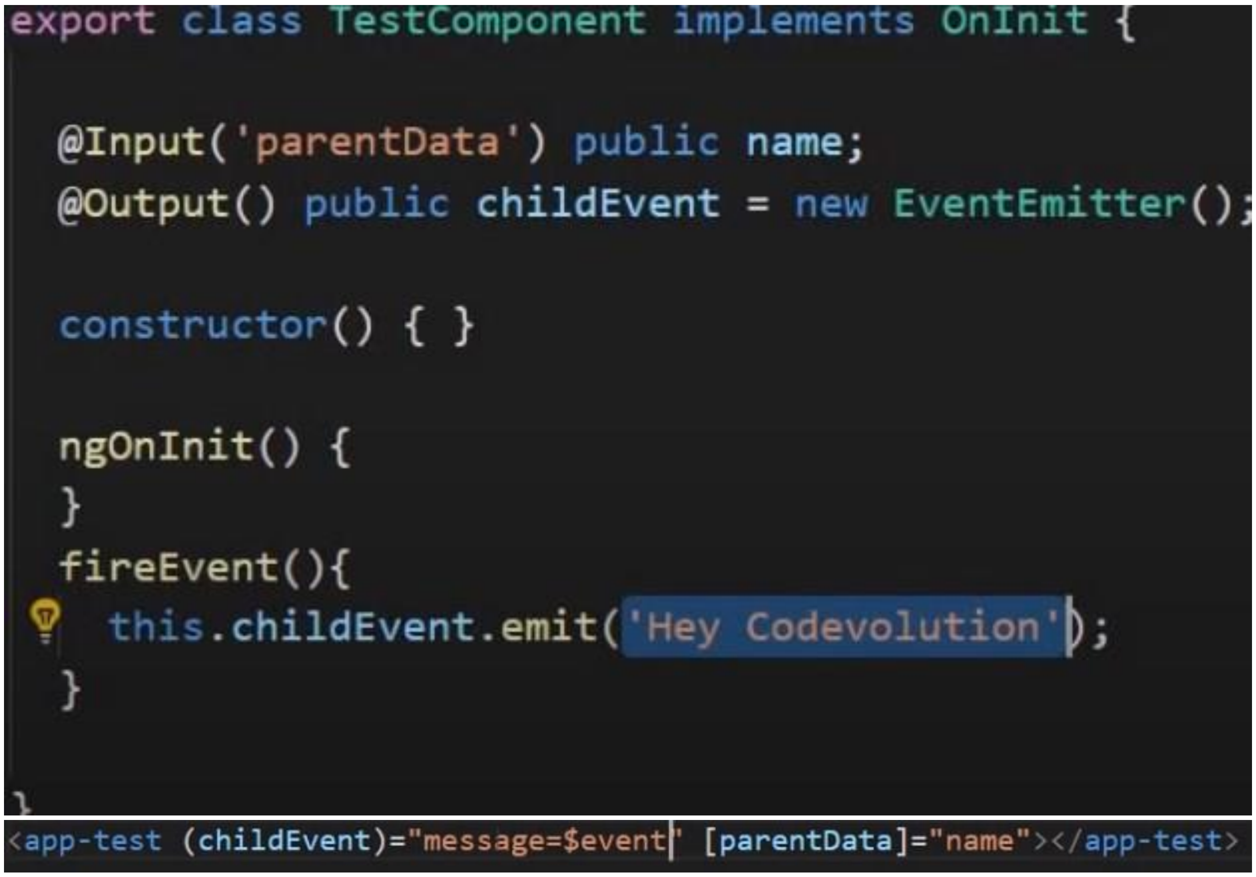
in @NgModule

```
import {  Directive, SimpleChange, ViewContainerRef, TemplateRef, Input} from "@a
ngular/core";

@Directive({
    selector: "[paIf]"

})
export class PaStructureDirective {

    constructor(private container: ViewContainerRef,
        private template: TemplateRef<Object>) { }

    @Input("paIf")
    expressionResult: boolean;

    ngOnChanges(changes: { [property: string]: SimpleChange }) {
        let change = changes["expressionResult"];
        if (!change.isFirstChange() && !change.currentValue) {
            this.container.clear();
        } else if (change.currentValue) {

            this.container.createEmbeddedView(this.template);
        }
    }
}
```

use the directive

```
<div *paIf="false">Pa if applied on this div</div>
```

Component interaction child to parent



```
export class TestComponent implements OnInit {

  @Input('parentData') public name;
  @Output() public childEvent = new EventEmitter();

  constructor() { }

  ngOnInit() {
  }
  fireEvent(){
    this.childEvent.emit('Hey Codevolution');
  }

}
<app-test (childEvent)="message=$event" [parentData]="name"></app-test>
```

Services

all the components in the application that have declared a dependency
on DiscountService have received the same object.
each component obtaining the share objects it needs through the dependency
injection feature, rather than relying on its parent component to provide it.