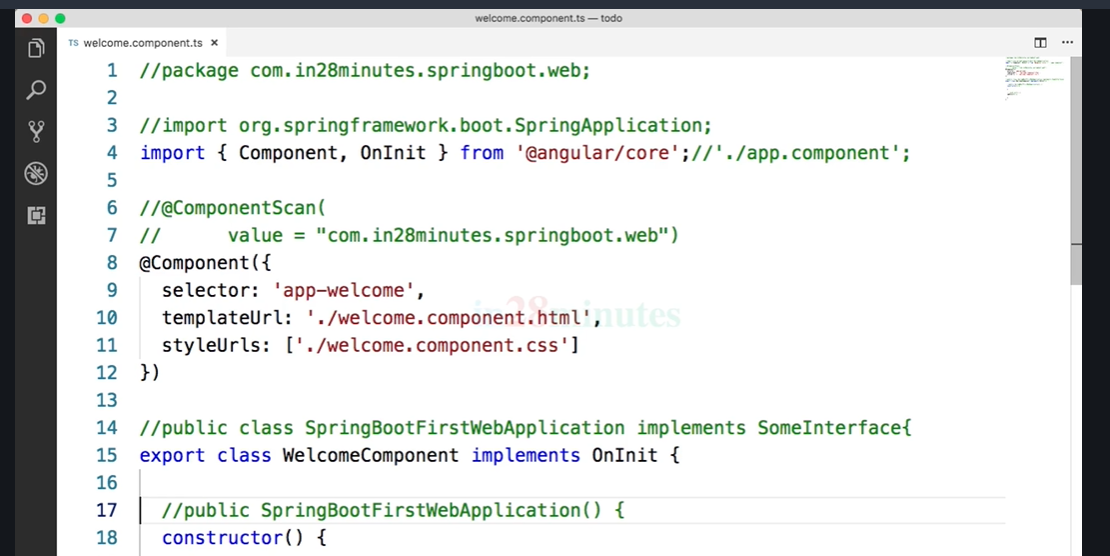
Full-stack with Angular and Spring boot.

# Language variation java vs typescript.



# Interpolation

Refer: <https://angular.io/guide/interpolation>

Interpolation refers to embedding expressions into marked up text. By default, interpolation uses as its delimiter the double curly braces, {{ and }}.

In the following snippet, {{ currentCustomer }} is an example of interpolation.

src/app/app.component.html

content\_copy<h3>Current customer: {{ currentCustomer }}</h3>

The text between the braces is often the name of a component property. Angular replaces that name with the string value of the corresponding component property.

src/app/app.component.html

content\_copy<p>{{title}}</p>

<div><img src="{{itemImageUrl}}"></div>

In the example above, Angular evaluates the title and itemImageUrl properties and fills in the blanks, first displaying some title text and then an image.

More generally, the text between the braces is a **template expression** that Angular first **evaluates** and then **converts to a string**. The following interpolation illustrates the point by adding two numbers:

src/app/app.component.html

content\_copy<!-- "The sum of 1 + 1 is 2" -->

<p>The sum of 1 + 1 is {{1 + 1}}.</p>

The expression can invoke methods of the host component such as getVal() in the following example:

src/app/app.component.html

content\_copy<!-- "The sum of 1 + 1 is not 4" -->

<p>The sum of 1 + 1 is not {{1 + 1 + getVal()}}.</p>

Angular evaluates all expressions in double curly braces, converts the expression results to strings, and links them with neighboring literal strings. Finally, it assigns this composite interpolated result to an **element or directive property**.

You appear to be inserting the result between element tags and assigning it to attributes. However, interpolation is a special syntax that Angular converts into a property binding.

If you'd like to use something other than {{ and }}, you can configure the interpolation delimiter via the [interpolation](https://angular.io/api/core/Component#interpolation) option in the [Component](https://angular.io/api/core/Component) metadata.

## **Template expressions**

A template **expression** produces a value and appears within the double curly braces, {{ }}. Angular executes the expression and assigns it to a property of a binding target; the target could be an HTML element, a component, or a directive.

The interpolation braces in {{1 + 1}} surround the template expression 1 + 1. In the property binding, a template expression appears in quotes to the right of the = symbol as in [property]="expression".

In terms of syntax, template expressions are similar to JavaScript. Many JavaScript expressions are legal template expressions, with a few exceptions.

You can't use JavaScript expressions that have or promote side effects, including:

* Assignments (=, +=, -=, ...)
* Operators such as new, typeof, instanceof, etc.
* Chaining expressions with ; or ,
* The increment and decrement operators ++ and --
* Some of the ES2015+ operators

Other notable differences from JavaScript syntax include:

* No support for the bitwise operators such as | and &
* New [template expression operators](https://angular.io/guide/template-expression-operators), such as |, ?. and !

## **Expression context**

The expression context is typically the component instance. In the following snippets, the recommended within double curly braces and the itemImageUrl2 in quotes refer to properties of the AppComponent.

src/app/app.component.html

content\_copy<h4>{{recommended}}</h4>

<img [src]="itemImageUrl2">

An expression may also refer to properties of the template's context such as a template input variable,

let customer, or a template reference variable, #customerInput.

src/app/app.component.html (template input variable)

content\_copy<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let customer of customers">{{customer.name}}</li>

</ul>

src/app/app.component.html (template reference variable)

content\_copy<label>[Type](https://angular.io/api/core/Type) something:

<input #customerInput>{{customerInput.value}}

</label>

The context for terms in an expression is a blend of the template variables, the directive's context object (if it has one), and the component's members. If you reference a name that belongs to more than one of these namespaces, the template variable name takes precedence, followed by a name in the directive's context, and, lastly, the component's member names.

The previous example presents such a name collision. The component has a customer property and the \*[ngFor](https://angular.io/api/common/NgForOf) defines a customer template variable.

The customer in {{customer.name}} refers to the template input variable, not the component's property.

Template expressions cannot refer to anything in the global namespace, except undefined. They can't refer to window or document. Additionally, they can't call console.log() or Math.max() and they are restricted to referencing members of the expression context.

# Template Statements

A template **statement** responds to an **event** raised by a binding target such as an element, component, or directive.

See the [Template syntax](https://angular.io/generated/live-examples/template-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/template-syntax/template-syntax.zip) for the syntax and code snippets in this guide.

The following template statement appears in quotes to the right of the = symbol as in (event)="statement".

src/app/app.component.html

content\_copy<button (click)="deleteHero()">Delete hero</button>

A template statement has a side effect. That's the whole point of an event. It's how you update application state from user action.

Responding to events is the other side of Angular's "unidirectional data flow". You're free to change anything, anywhere, during this turn of the event loop.

Like template expressions, template statements use a language that looks like JavaScript. The template statement parser differs from the template expression parser and specifically supports both basic assignment (=) and chaining expressions with ;.

However, certain JavaScript and template expression syntax is not allowed:

* new
* increment and decrement operators, ++ and --
* operator assignment, such as += and -=
* the bitwise operators, such as | and &
* the [pipe operator](https://angular.io/guide/template-expression-operators#pipe)

## **Statement context**

As with expressions, statements can refer only to what's in the statement context such as an event handling method of the component instance.

The statement context is typically the component instance. The deleteHero in (click)="deleteHero()" is a method of the data-bound component.

src/app/app.component.html

content\_copy<button (click)="deleteHero()">Delete hero</button>

The statement context may also refer to properties of the template's own context. In the following examples, the template $event object, a [template input variable](https://angular.io/guide/built-in-directives#template-input-variable) (let hero), and a [template reference variable](https://angular.io/guide/template-reference-variables) (#heroForm) are passed to an event handling method of the component.

src/app/app.component.html

content\_copy<button (click)="onSave($event)">Save</button>

<button \*[ngFor](https://angular.io/api/common/NgForOf)="let hero of heroes" (click)="deleteHero(hero)">{{hero.name}}</button>

<form #heroForm (ngSubmit)="onSubmit(heroForm)"> ... </form>

Template context names take precedence over component context names. In deleteHero(hero) above, the hero is the template input variable, not the component's hero property.

## **Statement guidelines**

Template statements cannot refer to anything in the global namespace. They can't refer to window or document. They can't call console.log or Math.max.

As with expressions, avoid writing complex template statements. A method call or simple property assignment should be the norm.

# Data Binding

Angular provides many kinds of data-binding. Binding types can be grouped into three categories distinguished by the direction of data flow:

* From the source-to-view
* From view-to-source
* Two-way sequence: view-to-source-to-view

|  |  |  |
| --- | --- | --- |
| **Type** | **Syntax** | **Category** |
| Interpolation Property Attribute Class Style | content\_copy{{expression}}  [target]="expression"  bind-target="expression" | One-way from data source to view target |
| Event | content\_copy(target)="statement"  on-target="statement" | One-way from view target to data source |
| Two-way | content\_copy[(target)]="expression"  bindon-target="expression" | Two-way |

Binding types other than interpolation have a **target name** to the left of the equal sign, either surrounded by punctuation, [] or (), or preceded by a prefix: bind-, on-, bindon-.

The target of a binding is the property or event inside the binding punctuation: [], () or [()].

Every public member of a **source** directive is automatically available for binding. You don't have to do anything special to access a directive member in a template expression or statement.

### **Data-binding and HTML**

In the normal course of HTML development, you create a visual structure with HTML elements, and you modify those elements by setting element attributes with string constants.

content\_copy<div class="special">Plain old HTML</div>

<img src="images/item.png">

<button disabled>Save</button>

With data-binding, you can control things like the state of a button:

src/app/app.component.html

content\_copy<!-- Bind button disabled [state](https://angular.io/api/animations/state) to `isUnchanged` property -->

<button [disabled]="isUnchanged">Save</button>

Notice that the binding is to the disabled property of the button's DOM element, **not** the attribute. This applies to data-binding in general. Data-binding works with properties of DOM elements, components, and directives, not HTML attributes.

### **HTML attribute vs. DOM property**

The distinction between an HTML attribute and a DOM property is key to understanding how Angular binding works. **Attributes are defined by HTML. Properties are accessed from DOM (Document Object Model) nodes.**

* A few HTML attributes have 1:1 mapping to properties; for example, id.
* Some HTML attributes don't have corresponding properties; for example, aria-\*.
* Some DOM properties don't have corresponding attributes; for example, textContent.

It is important to remember that HTML attribute and the DOM property are different things, even when they have the same name. In Angular, the only role of HTML attributes is to initialize element and directive state.

**Template binding works with**properties**and**events**, not**attributes**.**

When you write a data-binding, you're dealing exclusively with the DOM properties and events of the target object.

This general rule can help you build a mental model of attributes and DOM properties: **Attributes initialize DOM properties and then they are done. Property values can change; attribute values can't.**

There is one exception to this rule. Attributes can be changed by setAttribute(), which re-initializes corresponding DOM properties.

For more information, see the [MDN Interfaces documentation](https://developer.mozilla.org/en-US/docs/Web/API#Interfaces) which has API docs for all the standard DOM elements and their properties. Comparing the [<td> attributes](https://developer.mozilla.org/en-US/docs/Web/HTML/Element/td) attributes to the [<td> properties](https://developer.mozilla.org/en-US/docs/Web/API/HTMLTableCellElement) provides a helpful example for differentiation. In particular, you can navigate from the attributes page to the properties via "DOM interface" link, and navigate the inheritance hierarchy up to HTMLTableCellElement.

#### **Example 1: an <input>**

When the browser renders <input type="text" value="Sarah">, it creates a corresponding DOM node with a value property initialized to "Sarah".

content\_copy<input type="text" value="Sarah">

When the user enters "Sally" into the <input>, the DOM element value property becomes "Sally". However, if you look at the HTML attribute value using input.getAttribute('value'), you can see that the attribute remains unchanged—it returns "Sarah".

The HTML attribute value specifies the initial value; the DOM value property is the current value.

To see attributes versus DOM properties in a functioning app, see the [live example](https://angular.io/generated/live-examples/binding-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/binding-syntax/binding-syntax.zip) especially for binding syntax.

#### **Example 2: a disabled button**

The disabled attribute is another example. A button's disabled property is false by default so the button is enabled.

When you add the disabled attribute, its presence alone initializes the button's disabled property to true so the button is disabled.

content\_copy<button disabled>Test Button</button>

Adding and removing the disabled attribute disables and enables the button. However, the value of the attribute is irrelevant, which is why you cannot enable a button by writing <button disabled="false">Still Disabled</button>.

To control the state of the button, set the disabled property,

Though you could technically set the [attr.disabled] attribute binding, the values are different in that the property binding requires to a boolean value, while its corresponding attribute binding relies on whether the value is null or not. Consider the following:

content\_copy<input [disabled]="condition ? true : false">

<input [attr.disabled]="condition ? 'disabled' : null">

Generally, use property binding over attribute binding as it is more intuitive (being a boolean value), has a shorter syntax, and is more performant.

To see the disabled button example in a functioning app, see the [live example](https://angular.io/generated/live-examples/binding-syntax/stackblitz.html) / [download example](https://angular.io/generated/zips/binding-syntax/binding-syntax.zip) especially for binding syntax. This example shows you how to toggle the disabled property from the component.

## **Binding types and targets**

The **target of a data-binding** is something in the DOM. Depending on the binding type, the target can be a property (element, component, or directive), an event (element, component, or directive), or sometimes an attribute name. 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](https://angular.io/api/common/NgClass) in the following:  content\_copy<img [src]="heroImageUrl">  <app-hero-detail [hero]="currentHero"></app-hero-detail>  <div [[ngClass](https://angular.io/api/common/NgClass)]="{'special': isSpecial}"></div> |
| Event | Element event Component event Directive event | click, deleteRequest, and myClick in the following:  content\_copy<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 | content\_copy<input [([ngModel](https://angular.io/api/forms/NgModel))]="name"> |
| Attribute | Attribute (the exception) | content\_copy<button [attr.aria-label]="help">help</button> |
| Class | class property | content\_copy<div [class.special]="isSpecial">Special</div> |
| Style | [style](https://angular.io/api/animations/style) property | content\_copy<button [style.color]="isSpecial ? 'red' : 'green'"> |