Template Syntax

The Angular application manages what the user sees and can do, achieving this through the interaction of a component class instance (the *component*) and its user-facing template.

You may be familiar with the component/template duality from your experience with model-view-controller (MVC) or model-view-viewmodel (MVVM). In Angular, the component plays the part of the controller/viewmodel, and the template represents the view.

This page is a comprehensive technical reference to the Angular template language. It explains basic principles of the template language and describes most of the syntax that you'll encounter elsewhere in the documentation.

Many code snippets illustrate the points and concepts, all of them available in the Template Syntax Live Code / download example.

HTML in templates

HTML is the language of the Angular template. Almost all HTML syntax is valid template syntax. The `<script>` element is a notable exception; it is forbidden, eliminating the risk of script injection attacks. In practice, `<script>` is ignored and a warning appears in the browser console. See the Security page for details.

Some legal HTML doesn't make much sense in a template. The `<html>`, `<body>`, and `<base>` elements have no useful role. Pretty much everything else is fair game.

You can extend the HTML vocabulary of your templates with components and directives that appear as new elements and attributes. In the following sections, you'll learn how to get and set DOM (Document Object Model) values dynamically through data binding.

Begin with the first form of data binding—interpolation—to see how much richer template HTML can be.

Interpolation and Template Expressions

Interpolation allows you to incorporate calculated strings into the text between HTML element tags and within attribute assignments. Template expressions are what you use to calculate those strings.

The interpolation live example / download example demonstrates all of the syntax and code snippets described in this section.

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

```
<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

```
<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

```
<!-- "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

```
<!-- "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 option in the `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, 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

```
<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)

```
<ul>
  <li *ngFor="let customer of customers">{{customer.name}}</li>
</ul>
```

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

```
<label>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` 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.

Expression guidelines

When using template expressions follow these guidelines:

- Simplicity
- Quick execution
- No visible side effects

Simplicity

Although it's possible to write complex template expressions, it's a better practice to avoid them.

A property name or method call should be the norm, but an occasional Boolean negation, `!`, is OK. Otherwise, confine application and business logic to the component, where it is easier to develop and test.

Quick execution

Angular executes template expressions after every change detection cycle. Change detection cycles are triggered by many asynchronous activities such as promise resolutions, HTTP results, timer events, key presses and mouse moves.

Expressions should finish quickly or the user experience may drag, especially on slower devices. Consider caching values when their computation is expensive.

No visible side effects

A template expression should not change any application state other than the value of the target property.

This rule is essential to Angular's "unidirectional data flow" policy. You should never worry that reading a component value might change some other displayed value. The view should be stable throughout a single rendering pass.

An idempotent expression is ideal because it is free of side effects and improves Angular's change detection performance. In Angular terms, an idempotent expression always returns *exactly the same thing* until one of its dependent values changes.

Dependent values should not change during a single turn of the event loop. If an idempotent expression returns a string or a number, it returns the same string or number when called twice in a row. If the expression returns an object, including an `array`, it returns the same object *reference* when called twice in a row.

> There is one exception to this behavior that applies to `*ngFor`. `*ngFor` has `trackBy` functionality that can deal with referential inequality of objects when iterating over them. See *ngFor with `trackBy` for details.

Template statements

A template **statement** responds to an **event** raised by a binding target such as an element, component, or directive. You'll see template statements in the event binding section, appearing in quotes to the right of the `=` symbol as in `(event)="statement"`.

src/app/app.component.html

```
<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 `;` or `,`).

However, certain JavaScript syntax is not allowed:

- `new`
- increment and decrement operators, `++` and `--`
- operator assignment, such as `+=` and `-=`
- the bitwise operators `|` and `&`
- the template expression operators

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

```
<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 (`let hero`), and a template reference variable (`#heroForm`) are passed to an event handling method of the component.

src/app/app.component.html

```
<button (click)="onSave($event)">Save</button>
<button *ngFor="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.

Binding syntax: an overview

Data-binding is a mechanism for coordinating what users see, specifically with application data values. While you could push values to and pull values from HTML, the application is easier to write, read, and maintain if you turn these tasks over to a binding framework. You simply declare bindings between binding sources, target HTML elements, and let the framework do the rest.

For a demonstration of the syntax and code snippets in this section, see the binding syntax example / download example.

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
	`{{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, 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.

```
<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

```
<!-- Bind button disabled 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 which has API docs for all the standard DOM elements and their properties. Comparing the `<td>` attributes attributes to the `<td>` properties 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".

```
<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 / download example 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.

```
<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:
>
> ```
> <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 / download example 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` 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'">`

Property binding `[property]`

Use property binding to *set* properties of target elements or directive `@Input()` decorators. For an example demonstrating all of the points in this section, see the property binding example / download example.

One-way in

Property binding flows a value in one direction, from a component's property into a target element property.

You can't use property binding to read or pull values out of target elements. Similarly, you cannot use property binding to call a method on the target element. If the element raises events, you can listen to them with an event binding.

If you must read a target element property or call one of its methods, see the API reference for ViewChild and ContentChild.

Examples

The most common property binding sets an element property to a component property value. An example is binding the `src` property of an image element to a component's `itemImageUrl` property:

src/app/app.component.html

```
<img [src]="itemImageUrl">
```

Here's an example of binding to the `colSpan` property. Notice that it's not `colspan`, which is the attribute, spelled with a lowercase `s`.

src/app/app.component.html

```
<!-- Notice the colSpan property is camel case -->
<tr><td [colSpan]="2">Span 2 columns</td></tr>
```

For more details, see the MDN HTMLTableCellElment documentation.

Another example is disabling a button when the component says that it `isUnchanged`:

src/app/app.component.html

```
<!-- Bind button disabled state to `isUnchanged` property -->
<button [disabled]="isUnchanged">Disabled Button</button>
```

Another is setting a property of a directive:

src/app/app.component.html

```
<p [ngClass]="classes">[ngClass] binding to the classes property making this
blue</p>
```

Yet another is setting the model property of a custom component—a great way for parent and child components to communicate:

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

Binding targets

An element property between enclosing square brackets identifies the target property. The target property in the following code is the image element's `src` property.

src/app/app.component.html

```
<img [src]="itemImageUrl">
```

There's also the `bind-` prefix alternative:

src/app/app.component.html

```
<img bind-src="itemImageUrl">
```

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

Element properties may be the more common targets, but Angular looks first to see if the name is a property of a known directive, as it is in the following example:

src/app/app.component.html

```
<p [ngClass]="classes">[ngClass] binding to the classes property making this blue</p>
```

Technically, Angular is matching the name to a directive `@Input()`, one of the property names listed in the directive's `inputs` array or a property decorated with `@Input()`. Such inputs map to the directive's own properties.

If the name fails to match a property of a known directive or element, Angular reports an “unknown directive” error.

> Though the target name is usually the name of a property, there is an automatic attribute-to-property mapping in Angular for several common attributes. These include `class`/`className`, `innerHtml`/`innerHTML`, and `tabindex`/`tabIndex`.

Avoid side effects

Evaluation of a template expression should have no visible side effects. The expression language itself, or the way you write template expressions, helps to a certain extent; you can't assign a value to anything in a property binding expression nor use the increment and decrement operators.

For example, you could have an expression that invoked a property or method that had side effects. The expression could call something like `getFoo()` where only you know what `getFoo()` does. If `getFoo()` changes something and you happen to be binding to that something, Angular may or may not display the changed value. Angular may detect the change and throw a warning error. As a best practice, stick to properties and to methods that return values and avoid side effects.

Return the proper type

The template expression should evaluate to the type of value that the target property expects. Return a string if the target property expects a string, a number if it expects a number, an object if it expects an object, and so on.

In the following example, the `childItem` property of the `ItemDetailComponent` expects a string, which is exactly what you're sending in the property binding:

src/app/app.component.html

```
<app-item-detail [childItem]="parentItem"></app-item-detail>
```

You can confirm this by looking in the `ItemDetailComponent` where the `@Input` type is set to a string:

src/app/item-detail/item-detail.component.ts (setting the @Input() type)

```
@Input() childItem: string;
```

As you can see here, the `parentItem` in `AppComponent` is a string, which the `ItemDetailComponent` expects:

src/app/app.component.ts

```
parentItem = 'lamp';
```

Passing in an object

The previous simple example showed passing in a string. To pass in an object, the syntax and thinking are the same.

In this scenario, `ListItemComponent` is nested within `AppComponent` and the `item` property expects an object.

src/app/app.component.html

```
<app-list-item [items]="currentItem"></app-list-item>
```

The `item` property is declared in the `ListItemComponent` with a type of `Item` and decorated with `@Input()`:

src/app/list-item.component.ts

```
@Input() items: Item[];
```

In this sample app, an `Item` is an object that has two properties; an `id` and a `name`.

src/app/item.ts

```
export class Item {
  id: number;
  name: string;
}
```

While a list of items exists in another file, `mock-items.ts`, you can specify a different item in `app.component.ts` so that the new item will render:

src/app.component.ts

```
currentItem = [{
  id: 21,
  name: 'phone'
}];
```

You just have to make sure, in this case, that you're supplying an object because that's the type of `item` and is what the nested component, `ListItemComponent`, expects.

In this example, `AppComponent` specifies a different `item` object (`currentItem`) and passes it to the nested `ListItemComponent`. `ListItemComponent` was able to use `currentItem` because it matches what an `Item` object is according to `item.ts`. The `item.ts` file is where `ListItemComponent` gets its definition of an `item`.

Remember the brackets

The brackets, `[]`, tell Angular to evaluate the template expression. If you omit the brackets, Angular treats the string as a constant and *initializes the target property* with that string:

src/app.component.html

```
<app-item-detail childItem="parentItem"></app-item-detail>
```

Omitting the brackets will render the string `parentItem`, not the value of `parentItem`.

One-time string initialization

You *should* omit the brackets when all of the following are true:

- The target property accepts a string value.
- The string is a fixed value that you can put directly into the template.
- This initial value never changes.

You routinely initialize attributes this way in standard HTML, and it works just as well for directive and component property initialization. The following example initializes the `prefix` property of the `StringInitComponent` to a fixed string, not a template expression. Angular sets it and forgets about it.

src/app/app.component.html

```
<app-string-init prefix="This is a one-time initialized string."></app-string-init>
```

The `[item]` binding, on the other hand, remains a live binding to the component's `currentItem` property.

Property binding vs. interpolation

You often have a choice between interpolation and property binding. The following binding pairs do the same thing:

src/app/app.component.html

```
<p><img src="{{itemImageUrl}}"> is the <i>interpolated</i> image.</p>
<p><img [src]="itemImageUrl"> is the <i>property bound</i> image.</p>

<p><span>"{{interpolationTitle}}" is the <i>interpolated</i> title.</span></p>
<p>"<span [innerHTML]="propertyTitle"></span>" is the <i>property bound</i> title.
</p>
```

Interpolation is a convenient alternative to property binding in many cases. When rendering data values as strings, there is no technical reason to prefer one form to the other, though readability tends to favor interpolation. However, *when setting an element property to a non-string data value, you must use property binding*.

Content security

Imagine the following malicious content.

src/app/app.component.ts

```
evilTitle = 'Template <script>alert("evil never sleeps")</script> Syntax';
```

In the component template, the content might be used with interpolation:

src/app/app.component.ts

```
<p><span>"{{evilTitle}}" is the <i>interpolated</i> evil title.</span></p>
```

Fortunately, Angular data binding is on alert for dangerous HTML. In the above case, the HTML displays as is, and the Javascript does not execute. Angular **does not** allow HTML with script tags to leak into the browser, neither with interpolation nor property binding.

In the following example, however, Angular sanitizes the values before displaying them.

src/app/app.component.html

```
<!--
 Angular generates a warning for the following line as it sanitizes them
 WARNING: sanitizing HTML stripped some content (see http://g.co/ng/security#xss).
-->
 <p>"<span [innerHTML]="evilTitle"></span>" is the <i>property bound</i> evil
title.</p>
```

Interpolation handles the `<script>` tags differently than property binding but both approaches render the content harmlessly. The following is the browser output of the `evilTitle` examples.

```
"Template  Syntax" is the interpolated evil title.
"Template alert("evil never sleeps")Syntax" is the property bound evil title.
```

Attribute, class, and style bindings

The template syntax provides specialized one-way bindings for scenarios less well-suited to property binding.

To see attribute, class, and style bindings in a functioning app, see the live example / download example especially for this section.

Attribute binding

Set the value of an attribute directly with an **attribute binding**. This is the only exception to the rule that a binding sets a target property and the only binding that creates and sets an attribute.

Usually, setting an element property with a property binding is preferable to setting the attribute with a string. However, sometimes there is no element property to bind, so attribute binding is the solution.

Consider the ARIA and SVG. They are purely attributes, don't correspond to element properties, and don't set element properties. In these cases, there are no property targets to bind to.

Attribute binding syntax resembles property binding, but instead of an element property between brackets, start with the prefix `attr`, followed by a dot (`.`), and the name of the attribute. You then set the attribute value, using an expression that resolves to a string, or remove the attribute when the expression resolves to `null`.

One of the primary use cases for attribute binding is to set ARIA attributes, as in this example:

src/app/app.component.html

```
<!-- create and set an aria attribute for assistive technology -->
<button [attr.aria-label]="actionName">{{actionName}} with Aria</button>
```

`colspan` and `colSpan`

Notice the difference between the `colspan` attribute and the `colSpan` property.

If you wrote something like this:

```
<tr><td colspan="{{1 + 1}}">Three-Four</td></tr>
```

You'd get this error:

```
Template parse errors:
Can't bind to 'colspan' since it isn't a known native property
```

As the message says, the `<td>` element does not have a `colspan` property. This is true because `colspan` is an attribute—`colSpan`, with a capital `S`, is the corresponding property. Interpolation and property binding can set only *properties*, not attributes.

Instead, you'd use property binding and write it like this:

src/app/app.component.html

```
<!-- Notice the colSpan property is camel case -->
<tr><td [colSpan]="1 + 1">Three-Four</td></tr>
```

Class binding

Add and remove CSS class names from an element's `class` attribute with a **class binding**.

Here's how to set the attribute without binding in plain HTML:

```
<!-- standard class attribute setting -->
<div class="item clearance special">Item clearance special</div>
```

Class binding syntax resembles property binding, but instead of an element property between brackets, start with the prefix `class`, optionally followed by a dot (`.`) and the name of a CSS class: `[class.class-name]`.

You can replace that with a binding to a string of the desired class names; this is an all-or-nothing, replacement binding.

src/app/app.component.html

```
<h3>Overwrite all existing classes with a new class:</h3>
<div class="item clearance special" [attr.class]="resetClasses">Reset all
classes at once</div>
```

You can also add append a class to an element without overwriting the classes already on the element:

src/app/app.component.html

```
<h3>Add a class:</h3>
<div class="item clearance special" [class.item-clearance]="itemClearance">Add
another class</div>
```

Finally, you can bind to a specific class name. Angular adds the class when the template expression evaluates to truthy. It removes the class when the expression is falsy.

src/app/app.component.html

```
<h3>toggle the "special" class on/off with a property:</h3>
<div [class.special]="isSpecial">The class binding is special.</div>

<h3>binding to class.special overrides the class attribute:</h3>
<div class="special" [class.special]="!isSpecial">This one is not so special.</div>

<h3>Using the bind- syntax:</h3>
<div bind-class.special="isSpecial">This class binding is special too.</div>
```

While this technique is suitable for toggling a single class name, consider the `NgClass` directive when managing multiple class names at the same time.

Style binding

You can set inline styles with a **style binding**.

Style binding syntax resembles property binding. Instead of an element property between brackets, start with the prefix `style`, followed by a dot (`.`) and the name of a CSS style property: `[style.style-property]`.

src/app/app.component.html

```
<button [style.color]="isSpecial ? 'red': 'green'">Red</button>
<button [style.background-color]="canSave ? 'cyan': 'grey'" >Save</button>
```

Some style binding styles have a unit extension. The following example conditionally sets the font size in “em” and “%” units .

src/app/app.component.html

```
<button [style.font-size.em]="isSpecial ? 3 : 1" >Big</button>
<button [style.font-size.%]="!isSpecial ? 150 : 50" >Small</button>
```

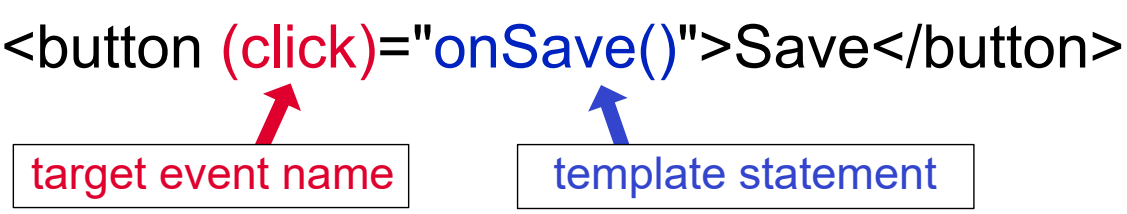
This technique is suitable for setting a single style, but consider the `NgStyle` directive when setting several inline styles at the same time.

> Note that a *style property* name can be written in either dash-case, as shown above, or camelCase, such as `fontSize`.

Event binding `(event)`

Event binding allows you to listen for certain events such as keystrokes, mouse movements, clicks, and touches. For an example demonstrating all of the points in this section, see the event binding example / download example.

Angular event binding syntax consists of a **target event** name within parentheses on the left of an equal sign, and a quoted template statement on the right. The following event binding listens for the button's click events, calling the component's `onSave()` method whenever a click occurs:



Target event

As above, the target is the button's click event.

src/app/app.component.html

```
<button (click)="onSave($event)">Save</button>
```

Alternatively, use the `on-` prefix, known as the canonical form:

src/app/app.component.html

```
<button on-click="onSave($event)">on-click Save</button>
```

Element events may be the more common targets, but Angular looks first to see if the name matches an event property of a known directive, as it does in the following example:

src/app/app.component.html

```
<h4>myClick is an event on the custom ClickDirective:</h4>
<button (myClick)="clickMessage=$event" clickable>click with myClick</button>
{{clickMessage}}
```

If the name fails to match an element event or an output property of a known directive, Angular reports an “unknown directive” error.

$event and event handling statements

In an event binding, Angular sets up an event handler for the target event.

When the event is raised, the handler executes the template statement. The template statement typically involves a receiver, which performs an action in response to the event, such as storing a value from the HTML control into a model.

The binding conveys information about the event. This information can include data values such as an event object, string, or number named `$event`.

The target event determines the shape of the `$event` object. If the target event is a native DOM element event, then `$event` is a DOM event object, with properties such as `target` and `target.value`.

Consider this example:

src/app/app.component.html

```
<input [value]="currentItem.name"
       (input)="currentItem.name=$event.target.value" >
without NgModel
```

This code sets the `<input>` `value` property by binding to the `name` property. To listen for changes to the value, the code binds to the `input` event of the `<input>` element. When the user makes changes, the `input` event is raised, and the binding executes the statement within a context that includes the DOM event object, `$event`.

To update the `name` property, the changed text is retrieved by following the path `$event.target.value`.

If the event belongs to a directive—recall that components are directives—`$event` has whatever shape the directive produces.

Custom events with `EventEmitter`

Directives typically raise custom events with an Angular EventEmitter. The directive creates an `EventEmitter` and exposes it as a property. The directive calls `EventEmitter.emit(payload)` to fire an event, passing in a message payload, which can be anything. Parent directives listen for the event by binding to this property and accessing the payload through the `$event` object.

Consider an `ItemDetailComponent` that presents item information and responds to user actions. Although the `ItemDetailComponent` has a delete button, it doesn't know how to delete the hero. It can only raise an event reporting the user's delete request.

Here are the pertinent excerpts from that `ItemDetailComponent`:

src/app/item-detail/item-detail.component.html (template)

```
<img src="{{itemImageUrl}}" [style.display]="displayNone">
<span [style.text-decoration]="lineThrough">{{ item.name }}
</span>
<button (click)="delete()">Delete</button>
```

src/app/item-detail/item-detail.component.ts (deleteRequest)

```
// This component makes a request but it can't actually delete a hero.
@Output() deleteRequest = new EventEmitter<Item>();

delete() {
  this.deleteRequest.emit(this.item);
  this.displayNone = this.displayNone ? '' : 'none';
  this.lineThrough = this.lineThrough ? '' : 'line-through';
}
```

The component defines a `deleteRequest` property that returns an `EventEmitter`. When the user clicks *delete*, the component invokes the `delete()` method, telling the `EventEmitter` to emit an `Item` object.

Now imagine a hosting parent component that binds to the `deleteRequest` event of the `ItemDetailComponent`.

src/app/app.component.html (event-binding-to-component)

```
<app-item-detail (deleteRequest)="deleteItem($event)" [item]="currentItem"></app-
item-detail>
```

When the `deleteRequest` event fires, Angular calls the parent component's `deleteItem()` method, passing the *item-to-delete* (emitted by `ItemDetail`) in the `$event` variable.

Template statements have side effects

Though template expressions shouldn't have side effects, template statements usually do. The `deleteItem()` method does have a side effect: it deletes an item.

Deleting an item updates the model, and depending on your code, triggers other changes including queries and saving to a remote server. These changes propagate through the system and ultimately display in this and other views.

Two-way binding `[(...)]`

Two-way binding gives your app a way to share data between a component class and its template.

For a demonstration of the syntax and code snippets in this section, see the two-way binding example / download example.

Basics of two-way binding

Two-way binding does two things:

1. Sets a specific element property.
2. Listens for an element change event.

Angular offers a special *two-way data binding* syntax for this purpose, `[()]`. The `[()]` syntax combines the brackets of property binding, `[]`, with the parentheses of event binding, `()`.

[( )] = BANANA IN A BOX

Visualize a *banana in a box* to remember that the parentheses go *inside* the brackets.

The `[()]` syntax is easy to demonstrate when the element has a settable property called `x` and a corresponding event named `xChange`. Here's a `SizerComponent` that fits this pattern. It has a `size` value property and a companion `sizeChange` event:

src/app/sizer.component.ts

```
import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-sizer',
  templateUrl: './sizer.component.html',
  styleUrls: ['./sizer.component.css']
})
export class SizerComponent {


  @Input()  size: number | string;
  @Output() sizeChange = new EventEmitter<number>();

  dec() { this.resize(-1); }
  inc() { this.resize(+1); }

  resize(delta: number) {
    this.size = Math.min(40, Math.max(8, +this.size + delta));
    this.sizeChange.emit(this.size);
  }

}
```

The initial `size` is an input value from a property binding. Clicking the buttons increases or decreases the `size`, within min/max value constraints, and then raises, or emits, the `sizeChange` event with the adjusted size.

Here's an example in which the `AppComponent.fontSizePx` is two-way bound to the `SizerComponent`:

src/app/app.component.html (two-way-1)

```
<app-sizer [(size)]="fontSizePx"></app-sizer>
<div [style.font-size.px]="fontSizePx">Resizable Text</div>
```

The `AppComponent.fontSizePx` establishes the initial `SizerComponent.size` value.

src/app/app.component.ts

```
fontSizePx = 16;
```

Clicking the buttons updates the `AppComponent.fontSizePx` via the two-way binding. The revised `AppComponent.fontSizePx` value flows through to the *style* binding, making the displayed text bigger or smaller.

The two-way binding syntax is really just syntactic sugar for a *property* binding and an *event* binding. Angular desugars the `SizerComponent` binding into this:

src/app/app.component.html (two-way-2)

```
<app-sizer [size]="fontSizePx" (sizeChange)="fontSizePx=$event"></app-sizer>
```

The `$event` variable contains the payload of the `SizerComponent.sizeChange` event. Angular assigns the `$event` value to the `AppComponent.fontSizePx` when the user clicks the buttons.

Two-way binding in forms

The two-way binding syntax is a great convenience compared to separate property and event bindings. It would be convenient to use two-way binding with HTML form elements like `<input>` and `<select>`. However, no native HTML element follows the `x` value and `xChange` event pattern.

For more on how to use two-way binding in forms, see Angular NgModel.

Built-in directives

Angular offers two kinds of built-in directives: attribute directives and structural directives. This segment reviews some of the most common built-in directives, classified as either *attribute* directives or *structural* directives and has its own built-in directives example / download example.

For more detail, including how to build your own custom directives, see Attribute Directives and Structural Directives.

Built-in attribute directives

Attribute directives listen to and modify the behavior of other HTML elements, attributes, properties, and components. You usually apply them to elements as if they were HTML attributes, hence the name.

Many NgModules such as the `RouterModule` and the `FormsModule` define their own attribute directives. The most common attribute directives are as follows:

- `NgClass`—adds and removes a set of CSS classes.
- `NgStyle`—adds and removes a set of HTML styles.
- `NgModel`—adds two-way data binding to an HTML form element.

NgClass

Add or remove several CSS classes simultaneously with `ngClass`.

src/app/app.component.html

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

> To add or remove a *single* class, use class binding rather than `NgClass`.

Consider a `setCurrentClasses()` component method that sets a component property, `currentClasses`, with an object that adds or removes three classes based on the `true`/`false` state of three other component properties. Each key of the object is a CSS class name; its value is `true` if the class should be added, `false` if it should be removed.

src/app/app.component.ts

```
currentClasses: {};
setCurrentClasses() {
  // CSS classes: added/removed per current state of component properties
  this.currentClasses =  {
    'saveable': this.canSave,
    'modified': !this.isUnchanged,
    'special':  this.isSpecial
  };
}
```

Adding an `ngClass` property binding to `currentClasses` sets the element's classes accordingly:

src/app/app.component.html

```
<div [ngClass]="currentClasses">This div is initially saveable, unchanged, and special.</div>
```

> Remember that in this situation you'd call `setCurrentClasses()`, both initially and when the dependent properties change.

`NgStyle`

Use `NgStyle` to set many inline styles simultaneously and dynamically, based on the state of the component.

Without `NgStyle`

For context, consider setting a *single* style value with style binding, without `NgStyle`.

src/app/app.component.html

```
<div [style.font-size]="isSpecial ? 'x-large' : 'smaller'">
  This div is x-large or smaller.
</div>
```

However, to set *many* inline styles at the same time, use the `NgStyle` directive.

The following is a `setCurrentStyles()` method that sets a component property, `currentStyles`, with an object that defines three styles, based on the state of three other component properties:

src/app/app.component.ts

```
currentStyles: {};
setCurrentStyles() {
  // CSS styles: set per current state of component properties
  this.currentStyles = {
    'font-style':  this.canSave      ? 'italic' : 'normal',
    'font-weight': !this.isUnchanged ? 'bold'   : 'normal',
    'font-size':   this.isSpecial    ? '24px'   : '12px'
  };
}
```

Adding an `ngStyle` property binding to `currentStyles` sets the element's styles accordingly:

src/app/app.component.html

```
<div [ngStyle]="currentStyles">
  This div is initially italic, normal weight, and extra large (24px).
</div>
```

> Remember to call `setCurrentStyles()`, both initially and when the dependent properties change.

`[(ngModel)]`: Two-way binding

The `NgModel` directive allows you to display a data property and update that property when the user makes changes. Here's an example:

src/app/app.component.html (NgModel example)

```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

Import `FormsModule` to use `ngModel`

Before using the `ngModel` directive in a two-way data binding, you must import the `FormsModule` and add it to the NgModule's `imports` list. Learn more about the `FormsModule` and `ngModel` in Forms.

Remember to import the `FormsModule` to make `[(ngModel)]` available as follows:

src/app/app.module.ts (FormsModule import)

```
import { FormsModule } from '@angular/forms'; // <--- JavaScript import from
Angular
/* . . . */
@NgModule({
/* . . . */

  imports: [
    BrowserModule,
    FormsModule // <--- import into the NgModule
  ],
/* . . . */
})
export class AppModule { }
```

You could achieve the same result with separate bindings to the `<input>` element's `value` property and `input` event:

src/app/app.component.html

```
<label for="without">without NgModel:</label>
<input [value]="currentItem.name" (input)="currentItem.name=$event.target.value"
id="without">
```

To streamline the syntax, the `ngModel` directive hides the details behind its own `ngModel` input and `ngModelChange` output properties:

src/app/app.component.html

```
<label for="example-change">(ngModelChange)="...name=$event":</label>
<input [ngModel]="currentItem.name" (ngModelChange)="currentItem.name=$event"
id="example-change">
```

The `ngModel` data property sets the element's value property and the `ngModelChange` event property listens for changes to the element's value.

`NgModel` and value accessors

The details are specific to each kind of element and therefore the `NgModel` directive only works for an element supported by a ControlValueAccessor that adapts an element to this protocol. Angular provides *value accessors* for all of the basic HTML form elements and the Forms guide shows how to bind to them.

You can't apply `[(ngModel)]` to a non-form native element or a third-party custom component until you write a suitable value accessor. For more information, see the API documentation on DefaultValueAccessor.

You don't need a value accessor for an Angular component that you write because you can name the value and event properties to suit Angular's basic two-way binding syntax and skip `NgModel` altogether. The `sizer` in the Two-way Binding section is an example of this technique.

Separate `ngModel` bindings are an improvement over binding to the element's native properties, but you can streamline the binding with a single declaration using the `[(ngModel)]` syntax:

src/app/app.component.html

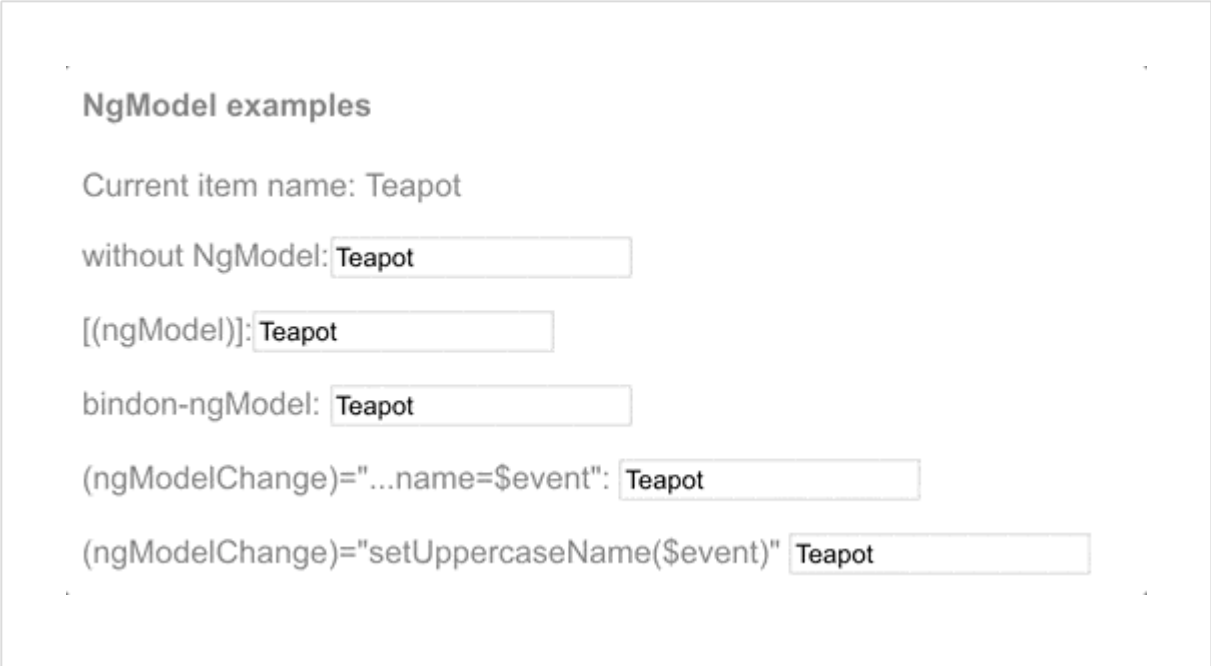
```
<label for="example-ngModel">[(ngModel)]:</label>
<input [(ngModel)]="currentItem.name" id="example-ngModel">
```

This `[(ngModel)]` syntax can only *set* a data-bound property. If you need to do something more, you can write the expanded form; for example, the following changes the `<input>` value to uppercase:

src/app/app.component.html

```
<input [ngModel]="currentItem.name" (ngModelChange)="setUppercaseName($event)"
id="example-uppercase">
```

Here are all variations in action, including the uppercase version:



Built-in *structural* directives

Structural directives are responsible for HTML layout. They shape or reshape the DOM's structure, typically by adding, removing, and manipulating the host elements to which they are attached.

This section is an introduction to the common built-in structural directives:

- `NgIf`—conditionally creates or destroys subviews from the template.
- `NgFor`—repeat a node for each item in a list.
- `NgSwitch`—a set of directives that switch among alternative views.

> The deep details of structural directives are covered in the Structural Directives guide, which explains the following:
>
> - Why you prefix the directive name with an asterisk (*).
> - Using `<ng-container>` to group elements when there is no suitable host element for the directive.
> - How to write your own structural directive.
> - That you can only apply one structural directive to an element.

NgIf

You can add or remove an element from the DOM by applying an `NgIf` directive to a host element. Bind the directive to a condition expression like `isActive` in this example.

src/app/app.component.html

```
<app-item-detail *ngIf="isActive" [item]="item"></app-item-detail>
```

> Don't forget the asterisk (`*`) in front of `ngIf`. For more information on the asterisk, see the asterisk (*) prefix section of Structural Directives.

When the `isActive` expression returns a truthy value, `NgIf` adds the `ItemDetailComponent` to the DOM. When the expression is falsy, `NgIf` removes the `ItemDetailComponent` from the DOM, destroying that component and all of its sub-components.

Show/hide vs. `NgIf`

Hiding an element is different from removing it with `NgIf`. For comparison, the following example shows how to control the visibility of an element with a class or style binding.

src/app/app.component.html

```
<!-- isSpecial is true -->
<div [class.hidden]="!isSpecial">Show with class</div>
<div [class.hidden]="isSpecial">Hide with class</div>

<p>ItemDetail is in the DOM but hidden</p>
<app-item-detail [class.hidden]="isSpecial"></app-item-detail>

<div [style.display]="isSpecial ? 'block' : 'none'">Show with style</div>
<div [style.display]="isSpecial ? 'none'  : 'block'">Hide with style</div>
```

When you hide an element, that element and all of its descendants remain in the DOM. All components for those elements stay in memory and Angular may continue to check for changes. You could be holding onto considerable computing resources and degrading performance unnecessarily.

`NgIf` works differently. When `NgIf` is `false`, Angular removes the element and its descendants from the DOM. It destroys their components, freeing up resources, which results in a better user experience.

If you are hiding large component trees, consider `NgIf` as a more efficient alternative to showing/hiding.

> For more information on `NgIf` and `ngIfElse`, see the API documentation about NgIf.

Guard against null

Another advantage of `ngIf` is that you can use it to guard against null. Show/hide is best suited for very simple use cases, so when you need a guard, opt instead for `ngIf`. Angular will throw an error if a nested expression tries to access a property of `null`.

The following shows `NgIf` guarding two `<div>`s. The `currentCustomer` name appears only when there is a `currentCustomer`. The `nullCustomer` will not be displayed as long as it is `null`.

src/app/app.component.html

```
<div *ngIf="currentCustomer">Hello, {{currentCustomer.name}}</div>
```

src/app/app.component.html

```
<div *ngIf="nullCustomer">Hello, <span>{{nullCustomer}}</span></div>
```

> See also the safe navigation operator below.

NgFor

`NgFor` is a repeater directive—a way to present a list of items. You define a block of HTML that defines how a single item should be displayed and then you tell Angular to use that block as a template for rendering each item in the list. The text assigned to `*ngFor` is the instruction that guides the repeater process.

The following example shows `NgFor` applied to a simple `<div>`. (Don't forget the asterisk (`*`) in front of `ngFor`.)

src/app/app.component.html

```
<div *ngFor="let item of items">{{item.name}}</div>
```

You can also apply an `NgFor` to a component element, as in the following example.

src/app/app.component.html

```
<app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

*NGFOR MICROSYNTAX

The string assigned to `*ngFor` is not a template expression. Rather, it's a *microsyntax*—a little language of its own that Angular interprets. The string `"let item of items"` means:

> *Take each item in the `items` array, store it in the local `item` looping variable, and make it available to the templated HTML for each iteration.*

Angular translates this instruction into an `<ng-template>` around the host element, then uses this template repeatedly to create a new set of elements and bindings for each `item` in the list. For more information about microsyntax, see the Structural Directives guide.

Template input variables

The `let` keyword before `item` creates a template input variable called `item`. The `ngFor` directive iterates over the `items` array returned by the parent component's `items` property and sets `item` to the current item from the array during each iteration.

Reference `item` within the `ngFor` host element as well as within its descendants to access the item's properties. The following example references `item` first in an interpolation and then passes in a binding to the `item` property of the `<app-item-detail>` component.

src/app/app.component.html

```
<div *ngFor="let item of items">{{item.name}}</div>
<!-- . . . -->
  <app-item-detail *ngFor="let item of items" [item]="item"></app-item-detail>
```

For more information about template input variables, see Structural Directives.

*ngFor with index

The `index` property of the `NgFor` directive context returns the zero-based index of the item in each iteration. You can capture the `index` in a template input variable and use it in the template.

The next example captures the `index` in a variable named `i` and displays it with the item name.

src/app/app.component.html

```
<div *ngFor="let item of items; let i=index">{{i + 1}} - {{item.name}}</div>
```

> `NgFor` is implemented by the `NgForOf` directive. Read more about the other `NgForOf` context values such as `last`, `even`, and `odd` in the NgForOf API reference.

*ngFor with trackBy

If you use `NgFor` with large lists, a small change to one item, such as removing or adding an item, can trigger a cascade of DOM manipulations. For example, re-querying the server could reset a list with all new item objects, even when those items were previously displayed. In this case, Angular sees only a fresh list of new object references and has no choice but to replace the old DOM elements with all new DOM elements.

You can make this more efficient with `trackBy`. Add a method to the component that returns the value `NgFor` should track. In this case, that value is the hero's `id`. If the `id` has already been rendered, Angular keeps track of it and doesn't re-query the server for the same `id`.

src/app/app.component.ts

```
trackByItems(index: number, item: Item): number { return item.id; }
```

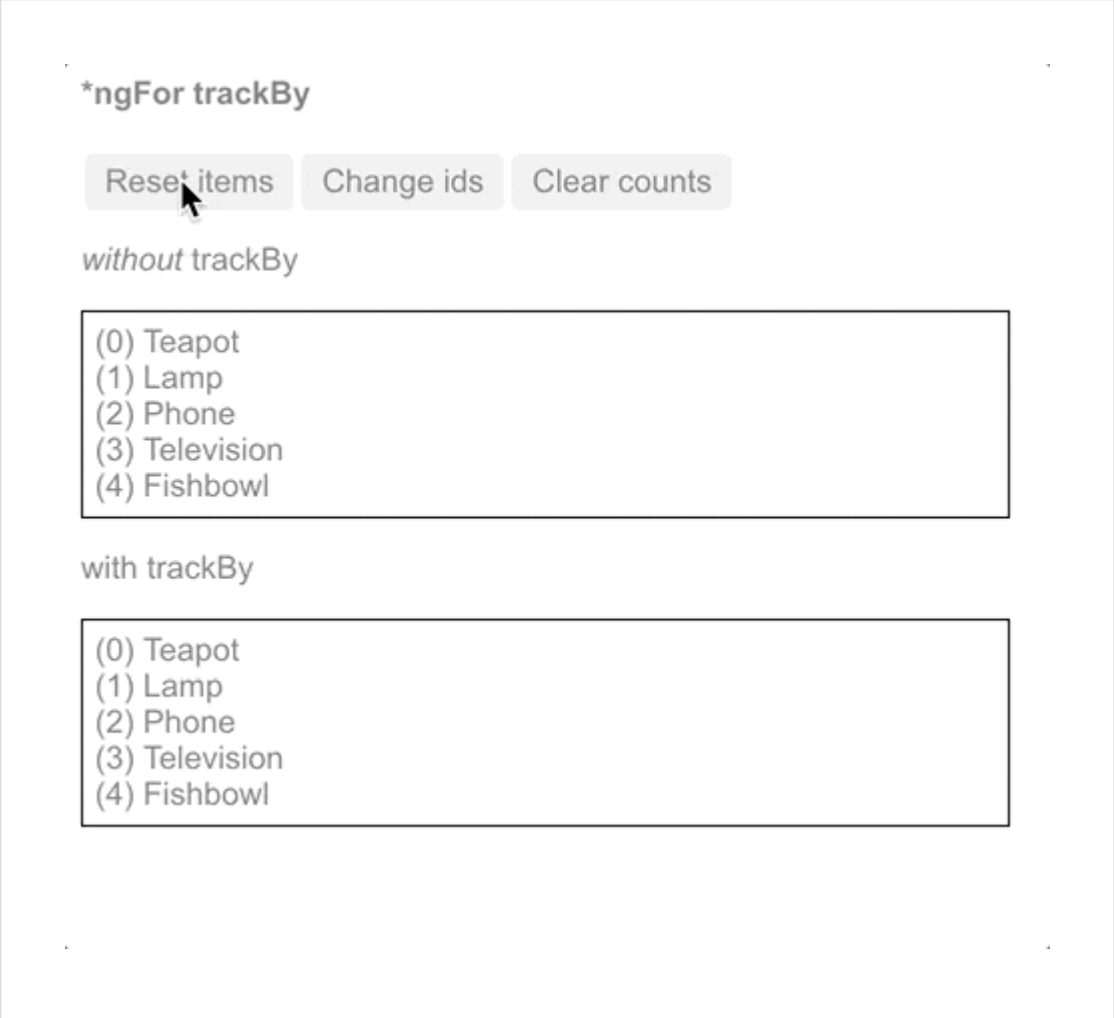
In the microsyntax expression, set `trackBy` to the `trackByItems()` method.

src/app/app.component.html

```
<div *ngFor="let item of items; trackBy: trackByItems">
  ({{item.id}}) {{item.name}}
</div>
```

Here is an illustration of the `trackBy` effect. "Reset items" creates new items with the same `item.id`s. "Change ids" creates new items with new `item.id`s.

- With no `trackBy`, both buttons trigger complete DOM element replacement.
- With `trackBy`, only changing the `id` triggers element replacement.



> Built-in directives use only public APIs; that is, they do not have special access to any private APIs that other directives can't access.

The `NgSwitch` directives

NgSwitch is like the JavaScript `switch` statement. It displays one element from among several possible elements, based on a switch condition. Angular puts only the selected element into the DOM.

`NgSwitch` is actually a set of three, cooperating directives: `NgSwitch`, `NgSwitchCase`, and `NgSwitchDefault` as in the following example.

src/app/app.component.html

```
<div [ngSwitch]="currentItem.feature">
  <app-stout-item    *ngSwitchCase="'stout'"    [item]="currentItem"></app-
stout-item>
  <app-device-item   *ngSwitchCase="'slim'"     [item]="currentItem"></app-
device-item>
  <app-lost-item     *ngSwitchCase="'vintage'"  [item]="currentItem"></app-lost-
item>
  <app-best-item     *ngSwitchCase="'bright'"   [item]="currentItem"></app-best-
item>
<!-- . . . -->
  <app-unknown-item  *ngSwitchDefault           [item]="currentItem"></app-
unknown-item>
</div>
```

`NgSwitch` is the controller directive. Bind it to an expression that returns the *switch value*, such as `feature`. Though the `feature` value in this example is a string, the switch value can be of any type.

Bind to `[ngSwitch]`. You'll get an error if you try to set `*ngSwitch` because `NgSwitch` is an *attribute* directive, not a *structural* directive. Rather than touching the DOM directly, it changes the behavior of its companion directives.

Bind to `*ngSwitchCase` **and** `*ngSwitchDefault`. The `NgSwitchCase` and `NgSwitchDefault` directives are *structural* directives because they add or remove elements from the DOM.

- `NgSwitchCase` adds its element to the DOM when its bound value equals the switch value and removes its bound value when it doesn't equal the switch value.
- `NgSwitchDefault` adds its element to the DOM when there is no selected `NgSwitchCase`.

The switch directives are particularly useful for adding and removing *component elements*. This example switches among four `item` components defined in the `item-switch.components.ts` file. Each component has an `item` input property which is bound to the `currentItem` of the parent component.

Switch directives work as well with native elements and web components too. For example, you could replace the `<app-best-item>` switch case with the following.

src/app/app.component.html

```
<div *ngSwitchCase="'bright'"> Are you as bright as {{currentItem.name}}?</div>
```

Template reference variables (#var)

A **template reference variable** is often a reference to a DOM element within a template. It can also refer to a directive (which contains a component), an element, TemplateRef, or a web component.

For a demonstration of the syntax and code snippets in this section, see the template reference variables example / download example.

Use the hash symbol (#) to declare a reference variable. The following reference variable, `#phone`, declares a `phone` variable on an `<input>` element.

src/app/app.component.html

```
<input #phone placeholder="phone number" />
```

You can refer to a template reference variable anywhere in the component's template. Here, a `<button>` further down the template refers to the `phone` variable.

src/app/app.component.html

```
<input #phone placeholder="phone number" />

<!-- lots of other elements -->

<!-- phone refers to the input element; pass its `value` to an event handler -->
<button (click)="callPhone(phone.value)">Call</button>
```

How a reference variable gets its value

In most cases, Angular sets the reference variable's value to the element on which it is declared. In the previous example, `phone` refers to the phone number `<input>`. The button's click handler passes the `<input>` value to the component's `callPhone()` method.

The `NgForm` directive can change that behavior and set the value to something else. In the following example, the template reference variable, `itemForm`, appears three times separated by HTML.

src/app/hero-form.component.html

```
<form #itemForm="ngForm" (ngSubmit)="onSubmit(itemForm)">
  <label for="name"
    >Name <input class="form-control" name="name" ngModel required />
  </label>
  <button type="submit">Submit</button>
</form>

<div [hidden]="!itemForm.form.valid">
  <p>{{ submitMessage }}</p>
</div>
```

The reference value of itemForm, without the ngForm attribute value, would be the HTMLFormElement. There is, however, a difference between a Component and a Directive in that a `Component` will be referenced without specifying the attribute value, and a `Directive` will not change the implicit reference (that is, the element).

However, with `NgForm`, `itemForm` is a reference to the NgForm directive with the ability to track the value and validity of every control in the form.

The native `<form>` element doesn't have a `form` property, but the `NgForm` directive does, which allows disabling the submit button if the `itemForm.form.valid` is invalid and passing the entire form control tree to the parent component's `onSubmit()` method.

Template reference variable considerations

A template *reference* variable (`#phone`) is not the same as a template *input* variable (`let phone`) such as in an `*ngFor`. See *Structural Directives* for more information.

The scope of a reference variable is the entire template. So, don't define the same variable name more than once in the same template as the runtime value will be unpredictable.

Alternative syntax

You can use the `ref-` prefix alternative to `#`. This example declares the `fax` variable as `ref-fax` instead of `#fax`.

src/app/app.component.html

```
<input ref-fax placeholder="fax number" />
<button (click)="callFax(fax.value)">Fax</button>
```

`@Input()` and `@Output()` properties

`@Input()` and `@Output()` allow Angular to share data between the parent context and child directives or components. An `@Input()` property is writable while an `@Output()` property is observable.

Consider this example of a child/parent relationship:

```
<parent-component>
  <child-component></child-component>
</parent-component>
```

Here, the `<child-component>` selector, or child directive, is embedded within a `<parent-component>`, which serves as the child's context.

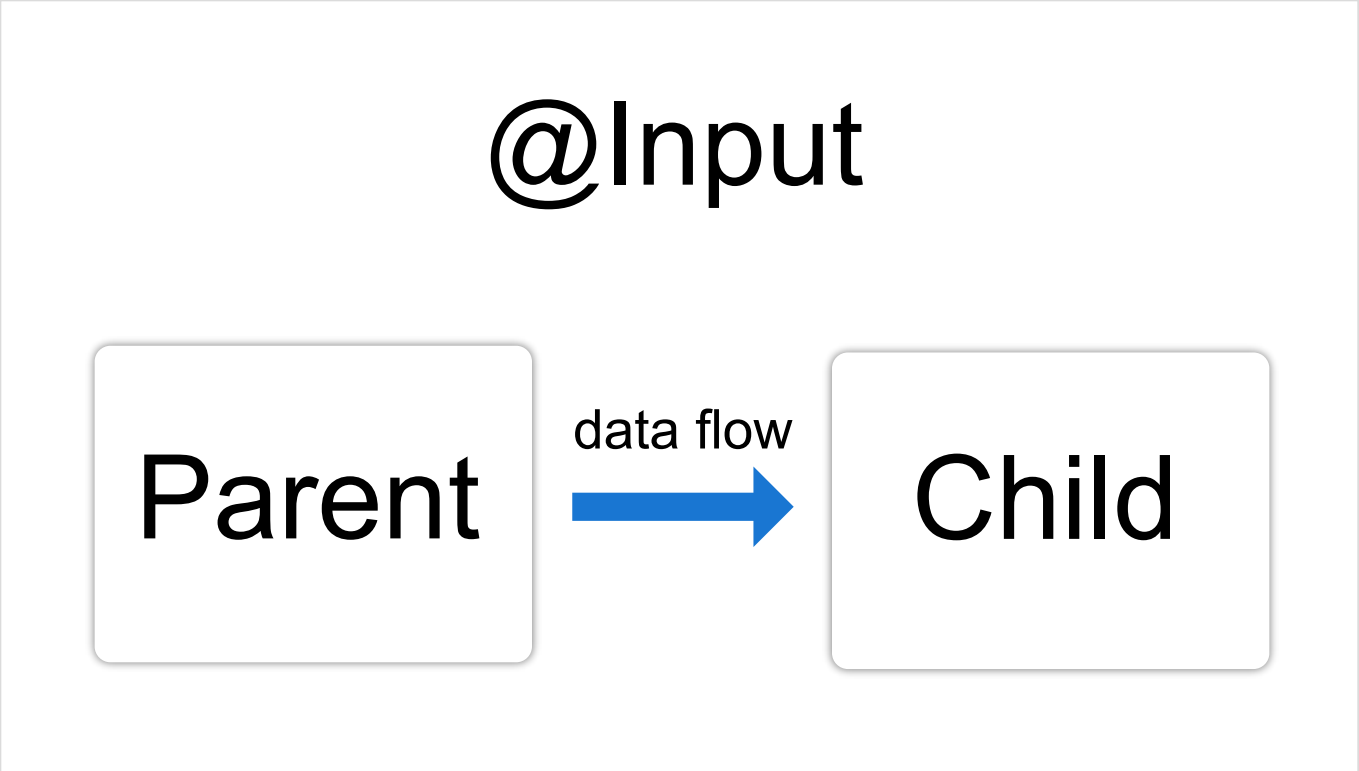
`@Input()` and `@Output()` act as the API, or application programming interface, of the child component in that they allow the child to communicate with the parent. Think of `@Input()` and `@Output()` like ports or doorways—`@Input()` is the doorway into the component allowing data to flow in while `@Output()` is the doorway out of the component, allowing the child component to send data out.

This section about `@Input()` and `@Output()` has its own live example / download example. The following subsections highlight key points in the sample app.

> ### `@Input()` and `@Output()` are independent
>
> Though `@Input()` and `@Output()` often appear together in apps, you can use them separately. If the nested component is such that it only needs to send data to its parent, you wouldn't need an `@Input()`, only an `@Output()`. The reverse is also true in that if the child only needs to receive data from the parent, you'd only need `@Input()`.

How to use `@Input()`

Use the `@Input()` decorator in a child component or directive to let Angular know that a property in that component can receive its value from its parent component. It helps to remember that the data flow is from the perspective of the child component. So an `@Input()` allows data to be input *into* the child component from the parent component.



To illustrate the use of `@Input()`, edit these parts of your app:

- The child component class and template
- The parent component class and template

In the child

To use the `@Input()` decorator in a child component class, first import `Input` and then decorate the property with `@Input()`:

src/app/item-detail/item-detail.component.ts

```
import { Component, Input } from '@angular/core'; // First, import Input
export class ItemDetailComponent {
  @Input() item: string; // decorate the property with @Input()
}
```

In this case, `@Input()` decorates the property `item`, which has a type of `string`, however, `@Input()` properties can have any type, such as `number`, `string`, `boolean`, or `object`. The value for `item` will come from the parent component, which the next section covers.

Next, in the child component template, add the following:

src/app/item-detail/item-detail.component.html

```
<p>
  Today's item: {{item}}
</p>
```

In the parent

The next step is to bind the property in the parent component's template. In this example, the parent component template is `app.component.html`.

First, use the child's selector, here `<app-item-detail>`, as a directive within the parent component template. Then, use property binding to bind the property in the child to the property of the parent.

src/app/app.component.html

```
<app-item-detail [item]="currentItem"></app-item-detail>
```

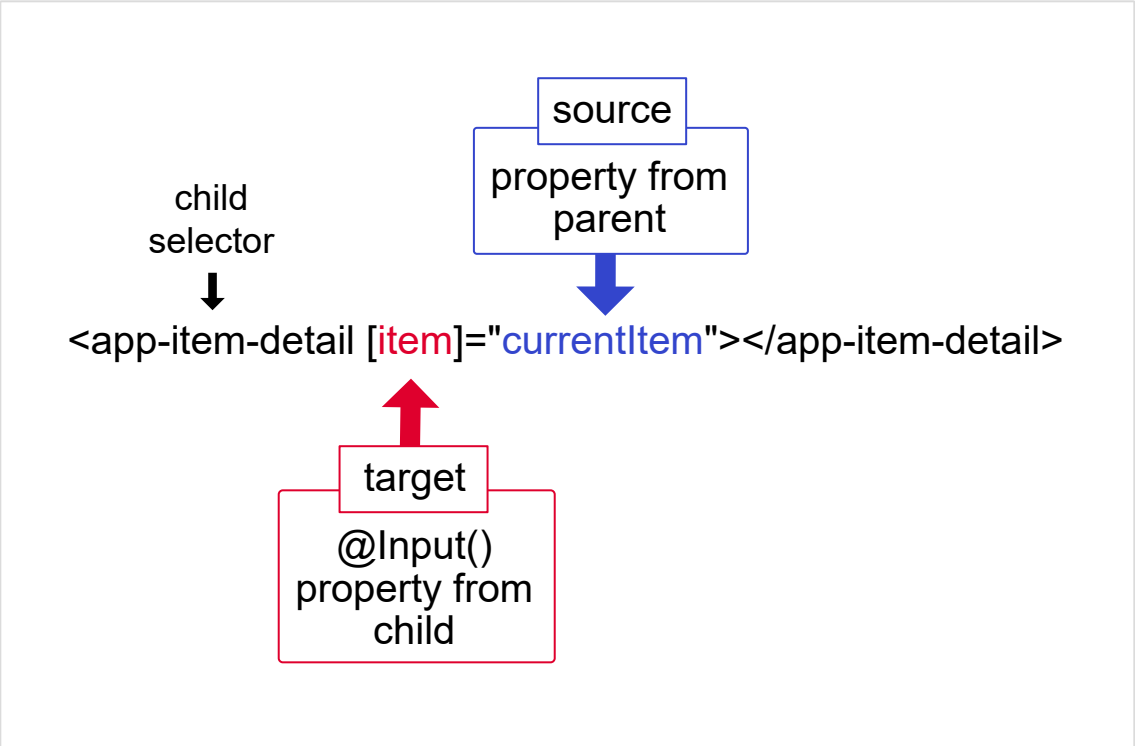
Next, in the parent component class, `app.component.ts`, designate a value for `currentItem`:

src/app/app.component.ts

```
export class AppComponent {
  currentItem = 'Television';
}
```

With `@Input()`, Angular passes the value for `currentItem` to the child so that `item` renders as `Television`.

The following diagram shows this structure:



The target in the square brackets, `[]`, is the property you decorate with `@Input()` in the child component. The binding source, the part to the right of the equal sign, is the data that the parent component passes to the nested component.

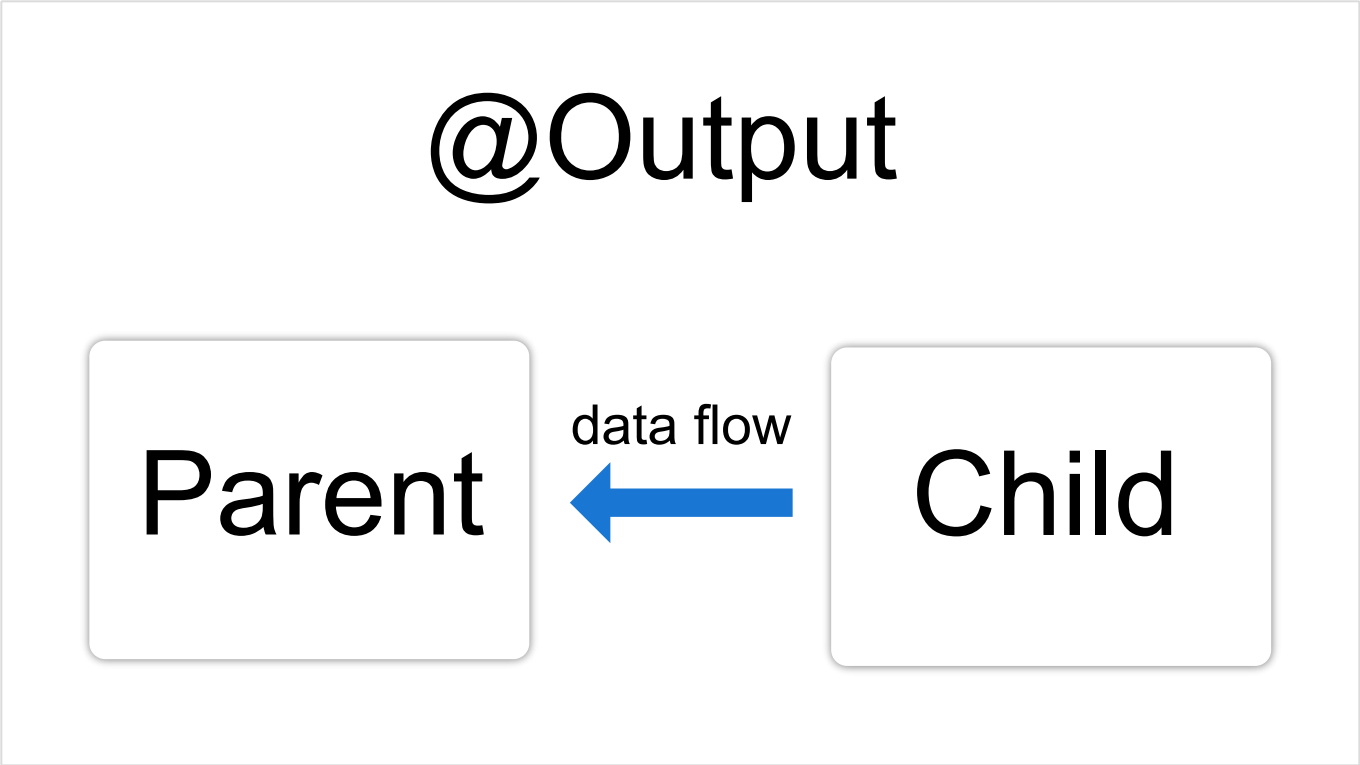
The key takeaway is that when binding to a child component's property in a parent component—that is, what's in square brackets—you must decorate the property with `@Input()` in the child component.

> ### `OnChanges` and `@Input()`
>
> To watch for changes on an `@Input()` property, use `OnChanges`, one of Angular's lifecycle hooks. `OnChanges` is specifically designed to work with properties that have the `@Input()` decorator. See the `OnChanges` section of the Lifecycle Hooks guide for more details and examples.

How to use `@Output()`

Use the `@Output()` decorator in the child component or directive to allow data to flow from the child *out* to the parent.

An `@Output()` property should normally be initialized to an Angular `EventEmitter` with values flowing out of the component as events.



Just like with `@Input()`, you can use `@Output()` on a property of the child component but its type should be `EventEmitter`.

`@Output()` marks a property in a child component as a doorway through which data can travel from the child to the parent. The child component then has to raise an event so the parent knows something has changed. To raise an event, `@Output()` works hand in hand with `EventEmitter`, which is a class in `@angular/core` that you use to emit custom events.

When you use `@Output()`, edit these parts of your app:

- The child component class and template
- The parent component class and template

The following example shows how to set up an `@Output()` in a child component that pushes data you enter in an HTML `<input>` to an array in the parent component.

> The HTML element `<input>` and the Angular decorator `@Input()` are different. This documentation is about component communication in Angular as it pertains to `@Input()` and `@Output()`. For more information on the HTML element `<input>`, see the W3C Recommendation.

In the child

This example features an `<input>` where a user can enter a value and click a `<button>` that raises an event. The `EventEmitter` then relays the data to the parent component.

First, be sure to import `Output` and `EventEmitter` in the child component class:

```
import { Output, EventEmitter } from '@angular/core';
```

Next, still in the child, decorate a property with `@Output()` in the component class. The following example `@Output()` is called `newItemEvent` and its type is `EventEmitter`, which means it's an event.

src/app/item-output/item-output.component.ts

```
@Output() newItemEvent = new EventEmitter<string>();
```

The different parts of the above declaration are as follows:

- `@Output()`—a decorator function marking the property as a way for data to go from the child to the parent
- `newItemEvent`—the name of the `@Output()`
- `EventEmitter<string>`—the `@Output()`'s type
- `new EventEmitter<string>()`—tells Angular to create a new event emitter and that the data it emits is of type string. The type could be any type, such as `number`, `boolean`, and so on. For more information on `EventEmitter`, see the EventEmitter API documentation.

Next, create an `addNewItem()` method in the same component class:

src/app/item-output/item-output.component.ts

```
export class ItemOutputComponent {

  @Output() newItemEvent = new EventEmitter<string>();

  addNewItem(value: string) {
    this.newItemEvent.emit(value);
  }
}
```

The `addNewItem()` function uses the `@Output()`, `newItemEvent`, to raise an event in which it emits the value the user types into the `<input>`. In other words, when the user clicks the add button in the UI, the child lets the parent know about the event and gives that data to the parent.

In the child's template

The child's template has two controls. The first is an HTML `<input>` with a template reference variable , `#newItem`, where the user types in an item name. Whatever the user types into the `<input>` gets stored in the `#newItem` variable.

src/app/item-output/item-output.component.html

```
<label>Add an item: <input #newItem></label>
<button (click)="addNewItem(newItem.value)">Add to parent's list</button>
```

The second element is a `<button>` with an event binding. You know it's an event binding because the part to the left of the equal sign is in parentheses, `(click)`.

The `(click)` event is bound to the `addNewItem()` method in the child component class which takes as its argument whatever the value of `#newItem` is.

Now the child component has an `@Output()` for sending data to the parent and a method for raising an event. The next step is in the parent.

In the parent

In this example, the parent component is `AppComponent`, but you could use any component in which you could nest the child.

The `AppComponent` in this example features a list of `items` in an array and a method for adding more items to the array.

src/app/app.component.ts

```
export class AppComponent {
  items = ['item1', 'item2', 'item3', 'item4'];

  addItem(newItem: string) {
    this.items.push(newItem);
  }
}
```

The `addItem()` method takes an argument in the form of a string and then pushes, or adds, that string to the `items` array.

In the parent's template

Next, in the parent's template, bind the parent's method to the child's event. Put the child selector, here `<app-item-output>`, within the parent component's template, `app.component.html`.

src/app/app.component.html

```
<app-item-output (newItemEvent)="addItem($event)"></app-item-output>
```

The event binding, `(newItemEvent)='addItem($event)'`, tells Angular to connect the event in the child, `newItemEvent`, to the method in the parent, `addItem()`, and that the event that the child is notifying the parent about is to be the argument of `addItem()`. In other words, this is where the actual hand off of data takes place. The `$event` contains the data that the user types into the `<input>` in the child template UI.

Now, in order to see the `@Output()` working, add the following to the parent's template:

```
<ul>
  <li *ngFor="let item of items">{{item}}</li>
</ul>
```

The `*ngFor` iterates over the items in the `items` array. When you enter a value in the child's `<input>` and click the button, the child emits the event and the parent's `addItem()` method pushes the value to the `items` array and it renders in the list.

`@Input()` and `@Output()` together

You can use `@Input()` and `@Output()` on the same child component as in the following:

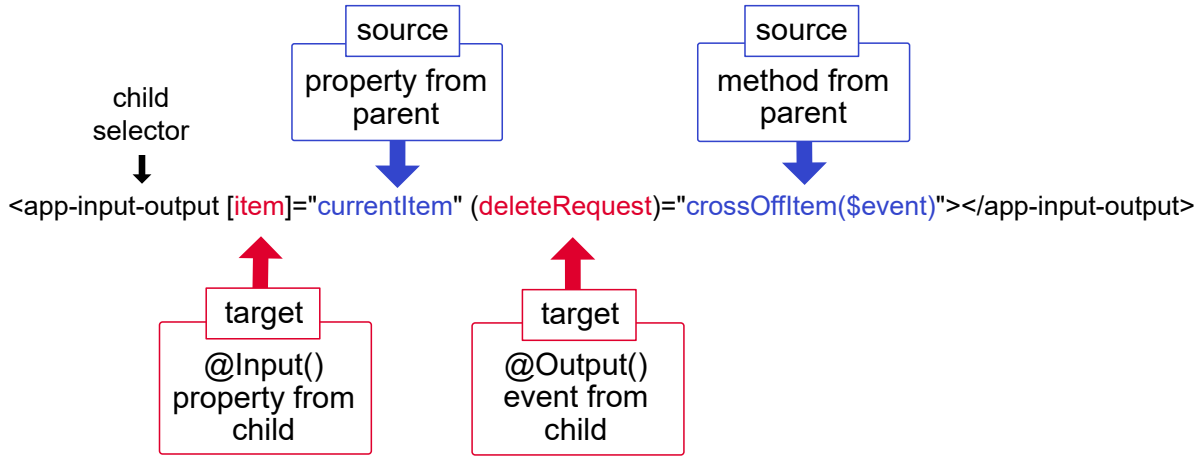
src/app/app.component.html

```
<app-input-output [item]="currentItem" (deleteRequest)="crossOffItem($event)">
</app-input-output>
```

The target, `item`, which is an `@Input()` property in the child component class, receives its value from the parent's property, `currentItem`. When you click delete, the child component raises an event,

`deleteRequest`, which is the argument for the parent's `crossOffItem()` method.

The following diagram is of an `@Input()` and an `@Output()` on the same child component and shows the different parts of each:



As the diagram shows, use inputs and outputs together in the same manner as using them separately. Here, the child selector is `<app-input-output>` with `item` and `deleteRequest` being `@Input()` and `@Output()` properties in the child component class. The property `currentItem` and the method `crossOffItem()` are both in the parent component class.

To combine property and event bindings using the banana-in-a-box syntax, `[()]`, see Two-way Binding.

For more detail on how these work, see the previous sections on Input and Output. To see it in action, see the Inputs and Outputs Example / download example.

`@Input()` and `@Output()` declarations

Instead of using the `@Input()` and `@Output()` decorators to declare inputs and outputs, you can identify members in the `inputs` and `outputs` arrays of the directive metadata, as in this example:

src/app/in-the-metadata/in-the-metadata.component.ts

```
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property
inputs: ['clearanceItem'],
outputs: ['buyEvent']
// tslint:enable: no-inputs-metadata-property no-outputs-metadata-property
```

While declaring `inputs` and `outputs` in the `@Directive` and `@Component` metadata is possible, it is a better practice to use the `@Input()` and `@Output()` class decorators instead, as follows:

src/app/input-output/input-output.component.ts

```
@Input() item: string;
@Output() deleteRequest = new EventEmitter<string>();
```

See the Decorate input and output properties section of the Style Guide for details.

If you get a template parse error when trying to use inputs or outputs, but you know that the properties do indeed exist, double check that your properties are annotated with `@Input()` / `@Output()` or that you've declared them in an `inputs`/`outputs` array:

```
Uncaught Error: Template parse errors:
Can't bind to 'item' since it isn't a known property of 'app-item-detail'
```

Aliasing inputs and outputs

Sometimes the public name of an input/output property should be different from the internal name. While it is a best practice to avoid this situation, Angular does offer a solution.

Aliasing in the metadata

Alias inputs and outputs in the metadata using a colon-delimited (`:`) string with the directive property name on the left and the public alias on the right:

src/app/aliasing/aliasing.component.ts

```
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property
inputs: ['input1: saveForLaterItem'], // propertyName:alias
outputs: ['outputEvent1: saveForLaterEvent']
// tslint:disable: no-inputs-metadata-property no-outputs-metadata-property
```

Aliasing with the `@Input()`/`@Output()` decorator

You can specify the alias for the property name by passing the alias name to the `@Input()`/`@Output()` decorator. The internal name remains as usual.

src/app/aliasing/aliasing.component.ts

```
@Input('wishListItem') input2: string; //  @Input(alias)
@Output('wishEvent') outputEvent2 = new EventEmitter<string>(); //  @Output(alias)
propertyName = ...
```

Template expression operators

The Angular template expression language employs a subset of JavaScript syntax supplemented with a few special operators for specific scenarios. The next sections cover three of these operators:

- pipe
- safe navigation operator
- non-null assertion operator

The pipe operator (|)

The result of an expression might require some transformation before you're ready to use it in a binding. For example, you might display a number as a currency, change text to uppercase, or filter a list and sort it.

Pipes are simple functions that accept an input value and return a transformed value. They're easy to apply within template expressions, using the pipe operator (|):

src/app/app.component.html

```
<p>Title through uppercase pipe: {{title | uppercase}}</p>
```

The pipe operator passes the result of an expression on the left to a pipe function on the right.

You can chain expressions through multiple pipes:

src/app/app.component.html

```
<!-- convert title to uppercase, then to lowercase -->
<p>Title through a pipe chain: {{title | uppercase | lowercase}}</p>
```

And you can also apply parameters to a pipe:

src/app/app.component.html

```
<!-- pipe with configuration argument => "February 25, 1980" -->
<p>Manufacture date with date format pipe: {{item.manufactureDate |
date:'longDate'}}</p>
```

The `json` pipe is particularly helpful for debugging bindings:

src/app/app.component.html

```
<p>Item json pipe: {{item | json}}</p>
```

The generated output would look something like this:

```
{ "name": "Telephone",
  "manufactureDate": "1980-02-25T05:00:00.000Z",
  "price": 98 }
```

> The pipe operator has a higher precedence than the ternary operator (`?:`), which means `a ? b : c | x` is parsed as `a ? b : (c | x)`. Nevertheless, for a number of reasons, the pipe operator cannot be used without parentheses in the first and second operands of `?:`. A good practice is to use parentheses in the third operand too.

The safe navigation operator (`?`) and null property paths

The Angular safe navigation operator, `?`, guards against `null` and `undefined` values in property paths. Here, it protects against a view render failure if `item` is `null`.

src/app/app.component.html

```
<p>The item name is: {{item?.name}}</p>
```

If `item` is `null`, the view still renders but the displayed value is blank; you see only "The item name is:" with nothing after it.

Consider the next example, with a `nullItem`.

```
The null item name is {{nullItem.name}}
```

Since there is no safe navigation operator and `nullItem` is `null`, JavaScript and Angular would throw a `null` reference error and break the rendering process of Angular:

```
TypeError: Cannot read property 'name' of null.
```

Sometimes however, `null` values in the property path may be OK under certain circumstances, especially when the value starts out null but the data arrives eventually.

With the safe navigation operator, `?`, Angular stops evaluating the expression when it hits the first `null` value and renders the view without errors.

It works perfectly with long property paths such as `a?.b?.c?.d`.

The non-null assertion operator (`!`)

As of Typescript 2.0, you can enforce strict null checking with the `--strictNullChecks` flag. TypeScript then ensures that no variable is unintentionally null or undefined.

In this mode, typed variables disallow `null` and `undefined` by default. The type checker throws an error if you leave a variable unassigned or try to assign `null` or `undefined` to a variable whose type disallows `null` and `undefined`.

The type checker also throws an error if it can't determine whether a variable will be `null` or undefined at runtime. You tell the type checker not to throw an error by applying the postfix non-null assertion operator, !.

The Angular non-null assertion operator, `!`, serves the same purpose in an Angular template. For example, after you use *ngIf to check that `item` is defined, you can assert that `item` properties are also

defined.

src/app/app.component.html

```
<!--No color, no error -->
<p *ngIf="item">The item's color is: {{item!.color}}</p>
```

When the Angular compiler turns your template into TypeScript code, it prevents TypeScript from reporting that `item` might be `null` or `undefined`.

Unlike the *safe navigation operator*, the non-null assertion operator does not guard against `null` or `undefined`. Rather, it tells the TypeScript type checker to suspend strict `null` checks for a specific property expression.

The non-null assertion operator, `!`, is optional with the exception that you must use it when you turn on strict null checks.

back to top

Built-in template functions

The `$any()` type cast function

Sometimes a binding expression triggers a type error during AOT compilation and it is not possible or difficult to fully specify the type. To silence the error, you can use the `$any()` cast function to cast the expression to the `any` type as in the following example:

src/app/app.component.html

```
<p>The item's undeclared best by date is: {{$any(item).bestByDate}}</p>
```

When the Angular compiler turns this template into TypeScript code, it prevents TypeScript from reporting that `bestByDate` is not a member of the `item` object when it runs type checking on the template.

The `$any()` cast function also works with `this` to allow access to undeclared members of the component.

src/app/app.component.html

```
<p>The item's undeclared best by date is: {{$any(this).bestByDate}}</p>
```

The `$any()` cast function works anywhere in a binding expression where a method call is valid.

SVG in templates

It is possible to use SVG as valid templates in Angular. All of the template syntax below is applicable to both SVG and HTML. Learn more in the SVG 1.1 and 2.0 specifications.

Why would you use SVG as template, instead of simply adding it as image to your application?

When you use an SVG as the template, you are able to use directives and bindings just like with HTML templates. This means that you will be able to dynamically generate interactive graphics.

Refer to the sample code snippet below for a syntax example:

src/app/svg.component.ts

```
import { Component } from '@angular/core';

@Component({
  selector: 'app-svg',
  templateUrl: './svg.component.svg',
  styleUrls: ['./svg.component.css']
})
export class SvgComponent {
  fillColor = 'rgb(255, 0, 0)';

  changeColor() {
    const r = Math.floor(Math.random() * 256);
    const g = Math.floor(Math.random() * 256);
    const b = Math.floor(Math.random() * 256);
    this.fillColor = `rgb(${r}, ${g}, ${b})`;
  }
}
```

Add the following code to your `svg.component.svg` file:

src/app/svg.component.svg

```
<svg>
  <g>
    <rect x="0" y="0" width="100" height="100" [attr.fill]="fillColor"
(click)="changeColor()" />
    <text x="120" y="50">click the rectangle to change the fill color</text>
  </g>
</svg>
```

Here you can see the use of a `click()` event binding and the property binding syntax (`[attr.fill]="fillColor"`).