**Input**

decorator

Decorator that marks a class field as an input property and supplies configuration metadata. The input property is bound to a DOM property in the template. During change detection, Angular automatically updates the data property with the DOM property's value.

| **Option** | **Description** |
| --- | --- |
| [bindingPropertyName](https://angular.io/api/core/Input#bindingPropertyName) | The name of the DOM property to which the input property is bound. |

**See also**

* [Input and Output properties](https://angular.io/guide/template-syntax#input-and-output-properties)

**Options**

| **bindingPropertyName** |
| --- |

| [***mode\_edit***](https://github.com/angular/angular/edit/master/packages/core/src/metadata/directives.ts?message=docs(core)%3A%20describe%20your%20change...#L697-L700)[***code***](https://github.com/angular/angular/tree/8.2.14/packages/core/src/metadata/directives.ts#L697-L700) |
| --- |
| The name of the DOM property to which the input property is bound. |
| bindingPropertyName: string |
|  |

**Usage notes**

You can supply an optional name to use in templates when the component is instantiated, that maps to the name of the bound property. By default, the original name of the bound property is used for input binding.

The following example creates a component with two input properties, one of which is given a special binding name.

@[Component](https://angular.io/api/core/Component)({

selector: 'bank-account',

template: `

Bank Name: {{bankName}}

Account Id: {{id}}

`

})

class BankAccount {

// This property is bound using its original name.

@[Input](https://angular.io/api/core/Input)() bankName: string;

// this property value is bound to [a](https://angular.io/api/router/RouterLinkWithHref) different property name

// when this component is instantiated in [a](https://angular.io/api/router/RouterLinkWithHref) template.

@[Input](https://angular.io/api/core/Input)('account-id') id: string;

// this property is not bound, and is not automatically updated by Angular

normalizedBankName: string;

}

@[Component](https://angular.io/api/core/Component)({

selector: 'app',

template: `

<bank-account bankName="RBC" account-id="4747"></bank-account>

`

})

class App {}

## @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() properties

@[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() allow Angular to share data between the parent context and child directives or components. An @[Input](https://angular.io/api/core/Input)() property is writable while an @[Output](https://angular.io/api/core/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](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/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](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() like ports or doorways—@[Input](https://angular.io/api/core/Input)() is the doorway into the component allowing data to flow in while @[Output](https://angular.io/api/core/Output)() is the doorway out of the component, allowing the child component to send data out.

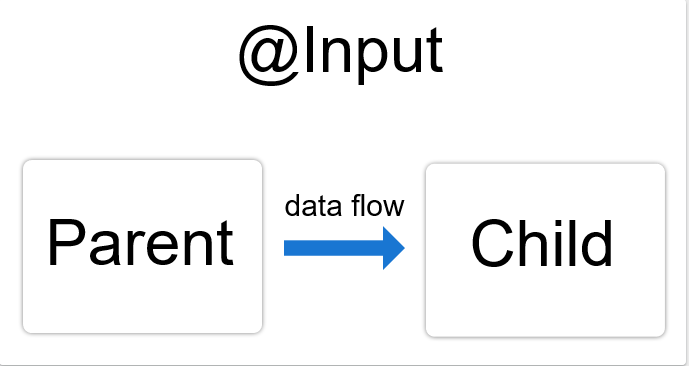
This section about @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() has its own [live example](https://angular.io/generated/live-examples/inputs-outputs/stackblitz.html) / [download example](https://angular.io/generated/zips/inputs-outputs/inputs-outputs.zip). The following subsections highlight key points in the sample app.

#### @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() are independent

Though @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/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](https://angular.io/api/core/Input)(), only an @[Output](https://angular.io/api/core/Output)(). The reverse is also true in that if the child only needs to receive data from the parent, you'd only need @[Input](https://angular.io/api/core/Input)().

## How to use @[Input](https://angular.io/api/core/Input)()

Use the @[Input](https://angular.io/api/core/Input)() decorator or in a child component 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](https://angular.io/api/core/Input)() allows data to be input into the child component from the parent component.



To illustrate the use of @[Input](https://angular.io/api/core/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](https://angular.io/api/core/Input)() decorator in a child component class, first import [Input](https://angular.io/api/core/Input) and then decorate the property with @[Input](https://angular.io/api/core/Input)():

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

import { [Component](https://angular.io/api/core/Component), [Input](https://angular.io/api/core/Input) } from '@angular/core'; // [First](https://angular.io/), import [Input](https://angular.io/api/core/Input)

export class ItemDetailComponent {

@[Input](https://angular.io/api/core/Input)() [item](https://angular.io/api/core/IterableChangeRecord#item): string; // decorate the property with @[Input](https://angular.io/api/core/Input)()

}

In this case, @[Input](https://angular.io/api/core/Input)() decorates the property item, which has a type of string, however, @[Input](https://angular.io/api/core/Input)() properties can have any type, such as number, string, boolean, or object. The value for [item](https://angular.io/api/core/IterableChangeRecord#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](https://angular.io/api/core/IterableChangeRecord#item): {{[item](https://angular.io/api/core/IterableChangeRecord#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](https://angular.io/guide/template-syntax#property-binding) to bind the property in the child to the property of the parent.

src/app/app.component.html

<app-item-detail [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem"></app-item-detail>

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

src/app/app.component.ts

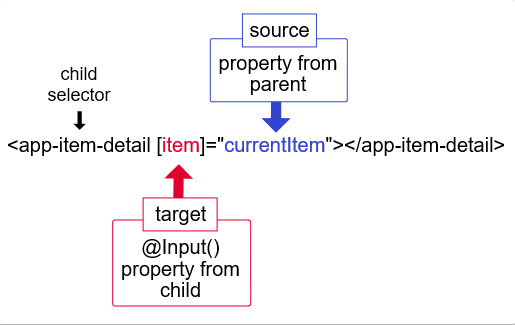
export class AppComponent {

currentItem = 'Television';

}

With @[Input](https://angular.io/api/core/Input)(), Angular passes the value for currentItem to the child so that [item](https://angular.io/api/core/IterableChangeRecord#item) renders as Television.

The following diagram shows this structure:



The target in the square brackets, [], is the property you decorate with @[Input](https://angular.io/api/core/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](https://angular.io/api/core/Input)() in the child component.

#### [OnChanges](https://angular.io/api/core/OnChanges) and @[Input](https://angular.io/api/core/Input)()

To watch for changes on an @[Input](https://angular.io/api/core/Input)() property, use [OnChanges](https://angular.io/api/core/OnChanges), one of Angular's [lifecycle hooks](https://angular.io/guide/lifecycle-hooks#onchanges). [OnChanges](https://angular.io/api/core/OnChanges) is specifically designed to work with properties that have the @[Input](https://angular.io/api/core/Input)() decorator. See the [OnChanges](https://angular.io/guide/lifecycle-hooks#onchanges) section of the [Lifecycle Hooks](https://angular.io/guide/lifecycle-hooks) guide for more details and examples.

## How to use @[Output](https://angular.io/api/core/Output)()

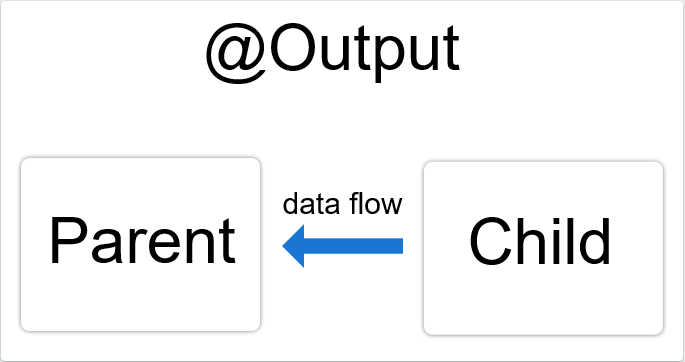
Use the @[Output](https://angular.io/api/core/Output)() decorator in the child component or directive to allow data to flow from the child out to the parent.

An @[Output](https://angular.io/api/core/Output)() property should normally be initialized to an Angular [EventEmitter](https://angular.io/api/core/EventEmitter) with values flowing out of the component as [events](https://angular.io/guide/template-syntax#event-binding).

Just like with @[Input](https://angular.io/api/core/Input)(), you can use @[Output](https://angular.io/api/core/Output)() on a property of the child component but its type should be [EventEmitter](https://angular.io/api/core/EventEmitter).

@[Output](https://angular.io/api/core/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](https://angular.io/api/core/Output)() works hand in hand with [EventEmitter](https://angular.io/api/core/EventEmitter), which is a class in @angular/core that you use to emit custom events.

When you use @[Output](https://angular.io/api/core/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](https://angular.io/api/core/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](https://angular.io/api/core/Input)() are different. This documentation is about component communication in Angular as it pertains to @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)(). For more information on the HTML element <input>, see the [W3C Recommendation](https://www.w3.org/TR/html5/sec-forms.html#the-input-element).

### 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](https://angular.io/api/core/EventEmitter) then relays the data to the parent component.

First, be sure to import [Output](https://angular.io/api/core/Output) and [EventEmitter](https://angular.io/api/core/EventEmitter) in the child component class:

import { [Output](https://angular.io/api/core/Output), [EventEmitter](https://angular.io/api/core/EventEmitter) } from '@angular/core';

Next, still in the child, decorate a property with @[Output](https://angular.io/api/core/Output)() in the component class. The following example @[Output](https://angular.io/api/core/Output)() is called newItemEvent and its type is [EventEmitter](https://angular.io/api/core/EventEmitter), which means it's an event.

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

@[Output](https://angular.io/api/core/Output)() newItemEvent = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>();

The different parts of the above declaration are as follows:

* @[Output](https://angular.io/api/core/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](https://angular.io/api/core/Output)()
* [EventEmitter](https://angular.io/api/core/EventEmitter)<string>—the @[Output](https://angular.io/api/core/Output)()'s type
* new [EventEmitter](https://angular.io/api/core/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](https://angular.io/api/core/EventEmitter), see the [EventEmitter API documentation](https://angular.io/api/core/EventEmitter).

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

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

export class ItemOutputComponent {

@[Output](https://angular.io/api/core/Output)() newItemEvent = new [EventEmitter](https://angular.io/api/core/EventEmitter)<string>();

addNewItem(value: string) {

this.newItemEvent.emit(value);

}

}

The addNewItem() function uses the @[Output](https://angular.io/api/core/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](https://angular.io/guide/template-syntax#ref-var) , #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](https://angular.io/api/core/IterableChangeRecord#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](https://angular.io/guide/template-syntax#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](https://angular.io/api/core/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](https://angular.io/api/core/Output)() working, add the following to the parent's template:

<ul>

<li \*[ngFor](https://angular.io/api/common/NgForOf)="let [item](https://angular.io/api/core/IterableChangeRecord#item) of items">{{[item](https://angular.io/api/core/IterableChangeRecord#item)}}</li>

</ul>

The \*[ngFor](https://angular.io/api/common/NgForOf) 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](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() together

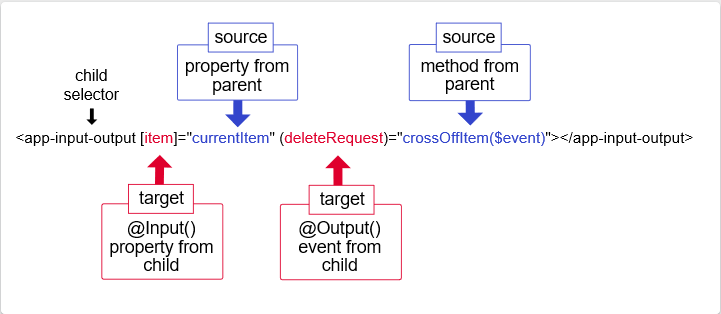
You can use @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() on the same child component as in the following:

src/app/app.component.html

<app-input-output [[item](https://angular.io/api/core/IterableChangeRecord#item)]="currentItem" (deleteRequest)="crossOffItem($event)"></app-input-output>

The target, [item](https://angular.io/api/core/IterableChangeRecord#item), which is an @[Input](https://angular.io/api/core/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](https://angular.io/api/core/Input)() and an @[Output](https://angular.io/api/core/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](https://angular.io/api/core/IterableChangeRecord#item) and deleteRequest being @[Input](https://angular.io/api/core/Input)() and @[Output](https://angular.io/api/core/Output)() properties in the child component class. The property currentItem and the method crossOffItem() are both in the parent component class.