



## Angular for TypeScript Cheat Sheet (v2.0.0-alpha.44)

Source: [angular.io/cheatsheet](http://angular.io/cheatsheet)

### Bootstrapping

```
import {bootstrap} from 'angular2/angular2';
```

```
bootstrap(MyAppComponent, [MyService,  
provide(...)]);
```

Bootstraps an application with `MyAppComponent` as the root component and configures the DI providers.

### Template syntax

```
<input [value]="firstName">
```

Binds property `value` to the result of expression `firstName`.

```
<div [attr.role]="myAriaRole">
```

Binds attribute `role` to the result of expression `myAriaRole`.

```
<div [class.extra-sparkle]="isDelightful">
```

Binds the presence of the css class `extra-sparkle` on the element to the truthiness of the expression `isDelightful`.

```
<div [style.width.px]="mySize">
```

Binds style property `width` to the result of expression `mySize` in pixels. Units are optional.

```
<button (click)="readRainbow($event)">
```

Calls method `readRainbow` when a click event is triggered on this button element (or its children) and passes in the event object.

```
<div title="Hello {{ponyName}}">
```

Binds a property to an interpolated string, e.g. "Hello Seabiscuit". Equivalent to: `<div [title]='Hello' + ponyName">`

```
<p>Hello {{ponyName}}</p>
```

Binds text content to an interpolated string, e.g. "Hello Seabiscuit".

```
<my-cmp [(title)]= "name">
```

Sets up two-way data binding. Equivalent to:  
`<my-cmp [title]="name" (title-change)="name=$event">`

```
<video #movieplayer ...>
```

```
<button (click)="movieplayer.play()" >
```

Creates a local variable `movieplayer` that provides access to the `video` element instance in data- and event-binding expressions in the current template.

```
<p *my-unless="myExpression">...</p>
```

The `*` symbol means that the current element will be turned into an embedded template. Equivalent to:

```
<template  
[my-unless]="myExpression"><p>...</p></template>
```

```
<p>Card No.: {{cardNumber |  
myCreditCardNumberFormatter}}</p>
```

Transforms the current value of expression `cardNumber` via pipe called `creditCardNumberFormatter`.

```
<p>Employer: {{employer?.companyName}}</p>
```

The Elvis operator (`?`) means that the `employer` field is optional and if undefined, the rest of the expression should be ignored.

Built-in directives	<code>import {NgIf, ...} from 'angular2/angular2';</code>
<code>&lt;section *ng-if="showSection"&gt;</code>	Removes or recreates a portion of the DOM tree based on the <code>showSection</code> expression.
<code>&lt;li *ng-for="#item of list"&gt;</code>	Turns the <code>li</code> element and its contents into a template, and uses that to instantiate a view for each <code>item</code> in <code>list</code> .
<code>&lt;div [ng-switch]="conditionExpression"&gt;   &lt;template [ng-switch-when]="case1Exp"&gt;...&lt;/template&gt;   &lt;template ng-switch-when="case2LiteralString"&gt;...&lt;/template&gt;   &lt;template ng-switch-default&gt;...&lt;/template&gt; &lt;/div&gt;</code>	Conditionally swaps the contents of the <code>div</code> by selecting one of the embedded templates based on the current value of <code>conditionExpression</code> .
<code>&lt;div [ng-class]="{active: isActive, disabled: isDisabled}"&gt;</code>	Binds the presence of <code>css</code> classes on the element to the truthiness of the associated map values. The right-hand side expression should return <code>{class-name: true/false}</code> map.
Forms	<code>import {FORM_DIRECTIVES} from 'angular2/angular2';</code>
<code>&lt;input [(ng-model)]="userName"&gt;</code>	Provides two-way data-binding, parsing and validation for form controls.
Class decorators	<code>import {Directive, ...} from 'angular2/angular2';</code>
<code>@Component({...}) class MyComponent() {}</code>	Declares that a class is a component and provides metadata about the component.
<code>@Pipe({...}) class MyPipe() {}</code>	Declares that a class is a pipe and provides metadata about the pipe.
<code>@Injectable() class MyService() {}</code>	Declares that a class has dependencies that should be injected into the constructor when the dependency injector is creating an instance of this class.
@Directive configuration (used as <code>@Directive({ property1: value1, ... })</code> )	
<code>selector: '.cool-button:not(a)'</code>	Specifies a <code>css</code> selector that identifies this directive within a template. Supported selectors include: <code>element</code> , <code>[attribute]</code> , <code>.class</code> , and <code>:not()</code> . Does not support parent-child relationship selectors.
<code>providers: [MyService, provide(...)]</code>	Array of dependency injection providers for this directive and its children.

**@Component configuration** (@Component extends @Directive, so the @Directive configuration above applies to components as well)

<b>viewProviders:</b> [MyService, provide(...)]	Array of dependency injection providers scoped to this component's view.
<b>template:</b> 'Hello {{name}}' <b>templateUrl:</b> 'my-component.html'	Inline template / external template url of the component's view.
<b>styles:</b> '.primary {color: red}' <b>styleUrls:</b> ['my-component.css']	List of inline css styles / external stylesheet urls for styling component's view.
<b>directives:</b> [MyDirective, MyComponent]	List of directives used in the the component's template.
<b>pipes:</b> [MyPipe, OtherPipe]	List of pipes used in the component's template.

**Class field decorators for directives and components**

`import {Input, ...} from 'angular2/angular2';`

<b>@Input()</b> myProperty;	Declares an input property that we can update via property binding, e.g. <code>&lt;my-cmp [my-property]="someExpression"&gt;</code>
<b>@Output()</b> myEvent = new EventEmitter();	Declares an output property that fires events to which we can subscribe with an event binding, e.g. <code>&lt;my-cmp (my-event)="doSomething()"&gt;</code>
<b>@HostBinding(' [class.valid]')</b> isValid;	Binds a host element property (e.g. css class <code>valid</code> ) to directive/component property (e.g. <code>isValid</code> )
<b>@HostListener('click', ['\$event'])</b> onClick(e) {...}	Subscribes to a host element event (e.g. <code>click</code> ) with a directive/component method (e.g., <code>onClick</code> ), optionally passing an argument ( <code>\$event</code> )
<b>@ContentChild(myPredicate)</b> myChildComponent;	Binds the first result of the component content query ( <code>myPredicate</code> ) to the <code>myChildComponent</code> property of the class.
<b>@ContentChildren(myPredicate)</b> myChildComponents;	Binds the results of the component content query ( <code>myPredicate</code> ) to the <code>myChildComponents</code> property of the class.
<b>@ViewChild(myPredicate)</b> myChildComponent;	Binds the first result of the component view query ( <code>myPredicate</code> ) to the <code>myChildComponent</code> property of the class. Not available for directives.
<b>@ViewChildren(myPredicate)</b> myChildComponents;	Binds the results of the component view query ( <code>myPredicate</code> ) to the <code>myChildComponents</code> property of the class. Not available for directives.

### Directive and component change detection and lifecycle hooks (implemented as class methods)

<code>constructor(myService: MyService, ...) { ... }</code>	The class constructor is called before any other lifecycle hook. Use it to inject dependencies, but avoid any serious work here.
<code>onChanges(changeRecord) { ... }</code>	Called after every change to input properties and before processing content or child views.
<code>onInit() { ... }</code>	Called after the constructor, initializing input properties, and the first call to <code>onChanges</code> .
<code>doCheck() { ... }</code>	Called every time that the input properties of a component or a directive are checked. Use it to extend change detection by performing a custom check.
<code>afterContentInit() { ... }</code>	Called after <code>onInit</code> when the component's or directive's content has been initialized.
<code>afterContentChecked() { ... }</code>	Called after every check of the component's or directive's content.
<code>afterViewInit() { ... }</code>	Called after <code>onContentInit</code> when the component's view has been initialized. Applies to components only.
<code>afterViewChecked() { ... }</code>	Called after every check of the component's view. Applies to components only.
<code>onDestroy() { ... }</code>	Called once, before the instance is destroyed.

### Dependency injection configuration

```
import {provide} from 'angular2/angular2';
```

<code>provide(MyService, {useClass: MyMockService})</code>	Sets or overrides the provider for <code>MyService</code> to the <code>MyMockService</code> class.
<code>provide(MyService, {useFactory: myFactory})</code>	Sets or overrides the provider for <code>MyService</code> to the <code>myFactory</code> factory function.
<code>provide(MyValue, {useValue: 41})</code>	Sets or overrides the provider for <code>MyValue</code> to the value 41.

**Routing and navigation**    `import {RouteConfig, ROUTER_DIRECTIVES, ROUTER_PROVIDERS, ...} from 'angular2/router';`

**@RouteConfig**(  
 { path: '/:myParam', component: MyComponent, as: 'MyCmp' },  
 { path: '/staticPath', component: ..., as: ...}  
 { path: '/\*wildCardParam', component: ..., as: ...}  
 )  
 class MyComponent() {}

Configures routes for the decorated component. Supports static, parameterized and wildcard routes.

**<router-outlet></router-outlet>**

Marks the location to load the component of the active route.

**<a [router-link]="[ '/MyCmp', {myParam: 'value' } ]">**

Creates a link to a different view based on a route instruction consisting of a route name and optional parameters. The route name matches the as property of a configured route. Add the '/' prefix to navigate to a root route; add the './' prefix for a child route.

**@CanActivate**(() => { ... })  
 class MyComponent() {}

A component decorator defining a function that the router should call first to determine if it should activate this component. Should return a boolean or a promise.

**onActivate**(nextInstruction, prevInstruction) { ... }

After navigating to a component, the router calls component's `onActivate` method (if defined).

**canReuse**(nextInstruction, prevInstruction) { ... }

The router calls a component's `canReuse` method (if defined) to determine whether to reuse the instance or destroy it and create a new instance. Should return a boolean or a promise.

**onReuse**(nextInstruction, prevInstruction) { ... }

The router calls the component's `onReuse` method (if defined) when it re-uses a component instance.

**canDeactivate**(nextInstruction, prevInstruction) { ... }

The router calls the `canDeactivate` methods (if defined) of every component that would be removed after a navigation. The navigation proceeds if and only if all such methods return `true` or a promise that is resolved.

**onDeactivate**(nextInstruction, prevInstruction) { ... }

Called before the directive is removed as the result of a route change. May return a promise that pauses removing the directive until the promise resolves.