This cheat sheet is provisional and may change. Angular 2 is currently in Beta.

Angular for TypeScript Cheat Sheet (v2.0.0-beta.17)

Bootstrapping	`import {bootstrap} from 'angular2/platform/browser';`
`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" (titleChange)="name=$event">`

`<video #movieplayer ...>`
`  <button (click)="movieplayer.play()">`
`</video>`
	Creates a local variable `movieplayer` that provides access to the `video` element instance in data-binding and event-binding expressions in the current template.

`<p *myUnless="myExpression">...</p>`	The `*` symbol means that the current element will be turned into an embedded template. Equivalent to: `<template [myUnless]="myExpression"><p>...</p></template>`
`<p>Card No.: {{cardNumber | myCreditCardNumberFormatter}}</p>`	Transforms the current value of expression `cardNumber` via the pipe called `myCreditCardNumberFormatter`.
`<p>Employer: {{employer?.companyName}}</p>`	The safe navigation operator (`?`) means that the `employer` field is optional and if `undefined`, the rest of the expression should be ignored.

Built-in directives	`import {NgIf, ...} from 'angular2/common';`
`<section *ngIf="showSection">`	Removes or recreates a portion of the DOM tree based on the showSection expression.
`<li *ngFor="let item of list">`	Turns the li element and its contents into a template, and uses that to instantiate a view for each item in list.

`<div [ngSwitch]="conditionExpression">`
`  <template [ngSwitchWhen]="case1Exp">...`
`</template>`
`  <template ngSwitchWhen="case2LiteralString">...`
`</template>`
`  <template ngSwitchDefault>...</template>`
`</div>`
	Conditionally swaps the contents of the div by selecting one of the embedded templates based on the current value of conditionExpression.

`<div [ngClass]="` `{active: isActive, disabled: isDisabled}">`	Binds the presence of CSS classes on the element to the truthiness of the associated map values. The right-hand side expression should return {class-name: true/false} map.

Forms	`import {FORM_DIRECTIVES} from 'angular2/common';`
`<input [(ngModel)]="userName">`	Provides two-way data-binding, parsing and validation for form controls.

Class decorators	`import {Directive, ...} from 'angular2/core';`
`@Component({...})` `class MyComponent() {}`	Declares that a class is a component and provides metadata about the component.
`@Directive({...})` `class MyDirective() {}`	Declares that a class is a directive and provides metadata about the directive.
`@Pipe({...})` `class MyPipe() {}`	Declares that a class is a pipe and provides metadata about the pipe.
`@Injectable()` `class MyService() {}`	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	`@Directive({ property1: value1, ... })`
`selector: '.cool-button:not(a)'`	Specifies a CSS selector that identifies this directive within a template. Supported selectors include `element`, `[attribute]`, `.class`, and `:not()`.

	Does not support parent-child relationship selectors.
`providers: [MyService, provide(...)]`	Array of dependency injection providers for this directive and its children.

Component configuration	`@Component` extends `@Directive`, so the `@Directive` configuration applies to components as well
`viewProviders: [MyService, provide(...)]`	Array of dependency injection providers scoped to this component's view.
`template: 'Hello {{name}}'` `templateUrl: 'my-component.html'`	Inline template / external template URL of the component's view.
`styles: ['.primary {color: red}']` `styleUrls: ['my-component.css']`	List of inline CSS styles / external stylesheet URLs for styling component's view.
`directives: [MyDirective, MyComponent]`	List of directives used in the the component's template.
`pipes: [MyPipe, OtherPipe]`	List of pipes used in the component's template.

Class field decorators for directives and components	`import {Input, ...} from 'angular2/core';`
`@Input() myProperty;`	Declares an input property that we can update via property binding (e.g. `<my-cmp [my-property]="someExpression">`).
`@Output() myEvent = new EventEmitter();`	Declares an output property that fires events to which we can subscribe with an event binding (e.g. `<my-cmp (my-event)="doSomething()">`).
`@HostBinding('[class.valid]') isValid;`	Binds a host element property (e.g. CSS class valid) to directive/component property (e.g. isValid).
`@HostListener('click', ['$event']) onClick(e) {...}`	Subscribes to a host element event (e.g.

	click) with a directive/component method (e.g. onClick), optionally passing an argument ($event).
`@ContentChild(myPredicate) myChildComponent;`	Binds the first result of the component content query (myPredicate) to the myChildComponent property of the class.
`@ContentChildren(myPredicate) myChildComponents;`	Binds the results of the component content query (myPredicate) to the myChildComponents property of the class.
`@ViewChild(myPredicate) myChildComponent;`	Binds the first result of the component view query (myPredicate) to the myChildComponent property of the class. Not available for directives.
`@ViewChildren(myPredicate) myChildComponents;`	Binds the results of the component view query (myPredicate) to the myChildComponents property of the class. Not available for directives.

Directive and component change detection and lifecycle hooks	(implemented as class methods)
`constructor(myService: MyService, ...) { ... }`	The class constructor is called before any other lifecycle hook. Use it to inject dependencies, but avoid any serious work here.
`ngOnChanges(changeRecord) { ... }`	Called after every change to input properties and before processing content or child views.
`ngOnInit() { ... }`	Called after the constructor, initializing input properties, and the first call to ngOnChanges.
`ngDoCheck() { ... }`	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.
`ngAfterContentInit() { ... }`	Called after ngOnInit when the component's or directive's content has been initialized.

`ngAfterContentChecked() { ... }`	Called after every check of the component's or directive's content.
`ngAfterViewInit() { ... }`	Called after ngAfterContentInit when the component's view has been initialized. Applies to components only.
`ngAfterViewChecked() { ... }`	Called after every check of the component's view. Applies to components only.
`ngOnDestroy() { ... }`	Called once, before the instance is destroyed.

Dependency injection configuration	`import {provide} from 'angular2/core';`
`provide(MyService, {useClass: MyMockService})`	Sets or overrides the provider for MyService to the MyMockService class.
`provide(MyService, {useFactory: myFactory})`	Sets or overrides the provider for MyService to the myFactory factory function.
`provide(MyValue, {useValue: 41})`	Sets or overrides the provider for MyValue to the value 41.

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

`@RouteConfig([`
`  { path: '/:myParam', component: MyComponent, name: 'MyCmp' },`
`  { path: '/staticPath', component: ..., name: ...},`
`  { path: '/*wildCardParam', component: ..., name: ...}`
`])`
`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 [routerLink]="[ '/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.
`routerOnActivate(nextInstruction, prevInstruction) { ... }`	After navigating to a component, the router calls the component's routerOnActivate method (if defined).
`routerCanReuse(nextInstruction, prevInstruction) { ... }`	The router calls a component's routerCanReuse 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.
`routerOnReuse(nextInstruction, prevInstruction) { ... }`	The router calls the component's routerOnReuse method (if defined) when it re-uses a component instance.
`routerCanDeactivate(nextInstruction, prevInstruction) { ... }`	The router calls the routerCanDeactivate 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.
`routerOnDeactivate(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.