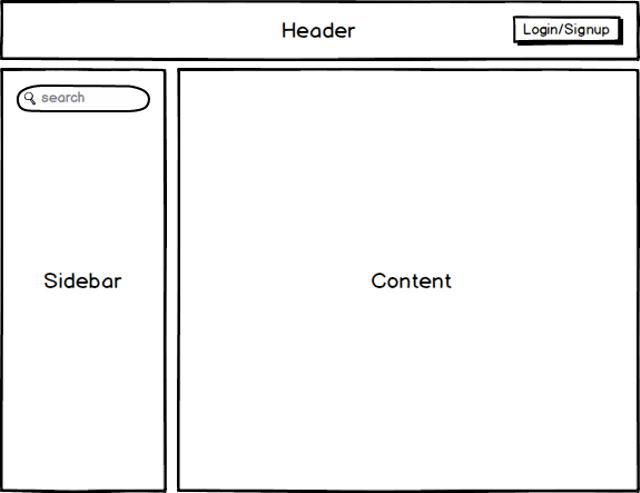
## Angular Components:

At the root of that tree is the top-level Component, the *root* Component.

When we *bootstrap* an Angular application we are telling the browser to render that top-level *root* Component which renders its child Components and so on.



**HeaderComponent**

**Responsibilities**

All aspects of authentication. Letting the user login/signup and logout.

**Inputs**

None

**Outputs**

* LoginChanged — An output event that is fired when the users login state changes.

**SidebarComponent**

**Responsibilities**

Performing searches

**Inputs**

None

**Outputs**

* SearchTermChanged — An output event that is fired when a user performs a search, $event contains the search term.

**ContentComponent**

**Responsibilities**

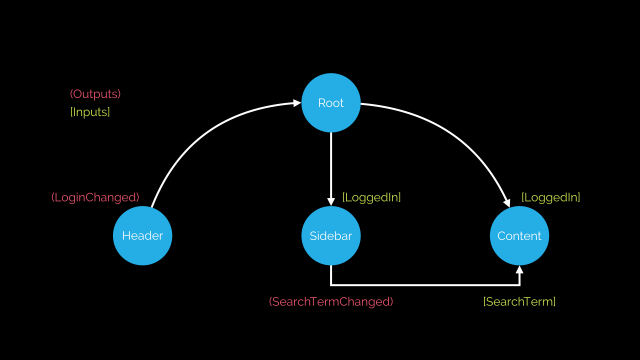
Showing the search results.

**Inputs**

* SearchTerm the search term that we want to filter the results by.

**Outputs**

None



The actual binding of inputs and outputs happens in HTML, in the templates of Components. The template for our root Component might end up looking like so:

HTML

<header (loginChanged)="loggedIn = $event"></header>

<sidebar (searchTermChanged)="searchTerm = $event"></sidebar>

<content [searchTerm]="searchTerm"></content>

Data flow describes how we glue Components together through their inputs and outputs.

Closely looking at the diagram above an interesting fact occurs; with one-way data binding, inputs go down the tree, outputs go up the tree. With one-way data binding reasoning about your application becomes a lot simpler, we can trace through the flow of events in our application easily.

It is also a type of directive with template, styles and logic part which is most famous type of directive among all in Angular2.

In this type of directive, you can use other directives whether it is custom or builtin in the @Component annotation like following:

@Component({

selector: "my-app"

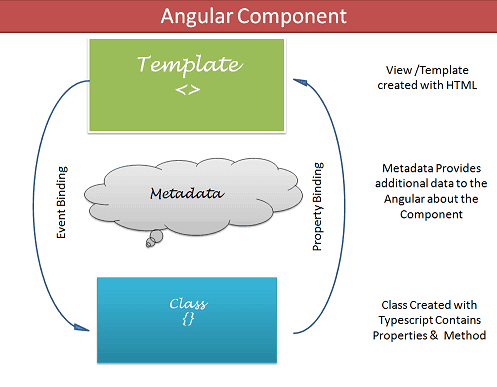
directives: [custom\_directive\_here]

})

Use this directive in your view as:

<my-app></my-app>

The Angular Components are plain JavaScript classes and defined using @component Decorator.



This Decorator provides the component with the View to display (known as Template) & Metadata about the class.

The component contains the data & user interaction logic that defines how the View looks and behaves. A view in Angular refers to a template (HTML).

The Angular Components are plain JavaScript classes and defined using @component Decorator. This Decorator provides the component with the View to display & Metadata about the class.

The Component is responsible to provide the data to the view. The Angular does this by using data binding to get the data from the Component to the View.

This is done using the special HTML markup knows as the Angular Template Syntax. The Component can also get notified when the View Changes.

The Angular applications will have lots of components. Each component handles a small part of UI. These components work together to produce the complete user interface of the application

The Components consists of three main building blocks

1) Template

2) Class

3) MetaData

# Building blocks of the Angular Component

## Template (View)

The template defines the layout of the View and defines what is rendered on the page.  Without the template, there is nothing for Angular to render to the DOM.

## What is Angular Directive:

Decorator that marks a class as an Angular directive. You can define your own directives to attach custom behavior to elements in the DOM.

***Angular* *Directive*** is basically a class with a *@Directive* decorator.

***Decorators***are functions that modify JavaScript classes. Decorators are used for attaching metadata to classes, it knows the configuration of those classes and how they should work.

A decorator is used to mark the class as the component in Angular, and it provides informational metadata that defines what kind of properties can be used by the existing component.

A component takes properties as metadata as object, and the object contains key-value pairs like selector, style, or styleUrl. All these properties make a component a complete reusable chunk for the Angular application.

Component is also a directive-with-a-template. A **@Component** decorator is actually a **@Directive** decorator extended with template-oriented features. Whenever Angular renders a directive, it changes the **DOM** according to the instructions given by the directive. Directive appears within an element tag similar to attributes.

The Angular Directive can be classified into two types: **structural** and **attribute** directives.

**Subclasses: is** [Component](https://angular.io/api/core/Component).

This section walks you through creating a highlight directive that sets the background color of the host element to yellow.

Example:

**To create a directive, use the CLI command**[**ng generate directive**](https://angular.io/cli/generate)**.**

C:> ng generate directive highlight

The CLI creates src/app/highlight.directive.ts, a corresponding test file src/app/highlight.directive.spec.ts, and declares the directive class in the AppModule.

The CLI generates the default src/app/highlight.directive.ts

Components have their own view (HTML and styles). Directives are just "behavior" added to existing elements and components.

**Component extends Directive.**

Because of that there can only be one component on a host element, but multiple directives.

Structural directives are directives applied to <template> elements and used to add/remove content (stamp the template). The \* in directive applications like \*ngIf causes a <template> tag to be created implicitly.

So in simple term it is explained below:

* If it has a template, it is a **Component**
* else if it has a selector in brackets "[likethis]", it is an **Attribute Directive**
* else it is a **Structural Directive**.
* **Component**: A view with associated behavior. This type of directive actually adds DOM elements
* **Attribute directives**: Can be attached to DOM elements (and components since they are DOM elements) to modify the appearance or behavior of an element.
* **Structural directives**: Can be attached to DOM elements (and components since they are DOM elements) to modify the DOM layout. Structural directives start with a \* and actually add or remove DOM element. For example \*ngIf which can insert or remove an DOM element (or angular component which is a custom DOM element, but still a DOM element).

## [**@Directive vs @Component in Angular**](https://stackoverflow.com/questions/32680244/directive-vs-component-in-angular)

1. A @Component requires a view whereas a @Directive does not.
2. Directives add behavior to an existing DOM element or an existing component instance. One example use case for a directive would be to log a click on an element.

import {Directive} from '@angular/core';

@Directive({

selector: "[logOnClick]",

hostListeners: {

'click': 'onClick()',

},

})

class LogOnClick {

constructor() {}

onClick() { console.log('Element clicked!'); }

}

Which would be used like so:

<button logOnClick>I log when clicked!</button>

## Components

A component, rather than adding/modifying behavior, actually creates its own view (hierarchy of DOM elements) with attached behavior. An example use case for this might be a contact card component:

1. To register a component, we use @Component meta-data annotation.
2. Component is a directive which uses shadow DOM to create encapsulated visual behavior called components. Components are typically used to create UI widgets.
3. Component is used to break up the application into smaller components.
4. Only one component can be present per DOM element.
5. @View decorator or template url template are mandatory in the component.

import {Component, View} from '@angular/core';

@Component({

selector: 'contact-card',

template: `

<div>

<h1>{{name}}</h1>

<p>{{city}}</p>

</div>

`

})

class ContactCard {

@Input() name: string

@Input() city: string

constructor() {}

**A @Component requires a view whereas a @Directive does not.**

## Directives

(Directives aren't limited to attribute usage.) Directives add behavior to an existing DOM element or an existing component instance. One example use case for a directive would be to log a click on an element.

1. To register directives, we use @Directive meta-data annotation.
2. Directive is used to add behavior to an existing DOM element.
3. Directive is use to design re-usable components.
4. Many directives can be used per DOM element.
5. Directive doesn't use View.

import {Directive} from '@angular/core';

@Directive({

selector: "[logOnClick]",

hostListeners: {

'click': 'onClick()',

},

})

class LogOnClick {

constructor() {}

onClick() { console.log('Element clicked!'); }

}

Which would be used like so:

<button logOnClick>I log when clicked!</button>

## Components

A component, rather than adding/modifying behavior, actually creates its own view (hierarchy of DOM elements) with attached behavior. An example use case for this might be a contact card component:

import {Component, View} from '@angular/core';

@Component({

selector: 'contact-card',

template: `

<div>

<h1>{{name}}</h1>

<p>{{city}}</p>

</div>

`

})

class ContactCard {

@Input() name: string

@Input() city: string

constructor() {}

}

Which would be used like so:

<contact-card [name]="'foo'" [city]="'bar'"></contact-card>

ContactCard is a reusable UI component that we could use anywhere in our application, even within other components. These basically make up the UI building blocks of our applications.

## Example:

import { Component, HostListener, HostBinding, Directive, ElementRef } from '@angular/core';

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(el: ElementRef) {

el.nativeElement.style.backgroundColor = 'yellow';

}

}

@Component({

selector: 'app-root',

template: `

<div \*ngIf='myBool' appHighlight>Hi there</div>

`,

styleUrls: ['./app.component.scss'],

})

export class AppComponent {

myBool:boolean = true;

}

An angular directive is one of the core building blocks in Angular. A directive is **instructions or guidelines for rendering a template**. It allows us to attach attributes on the element and with this attribute we can manipulate the appearance or behavior of a DOM element. We can change the appearance, behavior, or layout of a DOM element using the Directives. They help you to extend HTML.

A directive is very much like a component, except it does not have a template. In fact, the Component class inherits from the Directive class in the framework.

There are three kinds of directives in Angular:

1. Components—directives with a template.
2. Structural directives—change the DOM layout by adding and removing DOM elements.
3. Attribute directives—change the appearance or behavior of an element, component, or another directive.

* **Component Directive**

Components are special directives in Angular. They are the directive with a template (view).

* **Structural Directives**

Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol. It changes the structure of the view. Two examples are [NgFor](https://v2.angular.io/docs/ts/latest/guide/template-syntax.html#ngFor) and [NgIf](https://v2.angular.io/docs/ts/latest/guide/template-syntax.html#ngIf).

Let us briefly understand the two majorly used built-in structural directives:

<li \*ngFor="let movie of movies"></li>

<movie-detail \*ngIf="selectedMovie"></movie-detail>

\*ngFor is a looping variable that tells Angular to take one <li> per movie from the movies list.

* \*ngIf will include the MovieDetail component only if a movie is selected otherwise it will remove it from the DOM.
* \*ngIf will include the MovieDetail component only if a movie is selected otherwise it will remove it from the DOM.
* **Attribute directives**

Attribute directives attach behavior to elements. An Attribute directive changes the appearance or behavior of a DOM element. It used as attributes of elements. The built-in [NgStyle](https://v2.angular.io/docs/ts/latest/guide/template-syntax.html#ngStyle) directive in the Template Syntax guide, for example, can change several element styles at the same time.

They are used to give custom behavior or style to the existing elements by applying some functions/logic. Like ngStyle is an attribute directive to give style dynamically to the elements. We can create our own directive and use this as attribute of some predefined or custom elements, here is the example of a simple directive:

Firstly we have to import directive from @angular/core

import {Directive, ElementRef, Renderer, Input} from '@angular/core';

@Directive({

selector: '[Icheck]',

})

export class RadioCheckbox {

// custom logic here...

}

We can use this in the view as shown below:

<span Icheck>HEllo Directive</span>

Using the Angular CLI, run the following command, where unless is the name of the directive:

ng generate directive unless

Angular creates the directive class and specifies the CSS selector, appUnless, that identifies the directive in a template.

**Attribute Directive** is basically used to modify or alter the appearance and behavior of an element.

The selector is the property that identifies the attribute. It is used as HTML tag to target & insert an instance of the directive class where it finds that tag. The directive class implements the desired directive behavior.

Now we will create a *myHighlight* attribute directive to set an element’s background color when you hovers over that element.

It can be applied anywhere using the below code:

Highlight me!

Create the highlight directive typescript file i.e. src/app/highlight.directive.ts and embed the following code:

import { Directive, ElementRef, HostListener, Input } from '@angular/core';

@Directive({

selector: '[myHighlight]'

})

export class HighlightDirective {

constructor(private el: ElementRef) { }

@Input() defaultColor: string;

@Input('myHighlight') highlightColor: string;

@HostListener('mouseenter') onMouseEnter() {

this.highlight(this.highlightColor || this.defaultColor || 'red');

}

@HostListener('mouseleave') onMouseLeave() {

this.highlight(null);

}

private highlight(color: string) {

this.el.nativeElement.style.backgroundColor = color;

}

}

The import statement specifies the required dependencies used from the Angular core:

The directive provides the functionality of the @Directive decorator.

ElementRef injects into the directive’s constructor so that the code can access the DOM element.

Input allows data to flow from the binding expression into the directive.

Next, the @Directive decorator function contains the directive metadata in a configuration object as an argument.

@Directive requires a selector to identify that the HTML in the template is associated with the directive. Here, the directive’s selector is [myHighlight]. Angular locates all elements in the template which has an attribute named myHighlight.

**Here's the list of decorators available in Angular:**

@NgModule.

@Component.

@Injectable.

@Directive.

@Pipe.

@Input.

@Output.

@HostBinding.

## Angular Decorators

Before we look at creating a custom decorator and why/how Angular uses them, let’s look at the different types of decorators that Angular offers. There are four main types:

1. Class decorators, e.g. @Component and @NgModule
2. Property decorators for properties inside classes, e.g. @Input and @Output
3. Method decorators for methods inside classes, e.g. @HostListener
4. Parameter decorators for parameters inside class constructors, e.g. @Inject

Each decorator has a unique role, let’s jump to some examples to expand on the list above.

## Class Decorators

Angular offers us a few class decorators. These are the top-level decorators that we use to express intent for classes. They allow us to tell Angular that a particular class is a component, or module, for example. And the decorator allows us to define this intent without having to actually put any code inside the class.

A @Component and @NgModule decorator example with classes:

import { NgModule, Component } from '@angular/core';

@Component({

selector: 'example-component',

template: '<div>Woo a component!</div>',

})

export class ExampleComponent {

constructor() {

console.log('Hey I am a component!');

}

}

@NgModule({

imports: [],

declarations: [],

})

export class ExampleModule {

constructor() {

console.log('Hey I am a module!');

}

}

Notice how both classes by themselves are effectively the same. No code is needed within the class to tell Angular that it is a component or a module. All we need to do is decorate it, and Angular will do the rest.

## Property Decorators

These are probably the second most common decorators that you’ll come across. They allow us to decorate specific properties within our classes - an extremely powerful mechanism.

Let’s take a look at @Input(). Imagine that we have a property within our class that we want to be an input binding.

Without decorators, we’d have to define this property in our class anyway for TypeScript to know about it, and then somewhere else tell Angular that we’ve got a property that we want to be an input.

With decorators, we can simply put the @Input() decorator above the property - which Angular’s compiler will automatically create an input binding from the property name and link them.

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

@Component({

selector: 'example-component',

template: '<div>Woo a component!</div>'

})

export class ExampleComponent {

@Input()

exampleProperty: string;

}

We’d then pass the input binding via a component property binding:

<example-component

[exampleProperty]="exampleData">

</example-component>

The property decorator and “magic” happens within the ExampleComponent definition.

## Method Decorators

Method decorators are very similar to property decorators but are used for methods instead. This let’s us decorate specific methods within our class with functionality. A good example of this is @HostListener. This allows us to tell Angular that when an event on our host happens, we want the decorated method to be called with the event.

import { Component, HostListener } from '@angular/core';

@Component({

selector: 'example-component',

template: 'Woo a component!'

})

export class ExampleComponent {

@HostListener('click', ['$event'])

onHostClick(event: Event) {

// clicked, `event` available

}

}

## Parameter Decorators

Parameter decorators are quite interesting. You may have come across these when injecting primitives into a constructor, where you need to manually tell Angular to inject a particular provider.

Parameter decorators allow us to decorate parameters in our class constructors. An example of this is @Inject that lets us tell Angular what we want that parameter to be initiated with:

import { Component, Inject } from '@angular/core';

import { MyService } from './my-service';

@Component({

selector: 'example-component',

template: 'Woo a component!'

})

export class ExampleComponent {

constructor(@Inject(MyService) myService) {

console.log(myService); // MyService

}

}

Due to the metadata that TypeScript exposes for us we don’t actually have to do this for our providers. We can just allow TypeScript and Angular to do the hard work for us by specifying the provider to be injected as the parameter type:

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

import { MyService } from './my-service';

@Component({

selector: 'example-component',

template: 'Woo a component!'

})

export class ExampleComponent {

constructor(myService: MyService) {

console.log(myService); // MyService

}

}

Now that we’ve covered the types of decorators we can use, let’s dig into what they actually are doing - and why we need them.

Now that we’ve covered the types of decorators we can use, let’s dig into what they actually are doing - and why we need them.

## Creating a decorator

It makes things a lot easier if we understand what a decorator is actually doing before we look into how Angular uses them under the hood. To do this, we can create a quick example decorator.

## Decorator functions

Decorators are actually just functions, it’s as simple as that, and are called with whatever they are decorating. A method decorator will be called with the value of the method it’s decorating, and a class decorator will be called with the class to be decorated.

Let’s quickly make a decorator that we can use on a class to demonstrate this a little further. This decorator is just going to simply log the class to the console:

function Console(target) {

console.log('Our decorated class', target);

}

Here, we have created Console (using the uppercase naming convention Angular uses) and are specifying a single argument called target. The target will in fact be the class that we decorate, which means we can now decorate any class with our decorator and see it outputted in the console:

@Console

class ExampleClass {

constructor() {

console.log('Yo!');

}

}

## Storing metadata

The whole point of a decorator is to store metadata about a class, method or property as we’ve already explored. When you configure a component for example, you’re providing metadata for that class that tells Angular that we have a component, and that component has a specific configuration.

In Angular we can decorate a class with extra info using the @ syntax, like so:

**Typescript**

@Component({

selector: "thingy",

template: `foo`

})

class MyComponent {

}

It allows us to *decorate* classes and functions, similar to annotations in Java and decorators in Python.

Specific Angular implementations might be more complex and harder to read and understand but the concept is actually quite simple.

Simple No-Argument Decorator.

I’m going to explain by creating a decorator called @course for our Person class

TypeScript

@course

class Person {

firstName;

lastName;

constructor (firstName, lastName) {

this.firstName = firstName;

this.lastName = lastName;

}

}

@course is just a function, like so:

**TypeScript**

function course(target) {

Object.defineProperty(target.prototype, 'course', {value: () => "Angular 2"})

}

## What is Databinding?

Databinding in Angular is way of communicating between the DOM and the business logic. In order words, how does the data in the HTML page bind to the TypeScript code.

This is important because when you write TypeScript code, it is compiled into JavaScript and the final output is on a HTML template.

Therefore, when something changes in the code, users should be able to see the output immediately. This is why we need databinding.

There are two type of databinding we’ll discuss here.

1. One-way databinding.

This type of databinding is like a one-way communication. If the value in the TypeScript code changes, then the change is updated in the HTML element.

But not vice versa. Put in another way, we can update the view from the model but you can’t update the model from the view.

Example of one-way databinding includes:

1. Interpolation
2. Event Binding
3. Property Binding



* Interpolation {{…}}

This is simply a way to display dynamic data in the HTML page.

For example, if we have a string firstname in our ngOnInit() method as shown below

ngOnInit(): void {

this.firsname = 'Kindson The Genius';

}

then we could use it in the HTML

{{firsname}}

This would render the text on the page.

## What does Div mean in Html?

The Div is a tag in Html. Html <Div> tag is used for making the divisions or sections of the content in the web page. <Div> tag is a paired tag, so it is mandatory to close this tag. If we are using this tag, then there is no effect on the layout or content of the web page.

1. Div tag is Block level tag
2. It is a generic container tag
3. It is used to the group of various tags of HTML so that sections can be created and style can be applied to them.

## What is Dom in angular?

The **DOM** (Document Object Model) is an interface that represents how your HTML and XML documents are read by the browser. It allows a language (JavaScript) to manipulate, structure, and style your website.

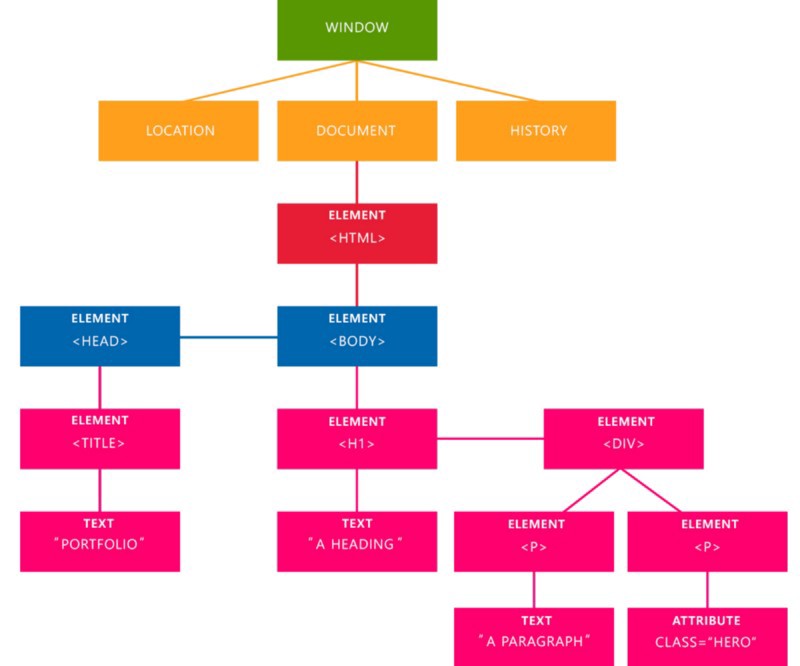
It defines events, methods, properties for all HTML elements as objects. DOM in AngularJS acts as an API (programming interface) for JavaScript. Whenever a web page is loaded, the browser creates a Document Model Object (DOM) of that page.

The DOM (Document Object Model) is an interface that represents how your HTML and XML documents are read by the browser. It allows a language (JavaScript) to manipulate, structure, and style your website. After the browser reads your HTML document, it creates a representational tree called the Document Object Model and defines how that tree can be accessed.

## Advantages

By manipulating the DOM, you have infinite possibilities. You can create applications that update the data of the page without needing a refresh. Also, you can create applications that are customizable by the user and then change the layout of the page without a refresh. You can drag, move, and delete elements.

## Representation by the browser



The representational tree that the browser creates after it read your document.

In the image above, we can see the representational tree and how it is created by the browser. In this example, we have four important elements that you’re going to see a lot:

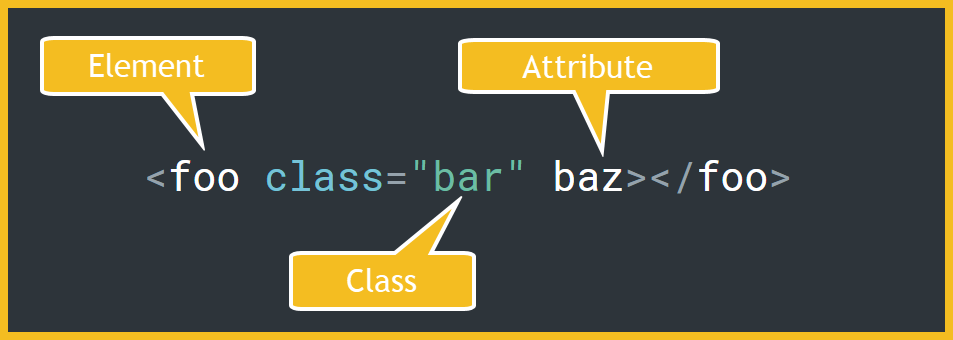
1. **Document:** It treats all the HTML documents**.**
2. **Elements:** All the tags that are inside your HTML or XML turn into a DOM element**.**
3. **Text:** All the tags’ content**.**
4. **Attributes:** All the attributes from a specific HTML element. In the image, the attribute class=” hero” is an attribute from the <p> element**.**

## Angular Elements

Angular Elements is a new package in Angular that helps us publish Angular components as custom elements. It does this by taking the Angular component and compiling it into a web component

A HTML element is can be broken down into individual segments:

* Element — is a html element tag name
* Class — is CSS class name associated with the element
* Attribute — provides additional information about HTML elements



## what is the use of Selector in Directive?

usage of the selector property of the @Directive decorator.

Selectors can be declared using any of the following as per the angular documentation:

1. element name or tag name
2. [attribute] or [attribute=value]
3. .class
4. :not(sub\_selector)
5. selector1, selector2

The above are essentially DOM query selectors used to target one or more HTML elements in the DOM.

the selector is a CSS selector. The same kind of selector you use in a CSS file to specify which elements are concerned by a CSS rule:

1) foo means elements with name foo

2) [foo] means elements with an attribute named foo

3) .foo means elements with a CSS class named foo

4) foo[bar] means elements named foo with an attribute named bar, etc.

## Basic Selectors

Selectors fall into one of the three categories:

## 1. Element Name Selector

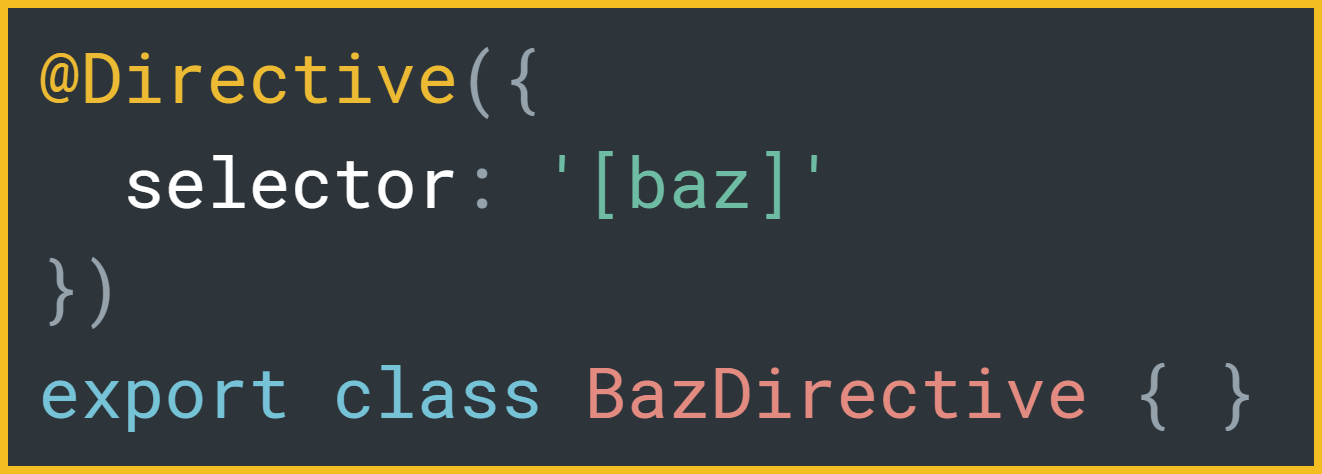
Using the element name selector, you can target specific HTML element tags, this can be either a standard Html element or a Angular component. The example below is targeting all element tags with the name **foo**.



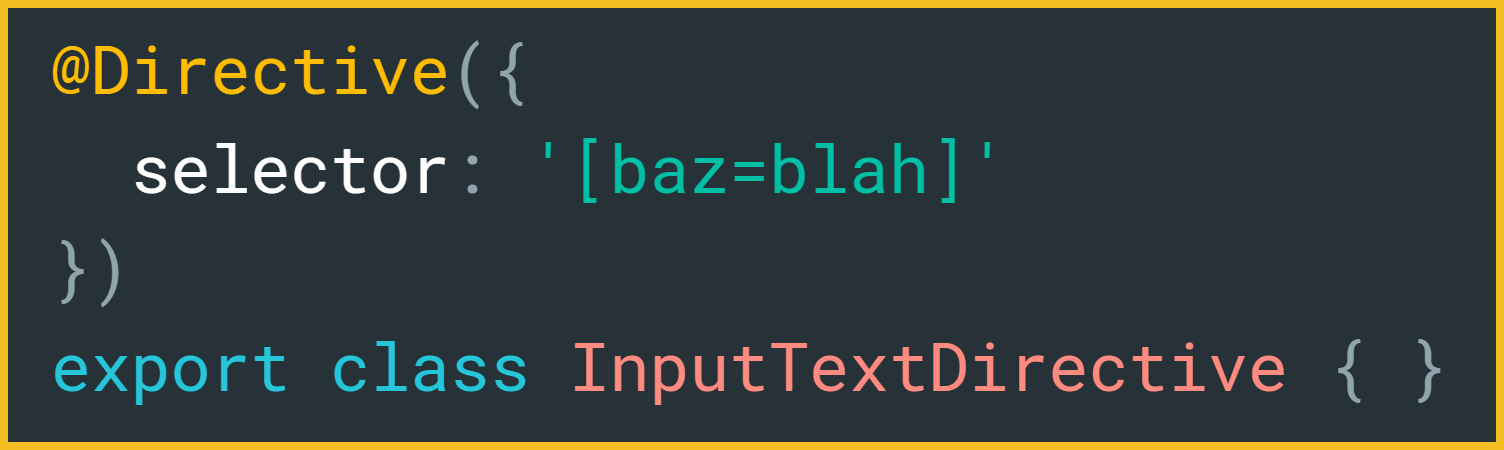
## 2. Attribute Selector

Using the attribute selector you can target specific HTML elements that contain the specific attribute(s). Attributes selectors are contained within square brackets **[ ]**.

The example below is targeting all elements that have an attribute with the name **baz**.



Attribute selectors can also be further restricted to the value the attribute is being set to. The example below is targeting all attributes that have an attribute with the name **baz** with a value of **blah**.



## Class Selector

Using the class selector you can target specific HTML elements with the specified CSS class names. The CSS selector is prefixed with a dot, which is the same as CSS syntax.

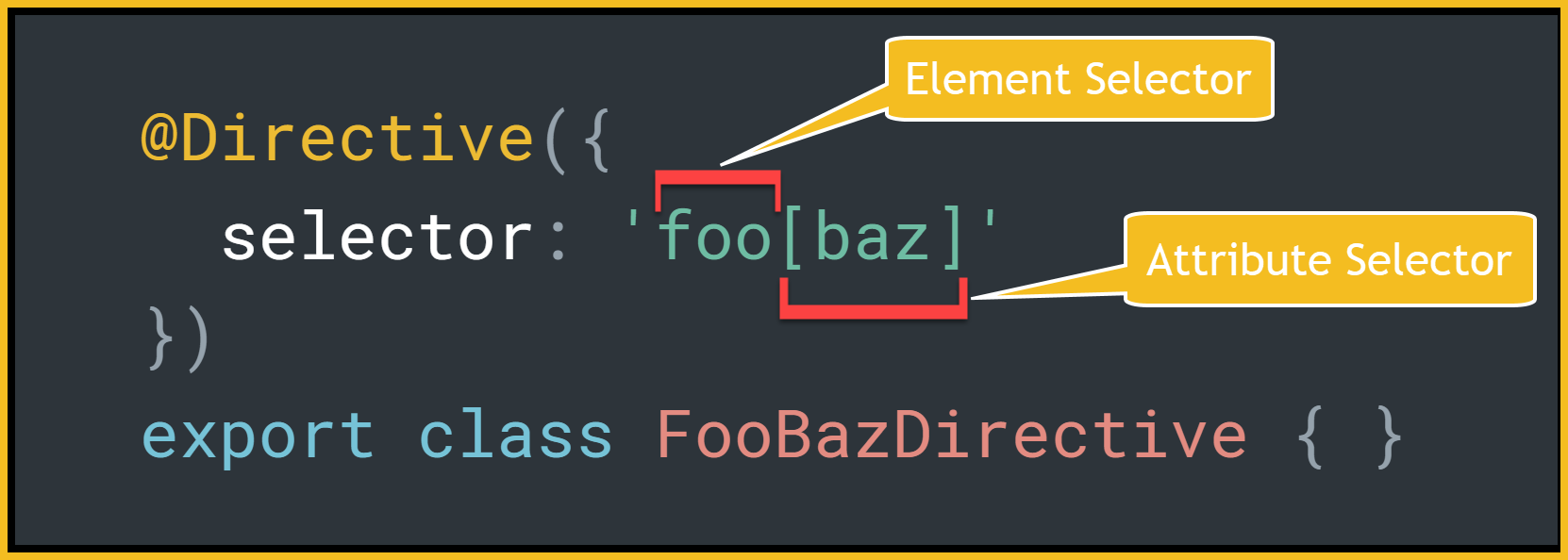
The example below is targeting all elements that have a CSS class names containing **bar**



One caveat with the class selector is that you cannot target class names that are dynamically inserted into the DOM using an expression binding either via a [class binding](https://angular.io/guide/template-syntax#class-binding) or [NgClass attribute directive](https://angular.io/guide/template-syntax#ngclass).

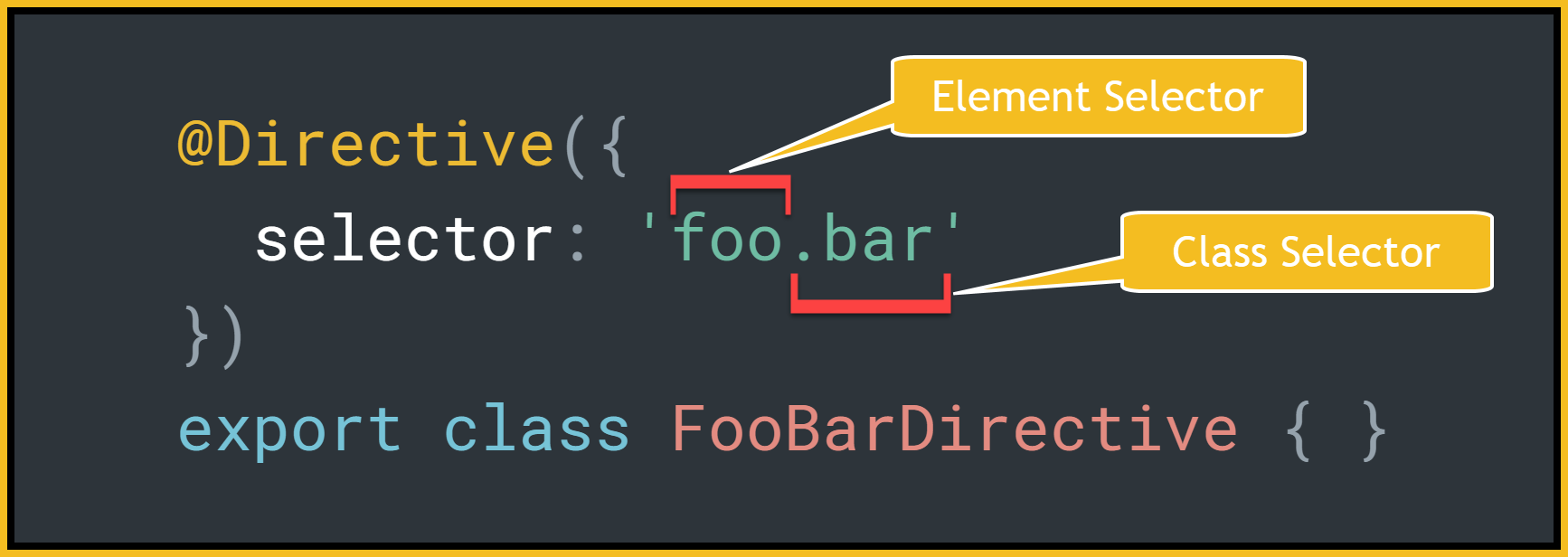
## **Chaining Selectors ( and )**

Each of the selectors (element, class and attribute) can be chained together to narrow the query for the relevant HTML elements like “selector and selector”. There are no tokens that separate each of the chained selectors, the selectors are separated using selector token itself either using square brackets for attribute selectors or a dot for class selectors.



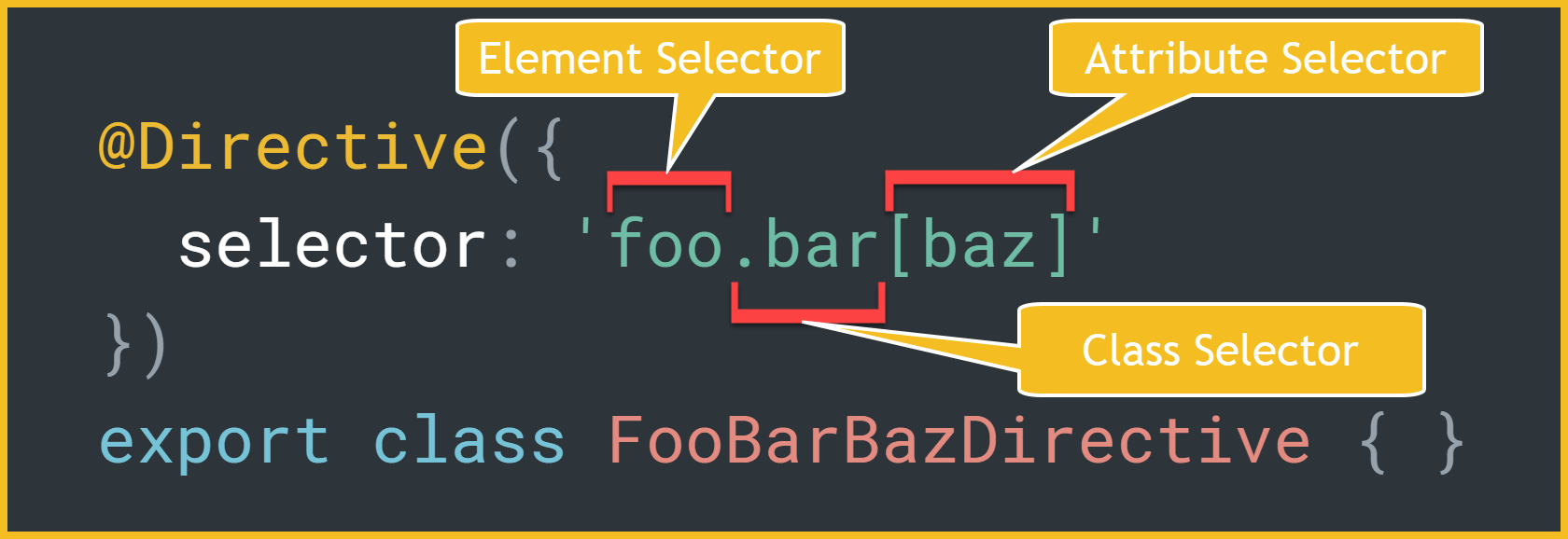
The example above will select all HTML elements that have an element tag named **foo** and an attribute named **baz**. Notice there is no space between the each of the individual selectors, only the token the square brackets is used to delineated between the selectors.

The only caveat with the order of selectors is a class selector cannot be succeeded by an element selector, as there is no symbol to separate the two selectors. Also a element name selector can only be specified once otherwise the directive will not find any elements.



The above example will select all HTML elements with an element name of **foo** and a CSS class name containing **bar**. The order cannot be reversed otherwise the selector would be querying for elements with a class name of **barfoo**, which is not desired.

Chaining selectors can only include a single element name selector with zero or more of either class name or attribute selectors.



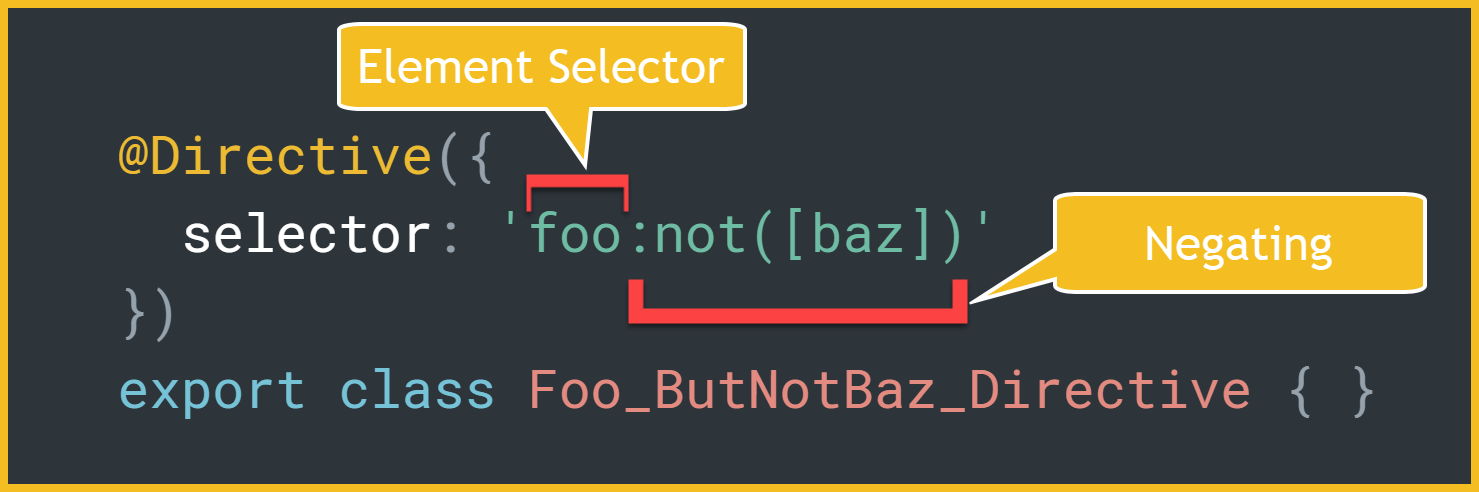
The above example will select all HTML elements with an element name of **foo,**a CSS class name containing **bar** and an attribute named **baz**.

# **Exclusion Selectors**

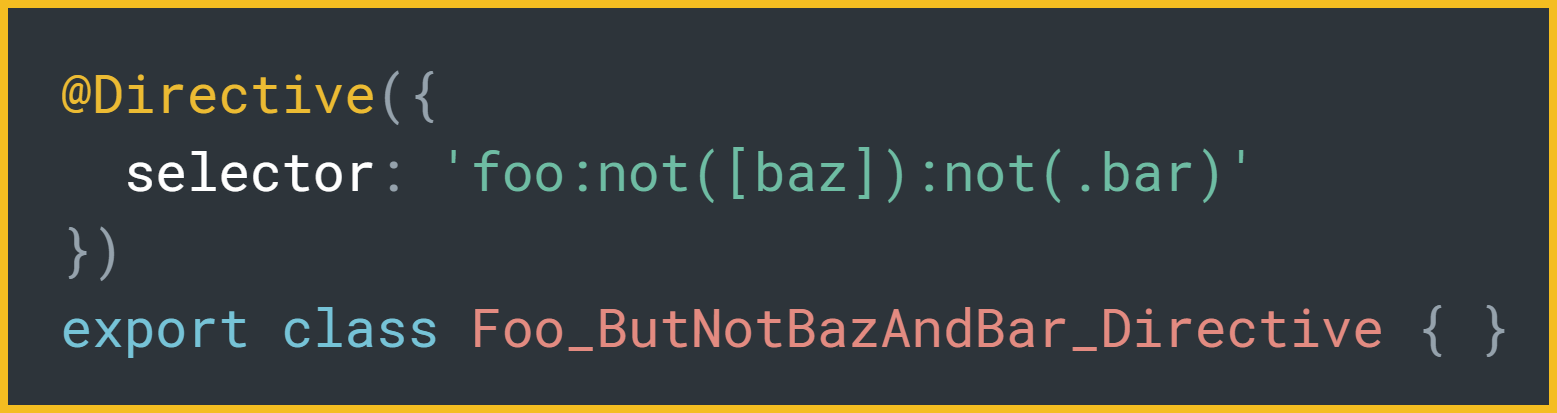
Querying for specific HTML elements sometimes there is a need to exclude specific elements that contain either a class name or attribute.

To exclude specific HTML elements use the keyword **:not(…)** with the specific selector inserted between the round brackets.

The example below will select all HTML elements with an element name of **foo** but exclude those with an attribute named **baz**.



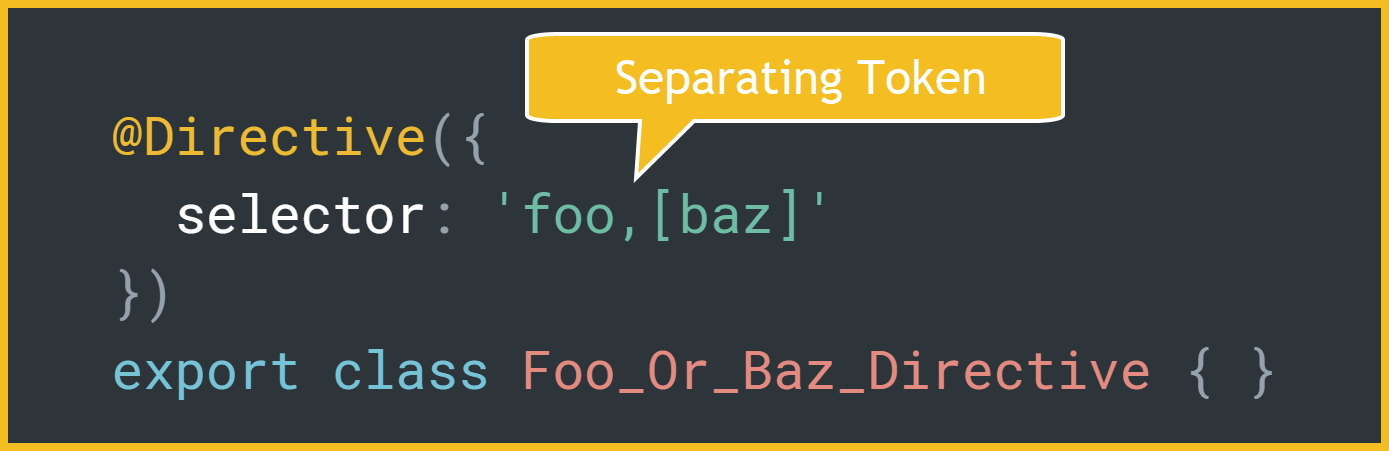
Exclusion selectors can be chained together to further refine the query.



The example above will select all HTML elements with an element name of **foo** but exclude those with an attribute name **baz** and a CSS class containing **bar**.

## Combining Selectors ( or )

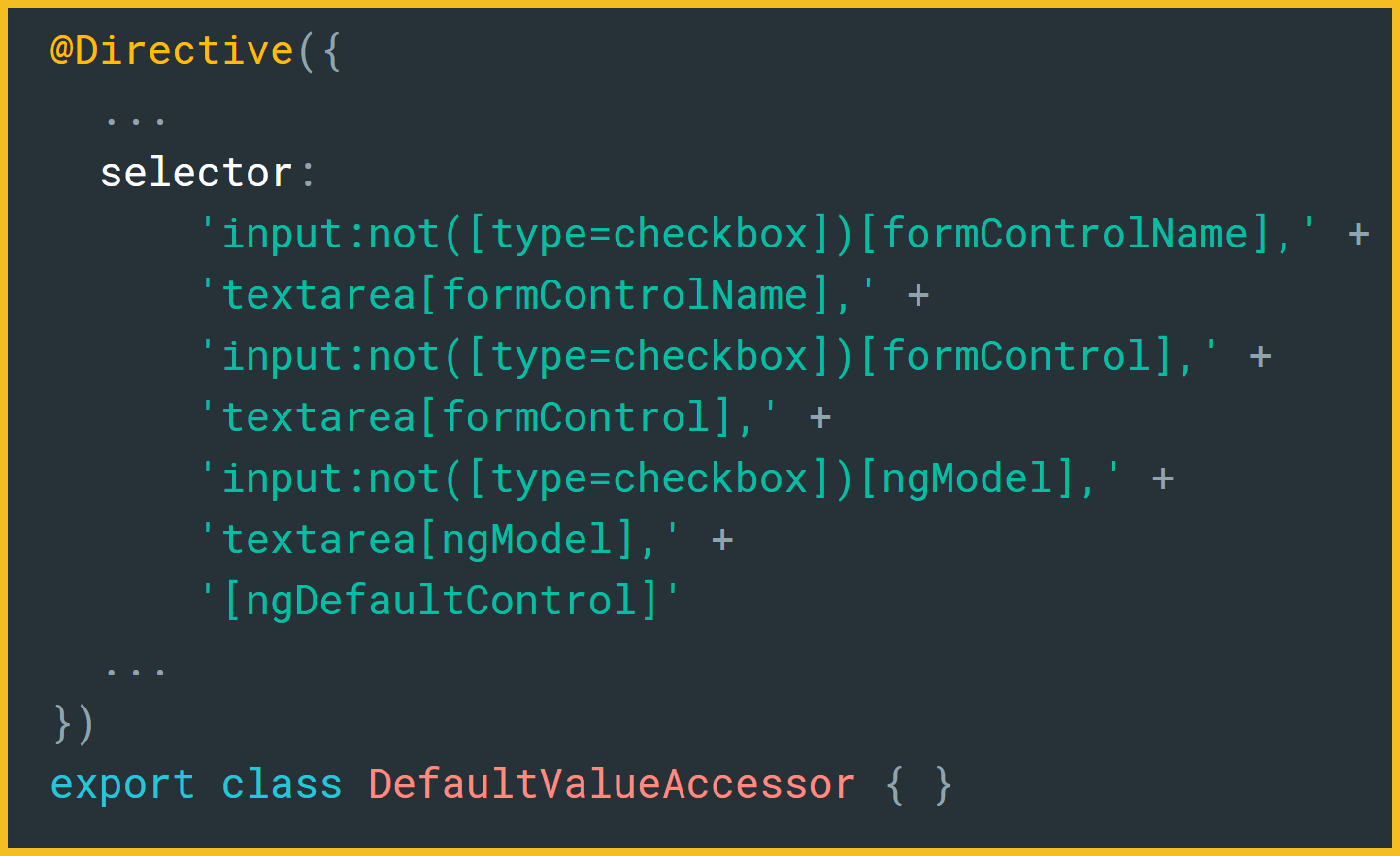
All the selectors combinations above can be combined together in groups using a comma as the separating token to create a OR like query. There is no limit on the number of selectors that can be combined together for targeting of the directive. Where their are multiple possible matches found, only a single instance of the directive will be instantiated for the HTML DOM element.



The example above will select all HTML elements with an element name of **foo** or with an attribute named **baz**. If a HTML element contains both matching criteria the directive will only be attached once.

## Angular Forms Directive Example

Directives are used to implement many of the features that we use in forms and other parts of the angular platform.



The example above is the **DefaultValueAccessor** directive extracted from the forms module. The selector combines all of the various selectors (element, attribute), chaining selectors, exclusion selectors and combining these selectors to target various HTML elements.

Let’s break this selector down into various parts:

* Select **input** elements with **formControlName** attribute but not with attribute with attribute **type=checkbox**.
* Select **textarea** elements with **formControlName** attribute.
* Select **input** elements with **formControl** attribute but not with attribute with attribute **type=checkbox**.
* Select **textarea** elements with **formControl** attribute.
* Select **input** elements with **ngModel**attribute but not with attribute with attribute **type=checkbox**.
* Select **textarea** elements with **ngModel**attribute.
* Select elements with **ngDefaultControl** attribute.