Starting from **Angular2** to the latest editions of i.e. **Angular8, Angular9 & Angular10.**

This Tutorial Applies to Angular 2 Tutorial, Angular 7 Tutorial, Angular 8 Tutorial, Angular 9 Tutorial, Angular 10 Tutorial. Angular 11 Tutorial

## **What is Angular :** The Angular is a development platform for building a Single Page Application for mobile and desktop. It uses Typescript & HTML to build Apps. The Angular itself is written using the Typescript. It now comes with every feature you need to build a complex and sophisticated web or mobile application. It comes with features like**[component](https://www.tektutorialshub.com/angular/angular-components/), [Directives](https://www.tektutorialshub.com/angular/angular-directives/), [Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/), [Pipes](https://www.tektutorialshub.com/angular/angular-pipes/), [HTTP Services](https://www.tektutorialshub.com/angular/angular-httpclient/), [Dependency Injection](https://www.tektutorialshub.com/angular/angular-dependency-injection/)**, etc.

### **Angular Versions :** The early version of the Angular was named as Angular2.  Then later it was renamed to just Angular. Then Angular Team releases new versions of the Angular versions Regularly and the last stable version that is available in Angular 10.0.12

The Angular 9 is released on 06.02.2020. The Angular 9 is a major release. This release switches to ivy compiler and runtime by default. The ivy brings smaller bundle sizes, faster & better testing debugging, Improved type checking, build times, etc

Angular 10 is released on 24-06-2020.

#### **Version History :**

| **Angular Version** | **Date** | **Description** |
| --- | --- | --- |
| Angular 2 | 14.09.2016 | Initial Version of Angular |
| Angular 4 | 23.03.2017 | Version 4 |
| Angular 5 | 11.11.2017 | Version 5 |
| Angular 6 | 03-05-2018 | Version 6 |
| Angular 7 | 18-10-2018 | Version 7 |
| Angular 8 | 25-08-2019 | Version 8 |
| Angular 9 | 06-02-2020 | Version 9 |
| Angular 10 | 24-06-2020 | Version 10 |
| Angular 10.0.12 | 24-08-2020 | Version 10.0.12 |

## **Prerequisites :** The Angular tutorial requires a working knowledge of Javascript, HTML & CSS**.** It also requires the concept of OOP.

We are going to use Typescript as our language. If you have knowledge of C# or Java, then you would find it very easy.

Angular is a UI framework for building mobile and desktop web applications. It is built using Javascript. Using Angular you can build amazing client-side applications using HTML, CSS, and Typescript. It is very important to know how the Angular framework works before you start using it.

**Angular 2**is now known as Angular is the new front end framework and is the successor to the most popular AngularJs. The Angular is an open-source and helps us build dynamic & single-page applications (SPAs).

Angular has many improvements over AngularJS. It has lots of innovations, which makes it easy to learn and develop enterprise-scale applications. You can build extendable, Maintainable, Testable and Standardized Applications using Angular.

**Features of Angular :** Some of the features are listed below :

* **Two-Way [Data Binding](https://www.tektutorialshub.com/angular/angular-data-binding/)**This is the coolest feature of the Angular. Data binding is automatic and fast. changes made in the View is automatically updated in the component class and vice versa
* **Powerful [Routing](https://www.tektutorialshub.com/angular/angular-routing-navigation/) Support**  
  The Angular Powerful routing engine loads the page asynchronously on the same page enabling us to create a Single Page Applications.
* **Expressive HTML**  
  Angular enables us to use programming constructs like if conditions, for loops, etc to render and control how the HTML pages.
* **Modular by Design**Angular follows the modular design. You can create [Angular modules](https://www.tektutorialshub.com/angular/angular-modules/) to better organize and manage our codebase
* **Built-in Back End Support**  
  Angular has built-in support to communicate with the back-end servers and execute any business logic or retrieve data
* **Active Community**  
  Angular is Supported by google and has a very good active community of supporters. This makes a lot of difference as your queries are quickly resolved.

Angular has changed massively from the AngularJS. Angular completely redesigned from scratch. There are many concepts of angularJS that have changed in Angular.

### **Key differences between AngularJs & Angular :**

#### **Support for ES6 :**Angular is completely written in Typescript and meets the ECMAScript 6 specification. This means that it has support for ES6 Modules, Class frameworks, etc.

#### **Components are new controllers:** In AngularJS we had Controllers. In Angular Controllers are replaced with [Angular Components](https://www.tektutorialshub.com/angular/angular-components/).In Angular, we are using Components. The simple component is written using Typescript.

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

@Component({

selector: 'app',

template: '<h1>{{message}} </h1>'

})

export class AppComponent {

message: string=’Hello Angular’;

}

The components can have child components and parent components.

#### **Directives :**The AngularJS had a lot of directives. Some of the most used directives are ng-repeat & ng-if

<ul>

    <li ng-repeat =customer **in** vm.customers>

        {{customer.name}}

    </li>

</ul>

<div ng-**if**=”vm.isVIP”>     <h3> VIP Customer </h3> </div>

#### The Angular also has [directives,](https://www.tektutorialshub.com/angular/angular-directives/) but with a different syntax. It has a \* before the directive name indicating it as a structural directive.

<ul>

    <li \*ngFor =#customer of customers>

        {{customer.name}}

    </li>

</ul>

<div \*ngIf=”vm.isVIP”>

    <h3> VIP Customer </h3>

</div>

The style directives like ng-style, ng-src & ng-href are all gone. These are now replaced by property binding HTML elements to the class properties

We can create a Custom Directives.

@Directive({

   selector: '[MyDirective]'

})

**class** MyDirective { }

#### **Data Bindings :**The powerful angular data bindings stay the same,  with minor syntax changes.

##### **Interpolation**

*//AngularJS*

<h3> {{vm.customer.Name}}</h3>

*//Angular*

<h3> {{customer.Name}}</h3>

**One way Binding :** The Angular can bind to any property of the HTML element.

*//AngularJS*

<h3> ng-bind=vm.customer.name></h3>

*//Angular*

<h3 [innerText]=”customer.name” ></h3>

##### **Event Binding :** The AngularJS uses the ngClick directive to bind to the event. In Angular ngClick Directive is removed. You can bind directly to the DOM events.

*//AngularJS*

<button ng-click=”vm.save()”>Save<button>

*//Angular*

<button (click)=”save()”>Save<button>

##### **Two- way binding**

*//AngularJS*

<input ng-model=”vm.customer.name”>

*//Angular*

<input [(ng-model)]=”customer.name”>

**$scopes are out :**Angular is not using $scope anymore to glue view and controller.

AngularJS used to run a dirty checking on the scope objects to see if any changes occurred. Then it triggers the watchers. And then it used to re-running the dirty checking again.

The Angular is using zone.js to detect changes. Zone.js apply patches on all the global asynchronous operations like click event, timer events, HTTP requests, etc. It then intimates the Angular, whenever the changes occur in Angular Application. The Angular then runs the change detection for the entire application

**Filters are renamed to Pipes :**In AngularJS, we used Filters and as shown below

<td>{{vn.customer.name | uppercase}}</td>

Angular uses the same syntax but names them as [pipes](https://www.tektutorialshub.com/angular/angular-pipes/)

td>{{customer.name | uppercase}}</td>

**Platform-specific Bootstrap :** In AngularJS we used the ng-app directive in our HTML, then the Angular would bootstrap and attach itself the ng-app

<body ng-app=’app’> </html>

The bootstrapping in Angular is done through code. The bootstrapping of Angular is not simple as that of AngularJS. The sample code below shows how Angular application bootstraps the AppModule using platformBrowserDynamic Module

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import {AppModule } from './app.module'; platformBrowserDynamic().bootstrapModule(AppModule);

The Bootstrap is also Platform-specific in Angular. You can have different bootstrapper for mobile & Web application.

**Services :** The AngularJS had Services, Factories, Providers, Constants and values, which used to create reusable code. These are then injected into Controllers so that it can use it.

**Mobile Support :** AngularJS was not built with mobile support in mind. Angular designed with mobile development in mind.

**NOTE :** The Angular has gone through a lot of changes since the version Angular 2. From the Angular 7 version, installing and creating a new project in angular has become very simple. The only thing you need to do in install and Visual Studio code, NPM Package manager & Angular CLI. Once, you install the required dependencies, creating a new project is as easy as running a simple command ng new. [Angular CLI](https://www.tektutorialshub.com/angular/angular-cli-tutorial/) takes care of Configuration & initialization of various libraries.

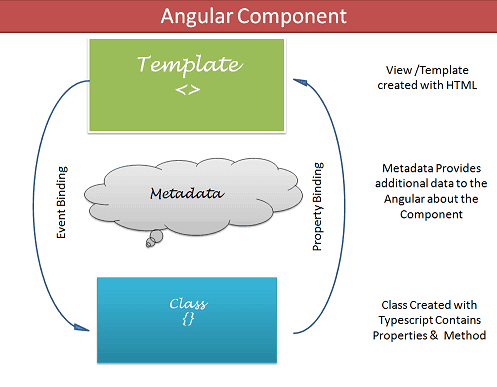
**Components :** The Component is the main building block of an Angular Application. A Component contains the definition of the View and the data that defines how the View looks and behaves.  The Angular Components are plain javascript classes and defined using @component Decorator. This Decorator provides the component with the View to display & Metadata about the class

The Component passes the data to the view using a process called Data Binding. This is done by Binding the DOM Elements to component properties. Binding can be used to display component class property values to the user, change element styles, respond to a user event, etc.

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

* Template
* Class
* MetaData



**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.

The Templates are created with HTML. You can add [Angular directives](https://www.tektutorialshub.com/angular/angular-directives/) and bindings on the template.

There are two ways you can specify the Template in Angular.

* Defining the Template Inline
* Provide an external Template

Class : The class is the code associated with Template (View). The Class is created with the Typescript. Class Contains the Properties & Methods.

The Properties of a class can be bind to the view using Data Binding.The Component classes in Angular are prefixed with the name “Component”, To easily identify them.

export **class** AppComponent{

    title : **string** ="app"

}

**Metadata** : Metadata Provides additional information about the component to the Angular. Angular uses this information to process the class. The Metadata is defined with a decorator.

A decorator is a function that adds metadata to class, its methods & to its properties. The Components are defined with a @component class decorator. It is @component decorator, which defines the class as Component to the Angular

#### **@Component decorator** : A class becomes a Component when Component Decorator is used. A Decorator is always prefixed with @. The Decorator must be positioned immediately before the class definition. We can also build our own decorators.

**Important Component metadata properties**

**Selector :**Selector specifies the simple CSS selector, where our view representing the component is placed by the Angular.The angular places the view (template) inside the selector app-root

**Providers :**The Providers are the services, that our component going to use. The Services provide service to the Components or to the other Services.

**Directives :**The[directives](https://www.tektutorialshub.com/angular/angular-directives/) that this component going to use are listed here.

**Styles/styleUrls :**The CSS Styles or style sheets, that this component needs. Here we can use either external stylesheet (using styleUrls) or inline styles (using Styles). The styles used here are specific to the component

**template/templateUrl :**The HTML template that defines our View. It tells Angular how to render the Component’s view. The templates can be inline (using a template) or we can use an external template (using a templateUrl). The Component can have only one template. You can either use inline template or external template and not both

The creation of the Angular component requires you to follow these steps

1. Create the Component file
2. Import the required external Classes/Functions
3. Create the Component class and export it
4. Add @Component decorator
5. Add metadata to @Component decorator
6. Create the Template
7. Create the CSS Styles
8. Register the Component in Angular Module

app.component.ts

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

styleUrls: ['./app.component.css']

})

export class AppComponent {

title = 'app';

}

**Angular Module :**

import { BrowserModule } from '@angular/platform-browser';

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

**import { AppComponent } from './app.component';**

@NgModule({

declarations: [

**AppComponent**

],

imports: [

BrowserModule

],

providers: [],

**bootstrap: [AppComponent]**

})

export class AppModule { }

We use @NgModule class decorator to define a Module and provide metadata about the Modules.

We add all the components, pipes and directives that are part of this module to the declarations array. We add all the other modules that are used by this module to imports array. We include the services in the providers’ array.

The Component that angular should load, when the app.module is loaded is assigned to the bootstrap property.

We want appComponent to be loaded when Angular starts, thus we assign it to bootstrap property

bootstrap: [AppComponent]

**Creating the inline Template & StyleUrls :** In the above example, we have used the external template & Styles.

We can also specify the Template, Styles inline using the template and styles property of @Component metadata as shown below.

@Component({

  selector: 'app-root',

  template: '<h1> {{title}} works </h1>',

  styles: ['h1 { font-weight: bold; }']

})

export **class** AppComponent {

  title = 'app';

}

In the case of a Multi-line template, you can use BackTicks ( ` ) to enclose the template string.

**The component selector :** The Angular renders the components view in the DOM inside the CSS selector, that we defined in the Component decorator

@Component({

selector: 'app-root',

The selector <app-root></app-root> is in the index.html (under src folder)

 <body>

    <app-root></app-root>

  </body>

When we build Angular Components, we are actually building new HTML elements. We specify the name of the HTML element in the selector property of the component metadata.  And then we use it in our HTML.

The Angular, when instantiating the component, searches for the selector in the HTML file and renders the Template associated with the component.

**Using the CSS class name**

@Component({

selector: '.app-root'

})

**And in the HTML markup use the CSS class name**

<div class="app-root"></div>

**Using attribute name**

@Component({ selector: '[app-root]' })

**And you can now the attribute as follows**

<div app-root></div>

**Using attribute name and value**

@Component({

selector: 'div[app=components]'

})

**now you can use it as follows**

<div app="components"></div>

# Data Binding in Angular : They also need to respond to user interactions and react to events. The data binding kees both component & view in sync with each other. We use techniques like **[Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/), [Property Binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/), [Event Binding](https://www.tektutorialshub.com/angular/event-binding-in-angular/) & [Two Way Binding](https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/)** to bind data. We also learn how to use the **[ngModel](https://www.tektutorialshub.com/angular/ngmodel-two-way-data-binding-in-angular/) directive** to achieve the two-way binding in [Angular Forms](https://www.tektutorialshub.com/angular/angular-forms-fundamentals/).

Data binding is a technique, where the data stays in sync between the component and the view. Whenever the user updates the data in the view, Angular updates the component. When the component gets new data, the Angular updates the view.

There are many uses of data binding. You can show models to the user, dynamically Change element style, respond to user events, etc.

**Data Binding in Angular :** The data binding in Angular can be broadly classified into two groups. **One way binding** or **two-way binding**

## **One way binding :**In one way binding data flows from one direction. Either from view to component or from component to view.

**From Component to View :**To bind data from component to view, we make use of **Interpolation** & **Property Binding**.

#### **Interpolation : [Interpolation](https://www.tektutorialshub.com/angular/interpolation-in-angular/)** allows us to include expressions as part of any string literal, which we use in our HTML. The angular evaluates the expressions into a string and replaces it in the original string and updates the view. You can use interpolation wherever you use a string literal in the view

The Angular uses the {{ }} (double curly braces) in the template to denote the interpolation. The syntax is as shown below

**{{ templateExpression }}**

The content inside the double braces is called **Template Expression**

The Angular first evaluates the Template Expression and converts it into a string. Then it replaces Template expression with the result in the original string in the HTML. Whenever the template expression changes, the Angular updates the original string again

**Example**

Welcome, {{firstName}} {{lastName}}

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html', styleUrls: ['./app.component.css']

})

export class AppComponent {

firstName= 'Sachin'; lastName=”Tendulkar”

}

Run the app and you will see Welcome, Sachin Tendulkar in the output. The Angular replaces both {{firstName}} & {{lastName}} with the values of firstName & lastName variable from the component.

Also, whenever the values of firstName & lastName change, Angular updates the view. But not the other way around.

**Property binding :** The [Property binding](https://www.tektutorialshub.com/angular/property-binding-in-angular/) allows us to bind HTML element property to a property in the component. Whenever the value of the component changes, the Angular updates the element property in the View. You can set the properties such as class, href, src, textContent, etc using property binding. You can also use it to set the properties of custom components or directives (properties decorated with @Input).

The Property Binding uses the following Syntax :

**[binding-target]=”binding-source”**

The binding-target (or target property) is enclosed in a square bracket []. It should match the name of the property of the enclosing element.

Binding-source is enclosed in quotes and we assign it to the binding-target. The Binding source must be a template expression. It can be property in the component, method in component, a template reference variable or an expression containing all of them.

Whenever the value of Binding-source changes, the view is updated by the Angular.

<h1 [innerText]="title"></h1>

<button [disabled]="isDisabled">I am disabled</button>

export **class** AppComponent {

  title="Angular Property Binding Example"

*//Example 1*

  isDisabled= **true**;

 }

The title property of the component class is bound to the innerText property of the h1 tag. Disabled Property of the button is bound to the isDisabled Property of the component

Whenever we modify the title or isDisabled in the component, the Angular automatically updates the view.

The property binding has special syntaxes for setting the class & styles. Also, both interpolation & property binding does not set the attributes of the HTML elements. Hence we have an attribute binding to such situations

##### **Class Binding :** You can set the class in the following ways. Click on the links to find out more

* ClassName Property binding
* Set the Class attribute with class binding
* ngClass directive

##### **Style Binding** :You can set the class in the following ways. Click on the links to find out more

* [Style Binding](https://www.tektutorialshub.com/angular/angular-style-binding/)
* [ngStyle directive](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)

##### **Attribute binding :**Sometimes there is no HTML element property to bind to. The examples are [aria](https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA) (accessibility) Attributes & [SVG](https://developer.mozilla.org/en-US/docs/Web/SVG). In such cases, you can make use of attribute binding

The attribute syntax starts with attr followed by a dot and then the name of the attribute as shown below :

<button [attr.aria-label]="closeLabel" (onclick)="closeMe()">X</button>

**Read more :**

1. **[ngClass Directive](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)**
2. **[Style Binding](https://www.tektutorialshub.com/angular/angular-style-binding/)**
3. **[ngStyle directive](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)**

status:**string**='error';

<button [style.color]="status=='error' ? 'red': 'blue'">Button 1</button>

getColor() {

**return** 'yellow';

}

<button [style.color]="getColor()">Button 2</button>

<p [style.color]="getColor()"

   [style.font-size.px]="'20'"

   [style.background-color]="status=='error' ? 'red': 'blue'">

   paragraph with multiple styles

</p>

**The style property name can be written in either dash-case (font-size), as shown in above example, or camelCase (fontSize)**

**From View to Component :**

**Event Binding :**Event binding allows us to bind events such as keystroke, clicks, hover, touch, etc to a method in component. It is one way from view to component.

**For Example,** when the user changes a input in a text box, we can update the model in the component, run some validations, etc. When the user submits the button, we can then save the model to the backend server.

Angular uses the following syntax for event binding

<target-event)="TemplateStatement"

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.

<button (click)="onSave()">Save</button>

The above example, binds the click event of a button to a onSave() method in the component class. Whenever user clicks on the button, the Angular invokes the onSave() method.

**Two Way binding :** Two-way binding means that changes made to our model in the component are propagated to the view and that any changes made in the view are immediately updated in the underlying component

Two-way binding is useful in data entry forms. Whenever a user makes changes to a form field, we would like to update our model. Similarly, when we update the model with new data, we would like to update the view as well

The two-way binding uses the special syntax known as a banana in a box [()]

**<someElement [(someProperty)]="value"></someElement>.**

The above syntax sets up both property binding & event binding. But to make use of it, the property must have the change event with the name <propertyName>Change

But, angular has a special directive ngModel, which sets up the two-way binding

**ngModel :**The Angular uses the ngModel directive to achieve the two-way binding on HTML Form elements. It binds to a form element like input, select, selectarea. etc.

The ngModel directive is not part of the Angular Core library. It is part of the @angular/forms. You need to import the FormsModule package into your Angular module.

import { FormsModule } from '@angular/forms';

Then you can use it using the two-way binding syntax as shown below :

<input type="text" name="value" [(ngModel)]="value">

When you bind to a ngModel directive, behind the scene it sets up property binding & event binding. It binds to the value property of the element using property binding. It then uses the ngModelChange event to sets up the event binding to listen to the changes to the value.

### IMG_256

The two way data binding nothing but both property binding & event binding applied together. Property Binding is one way from view to component. The event binding is one way from component to view. If we combine both we will get the Two-way binding.

h2>Example 1</h2>

<input type="text" [value]="name" (input)="name=$event.target.value">

<p> You entered {{name}}</p>

<button (click)="clearName()">Clear</button>

name=""

clearName() {

**this**.name="";

}

We bind the name property to the input element ([value]="name"). We also use the event binding (input)="name=$event.target.value". It updates the name property whenever the input changes. The Angular interpolation updates the {{name}}, so we know the value of name property.

**ngModel Example**

Import FormsModule

Open the app.module.ts and make the following changes

import { FormsModule } from '@angular/forms';

**Template**

<h2>Example 2</h2>

<input type="text" name="value" [(ngModel)]="value">

<p> You entered {{value}}</p>

<button (click)="clearValue()">Clear</button>

Component

value="";

clearValue() {

this.value="";

}

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

Run the project and see that as you modify the name, the component class model is automatically updated.

**Custom Two-way binding :**

As we mentioned earlier the [()] to work, we need to have a property with the change event as<nameofProperty>Change.

We do not have any HTML Elements which follows the above naming conventions, but we can create a custom component

create a new component and name it as counter.component.ts. Copy the following code.

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

@Component({

selector: 'counter',

template: `

<div>

<p>

Count: {{ count }}

<button (click)="increment()">Increment</button>

</p>

</div>

`

})

export class CounterComponent {

@Input() count: number = 0;

@Output() countChange: EventEmitter<number> = new EventEmitter<number>();

increment() {

this.count++;

this.countChange.emit(this.count);

}

}

The component has two properties one is input property count decorated with @Input(). The other in is an event (or output property), which we decorate with @Output(). We name the input property as count. Hence the output property becomes countChange

Now we can use this component and create two-way binding to the count property using the syntax [(count)].

<h2>Example 3</h2>

<counter [(count)]="count"></counter>

<p> Current Count {{count}}</p>

<button (click)="clearCount()">Clear</button>

**Directives :** The [Angular directive](https://www.tektutorialshub.com/angular/angular-directives/) helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the directives. They help you to extend HTML. The [Angular directives](https://www.tektutorialshub.com/angular/angular-directives/) are classified into three categories based on how they behave.  They are **Component, Structural and Attribute Directives**.

The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) allows us to Add/Remove DOM Element. It is similar to the switch statement of Javascript.  The [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) allows us to Add/Remove DOM Element.

The [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) Directive is an Angular Attribute Directive, which allows us to add or remove CSS classes to an HTML element. The [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/) directive allows you to modify the style of an HTML element using the expression.  Using the [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/) you can dynamically change the style of your HTML element.

1. [Angular Directives](https://www.tektutorialshub.com/angular/angular-directives/)
2. [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/)
3. [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/)
4. [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/)
5. [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/)
6. [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)
7. [ngFor Trackby](https://www.tektutorialshub.com/angular/angular-track-by-to-improve-ngfor-performance/)
8. [Custom Directive](https://www.tektutorialshub.com/angular/custom-directive-in-angular/)

## **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.

## **ngFor :T**he [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The [ngFor](https://www.tektutorialshub.com/angular/angular-ngfor-directive/) is similar to [ngRepeat](https://docs.angularjs.org/api/ng/directive/ngRepeat) in AngularJS.

<tr \*ngFor="let customer of customers;">

    <td>{{customer.customerNo}}</td>

    <td>{{customer.name}}</td>

    <td>{{customer.address}}</td>

    <td>{{customer.city}}</td>

    <td>{{customer.state}}</td>

</tr>

#### **ngSwitch :**The [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) directive lets you add/remove HTML elements depending on a match expression. [ngSwitch](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/) directive used along with [ngSwitchCase](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/" \l "ngswitchcase) and [ngSwitchDefault](https://www.tektutorialshub.com/angular/angular-ngswitch-directive/" \l "ngswitchdefault).

<div [ngSwitch]="Switch\_Expression">

    <div \*ngSwitchCase="MatchExpression1”> First Template</div>

    <div \*ngSwitchCase="MatchExpression2">Second template</div>

    <div \*ngSwitchCase="MatchExpression3">Third Template</div>

    <div \*ngSwitchCase="MatchExpression4">Third Template</div>

    <div \*ngSwitchDefault?>**Default** Template</div>

</div>

#### **ngIf :** The [ngIf](https://www.tektutorialshub.com/angular/angular-ngif-directive/) Directives is used to add or remove HTML elements based on an expression. The expression must return a boolean value. If the expression is false then the element is removed, else the element is inserted.

<div \*ngIf="condition">

**This** **is** shown **if** condition **is** **true**

</div>

## **Attribute Directives :** An Attribute or style directive can change the appearance or behavior of an element.

#### **ngModel :**The ngModel directive is used the achieve the[two-way data binding](https://www.tektutorialshub.com/angular/angular-data-binding/).

#### **ngClass :**The [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) is used to add or remove the CSS classes from an HTML element. Using the [ngClass](https://www.tektutorialshub.com/angular/angular-ngclass-directive/) one can create dynamic styles in HTML pages.

<div [ngClass]="'first second'">...</div>

#### **ngStyle :** [ngStyle](https://www.tektutorialshub.com/angular/angular-ngstyle-directive/)is used to change the multiple style properties of our HTML elements. We can also bind these properties to values that can be updated by the user or our components.

<div [ngStyle]="{'color': 'blue', 'font-size': '24px', 'font-weight': 'bold'}">

    some text

</div>

**Building Custom Directives :**You can also build custom directives in Angular. The Process is to create a JavaScript class and apply the @Directive attribute to that class. You can write the desired behavior in the class.

**Pipes :** The Angular pipes are used to Transform the Data. For Example, the Date pipe formats the date according to locale rules. We can pass arguments to pipe and chain pipes. The Angular also allows us to create the Custom Pipe.

Angular Pipes takes data as input and formats or transform the data to display in the template. We use them to change the appearance of the data before presenting it to the user. The most common [use case of pipes is displaying the dates](https://www.tektutorialshub.com/angular/formatting-dates-with-angular-date-pipe/) in the correct format as per the user’s locale.

Angular Pipes Syntax

The syntax of the pipe is as follows

**Expression | pipeOperator[:pipeArguments]**

Where

**Expression:** is the expression, which you want to transform

**| :** is the Pipe Character

**pipeOperator :** name of the Pipe

**pipeArguments:** arguments to the Pipe

Angular built in date pipe to transform the date.

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

import { FormsModule } from '@angular/forms';

@Component({

    selector: 'app-root',

    templateUrl: `<p> Unformatted date : {{toDate }} </p>

                  <p> Formatted date : {{toDate | date}} </p>`

})

export **class** AppComponent

{

    title: **string** = 'pipe Example' ;

    toDate: Date = **new** Date();

}

**Chaining Pipes :** Pipes can be chained together to make use of multiple pipes in one expression. For example in the following code, the toDate is passed to the Date Pipe. The output of the Date pipe is then passed to the uppercase pipe.

toDate | date | uppercase

**DatePipe :** The Date pipe formats the date according to locale rules. The syntax of the date pipe is as shown below

date\_expression | date[:format]

Where

date\_expression is a date object or a number

date is the name of the pipe

format is the date and time format string which indicates the format in which date/time components are displayed.

**Example of Datepipe :**

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

import { FormsModule } from '@angular/forms';

@Component({

    selector: 'app-root',

    template:`<p>medium : {{toDate | date:'medium'}} </p>

              <p>**short** : {{toDate | date:'short'}} </p>

              <p>fullDate : {{toDate | date:'fullDate'}} </p>

              <p>longDate : {{toDate | date:'longDate'}} </p>

              <p>mediumDate : {{toDate | date:'mediumDate'}} </p>

              <p>shortDate : {{toDate | date:'shortDate'}} </p>

              <p>mediumTime : {{toDate | date:'mediumTime'}} </p>

              <p>dd-MM-y : {{toDate | date:'dd-MM-y'}} </p>

              <p>dd-MM-yy HH:mm : {{toDate | date:'dd-MM-yy HH:mm'}} </p>`

})

export **class** AppComponent{

    title: **string** = 'Angular pipes Example' ;

    toDate: Date = **new** Date();

}

**UpperCasePipe & LowerCasePipe :** As the name suggests, these pipes transform the string to Uppercase or lowercase.

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

import { FormsModule } from '@angular/forms';

@Component({

    selector: 'app-root',

    template:`<p>Unformatted :{{msg}} </p>

              <p>Uppercase :{{msg | uppercase}} </p>

              <p>Lowercase :{{msg | lowercase}} </p>`

})

export **class** AppComponent{

    title: **string** = 'Angular pipes Example' ;

    msg: **string**= 'Welcome to Angular';

}

**SlicePipe :** Creates a new List or String containing a subset (slice) of the string or array. This Pipe uses the JavaScript API [Array.prototype.slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Array/slice) and [String.prototype.slice()](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/String/slice).

**array\_or\_string\_expression | slice:start[:end]**

Where

**array\_or\_string\_expression** is the string to slice

**slice**is the name of the pipe

**start** is the start position/index from where the slicing will start

**end** is the ending index/position in the array/string

The slice pipes take two arguments. The first argument start is the starting index of the string/array. The second argument end is the ending index of the string/array. If the start or end index is negative then the index is counted from end of the string/array

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

import { FormsModule } from '@angular/forms';

@Component({

    selector: 'app-root',

    template:`<p>Complete **String** :{{msg}} </p>

              <p>Example 1 :{{msg | slice:11:20}} </p>

              <p>Example 2 :{{msg | slice:-9}} </p>`

})

export **class** AppComponent

{

    title: **string** = 'Angular pipes Example' ;

    msg: **string**= 'Welcome to Angular ';

}

**DecimalPipe / NumberPipe :** The Decimal Pipe is used to Format a number as Text. This pipe will format the number according to locale rules.

Syntax

**number\_expression | number[:digitInfo]**

Where

number\_expression is the number you want to format

number is the name of the pipe

digitInfo is a string which has the following format

**{minIntegerDigits}.{minFractionDigits}-{maxFractionDigits}**

Where

**minIntegerDigits** is the minimum number of integer digits to use. Defaults to 1.

**minFractionDigits** is the minimum number of digits after fraction. Defaults to 0.

**maxFractionDigits** is the maximum number of digits after fraction. Defaults to 3.

**PercentePipe :**Formats the given number as a percentage according to locale rules.

### **CurrencyPipe :** Formats a number as currency using locale rules.

# How to Create Custom Pipe in Angular : The [Pipes](https://www.tektutorialshub.com/angular/angular-pipes/) are a great way to transform the appearance of elements in the template. The [Angular](https://www.tektutorialshub.com/angular-tutorial/)comes with some great built-in pipes like Date pipe, Currency pipe, and Number pipe, etc.

To create a custom pipe, first we need to create a pipe class. The pipe class must implement the PipeTransform interface. We also decorate it with @pipe decorator. Give a name to the pipe under name metadata of the @pipe decorator. Finally, we create the transform method, which transforms given value to the desired output.

To create a Custom Pipe, first, You need to follow these steps

1. Create a pipe class
2. Decorate the class with @pipe decorator.
3. Give a name to the pipe in the name meta data of the @pipe decorator. We will use this name in the template.
4. The pipe class must implement the PipeTransform interface. The interfaces contain only one method transform.
5. The first parameter to the transform method is the value to be transferred. The transform method must transform the value and return the result. You can add any number of additional arguments to the transform method.
6. Declare the pipe class in the Angular Module (app.module.ts)
7. Use the custom pipe just as you use other pipes.

## **Temparature Convertor Custom Pipe Example :**Create a new file temp-convertor.pipe.ts. Under the folder src/app. Copy the following code and paste it.

import {Pipe, PipeTransform} from '@angular/core';

@pipe({

    name: 'tempConverter'

})

export **class** TempConverterPipe implements PipeTransform {

    transform(value: number, unit: **string**) {

**if**(value && !isNaN(value)) {

            if (unit === 'C') {

                var temperature = (value - 32) /1.8 ;

**return** temperature.toFixed(2);

            } **else** **if** (unit === 'F'){

**var** temperature = (value \* 1.8 ) + 32

**return** temperature.toFixed(2);

            }

        }

**return**;

    }

}

The PipeTransform interface defines only one method transform. The interface definition is as follows.

interface PipeTransform {

transform(value: any, ...args: any[]): any

}

**Declare the Pipe :** Before using our pipe, we need to tell our component, where to find it. This is done by first by importing it and then including it in declarations array of the AppModule.

**Using the Custom Pipe :** The custom pipes are used in the same as the Angular built-in pipes are used. Add the following HTML code to your app.component.html file

<div **class**="row">

      <h3>Fahrenheit to Celsius </h3>    </div>

    <div **class**="row">

      <p> Fahrenheit : <input type="text" [(ngModel)]="Fahrenheit" />

      Celsius : {{Fahrenheit | tempConverter:'C'}} </p>

    </div>

    <div **class**="row">

      <h3>Celsius to Fahrenheit </h3>    </div>

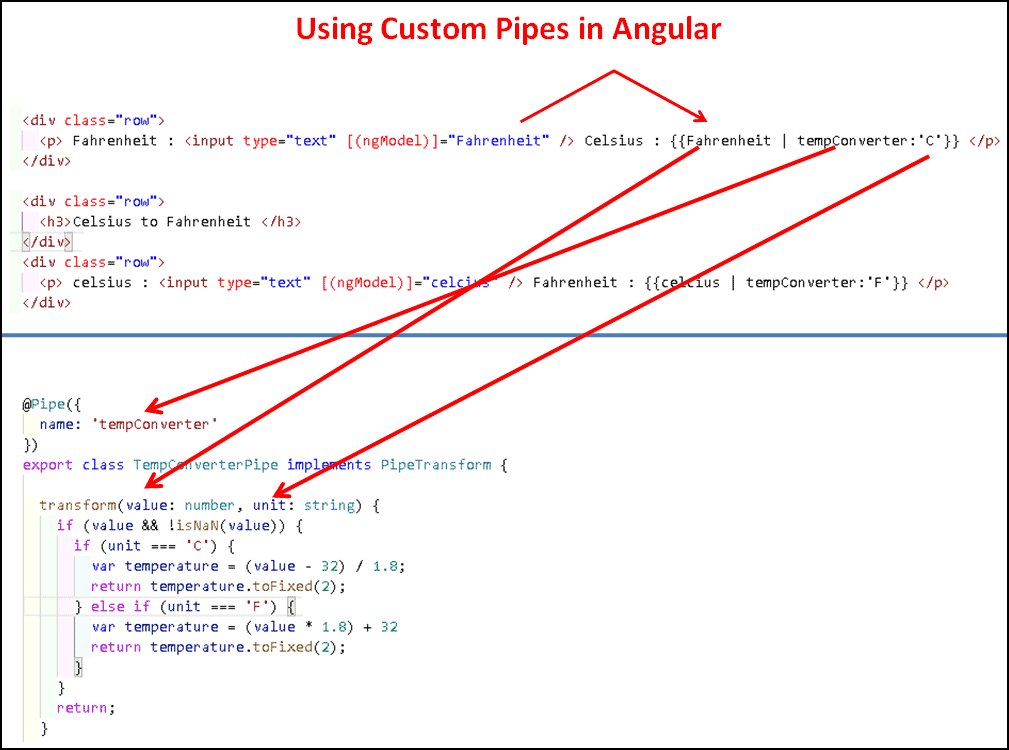
    <div **class**="row">

      <p> celsius : <input type="text" [(ngModel)]="celcius" />

       Fahrenheit : {{celcius | tempConverter:'F'}} </p>

    </div>

We use our pipe as follows. Fahrenheit is sent to the tempConverter as the first argument value. We use the | to indicate that the tempConverter is a pipe to angular. The C after the colon is the first argument. You can pass more than argument to the pipe by separating each argument by a : colon.



# **Child/Nested Components in Angular :** It creates AppComponent, which is the root component of our application. The AppComponent is [bootstrapped](https://www.tektutorialshub.com/angular/angular-bootstrapping-application/) in the AppModule  and loaded in the index.html file using the selector <app-root>Loading...</app-root>.

**What is a Child/Nested Component :** The Angular follows component-based Architecture, where each component manages a specific task or workflow. Each component is an independent block of the reusable unit.

In real life, angular applications will contain many components. The task of the root component is to just host these child components. These child components, in turn, can host the more child components creating a Tree-like structure called Component Tree.

ng **new** childComponent

**Applies to:** Angular 5 to the latest edition i.e. Angular 8, Angular 9. Angular 10, Angular 11

**Tell angular where to display the component :** Finally, we need to inform the Angular, where to display the child Component

We want our child Component as the child of the AppComponent. Open the app.component.html and add the following template

<h1>{{title}}. </h1>

<customer-list></customer-list>

The @Component decorator of the CustomerListComponent , we used the customer-list as the selectorin the metadata for the component. This CSS selector name must match the element tag that specified within the parent component’s template.

**Component Communication :** The Components are useless if they do not share data between them. The Parent Component communicates with the child component using the**@Input Annotation**. The child components detect changes to these Input properties using **OnChanges life Cycle** hook or with a **Property Setter**. The child component can communicate with the parent by **raising an event**, which the parent can listen.

how Angular Passes the data to the child component. The Angular Components communicate with each other using ****@Input Annotation****.

**Angular Pass data to child component :** In Angular, the Parent Component can communicate with the child component by setting its Property. To do that the Child component must expose its properties to the parent component. The Child Component does this by using the @Input decorator.

**In the Child Component**

* Import the @Input module from @angular/Core Library
* Mark those property, which you need data from parent as input property using @Input decorator

**In the Parent Component**

* Bind the Child component property in the Parent Component when instantiating the Child

**@Input Decorator :** The @Input Decorator is used to configure the input properties of the component.

When you mark a property as input property, then the Angular injects values into the component property using [Property Binding](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "property-binding). The Property Binding uses the [] brackets. The Binding Target (Property of the child component) is placed inside the square brackets. The Binding source is enclosed in quotes. [Property binding](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "property-binding) is one way from Component to the Target in the template.

## **@Input example:** Our application will have a counter which is incremented by the Parent Component. The Parent Component then passes this counter to the child component for display in its template.

**The Child Component with @Input Decorator**

Create the ChildComponent.ts under the src folder. And copy the following code

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

@Component({

    selector: 'child-component',

    template: `<h2>Child Component</h2>

               current count **is** {{ count }}

    `

})

export **class** ChildComponent {

    @Input() count: number;

}

* First, we import the @Input decorator from @angular/core.
* We have defined the inline template for the child component, where it displays the current count.
* The Child Component expects the count to come from the Parent Component. Hence in ChildComponent decorate the count property with @Input decorator.

**Bind to Child Property in Parent Component**

Now, time to pass the Count values to the Child Component from the Parent

Open the App.component.ts and copy the following code

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

@Component({

  selector: 'app-root',

  template: `

        <h1>Welcome to {{title}}!</h1>

        <button (click)="increment()">Increment</button>

        <button (click)="decrement()">decrement</button>

        <child-component [count]=Counter></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Component Interaction';

  Counter = 5;

  increment() {

**this**.Counter++;

  }

  decrement() {

**this**.Counter--;

  }

}

we are invoking the child component inside.Here, we are using count property, which is a property of the child Component inside the square bracket. We bind it to Counter property of the Parent Component.

Remember square bracket represents the [Property Binding in Angular](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "property-binding).

**Aliasing input Property :** You can Alias the input property and use the aliased name the parent component as shown below.

export **class** ChildComponent {

     @Input('MyCount') count: number;

}

Here, we are aliasing count property with MyCount alias

In the parent component, we can use the MyCount as shown below..

template: `

   <h1>Welcome to {{title}}!</h1>

   <child-component [MyCount]=ClickCounter></child-component>’

**Detecting the Input changes :**

We looked at how to pass the data from parent to the child using @Input annotation and property binding.

Passing the data to child component is not sufficient, the child Component needs to know when the input changes so that it can act upon it.

There are two ways of detecting when input changes in the child component in Angular

* Using OnChanges LifeCycle Hook
* Using Input Setter

**Using OnChanges LifeCycle Hook :**

ngOnChanges is a lifecycle hook, which angular fires when it detects changes to data bound input property. This method receives a SimpeChanges object, which contains the current and previous property values. We can Intercept input property changes in the child component using this hook.

How to use ngOnChanges for Change Detection

* Import the OnChanges interface, SimpleChanges, SimpleChange from @angule/core library.
* Implement the ngOnChanges() method. The method receives the SimpleChanges object containing the changes each input property.

Let us update our Child component to use the OnChanges hook

Open the Child Component.ts and make the following changes

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

@Component({

    selector: 'child-component',

    template: `<h2>Child Component</h2>

               current count **is** {{ count }}

    `

})

export **class** ChildComponent implements OnChanges {

    @Input() count: number;

    ngOnChanges(changes: SimpleChanges) {

**for** (let property **in** changes) {

**if** (property === 'count') {

              console.log('Previous:', changes[property].previousValue);

              console.log('Current:', changes[property].currentValue);

              console.log('firstChange:', changes[property].firstChange);

            }

        }

    }

}

* First, we are Importing the required libraries.
* Next, Modify the class to implement the Onchanges interface.
* ngOnChanges method.
* This method receives the all the changes made to the input properties as SimpleChanges object. The SimpleChanges object whose keys are property names and values are instances of SimpleChange.
* SimpleChange class Represents a basic change from a previous to a new value. It has three class members.

**Using Input Setter :** We can use the property getter and setter to detect the changes made to the input property as shown below

In the Child Component create a private property called \_count

private \_count = 0;

Create getter & setter on property count and attach @Input annotation. We intercept the input changes from setter function and log it to console

@Input()

set count(count: number) {

this.\_count = count;

console.log(count);

}

get count(): number { return this.\_count; }

**Angular Pass data to parent component :** how to Pass data to Parent Component from Child Component in Angular.

[Parent component communicates with its child](https://www.tektutorialshub.com/angular/angular-passing-data-child-component/) by setting its input property. The Child can send data to Parent by raising an event, Parent can interact with the child via local variable or Parent can call **@ViewChild** on the child.

**Pass data to parent component :** There are three ways in which parent component can interact with the child component

* Parent Listens to Child Event
* Parent uses Local Variable to access the child
* Parent uses a @ViewChild to get reference to the child component

### **Parent listens for child event :** The Child Component exposes an EventEmitter Property. This Property is adorned with the @Output decorator. When Child Component needs to communicate with the parent it raises the event. The Parent Component listens to that event and reacts to it.

**EventEmitter Property :** To Raise an event, the component must declare an EventEmmitter Property. The Event can be emitted by calling the .emit() method.

countChanged: EventEmitter<number> = **new** EventEmitter()

And then call emit method passing the whatever the data you want to send as shown below:

**this**.countChanged.emit(**this**.count);

**@Output Decorator :** Using the EventEmitter Property gives the components ability to raise an event. But to make that event accessible from parent component, you must decorate the property with **@Output** decorator.

How to Pass data to parent component using @Output

**In the child component**

* Declare a property of type EventEmitter and instantiate it
* Mark it with a @Output annotation
* Raise the event passing it with the desired data

**In the Parent Component**

* Bind to the Child Component using [Event Binding](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "Event-Binding) and listen to the child events
* Define the event handler function.

**Child Component**

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

@Component({

    selector: 'child-component',

    template: `<h2>**Child** Component</h2>

               <button (click)="increment()">Increment</button>

               <button (click)="decrement()">decrement</button>

               current count **is** {{ count }}

    `

})

export **class** ChildComponent {

    @Input() count: number;

    @Output() countChanged: EventEmitter<number> =   **new** EventEmitter();

    increment() {

**this**.count++;

**this**.countChanged.emit(**this**.count);

      }

    decrement() {

**this**.count--;

**this**.countChanged.emit(**this**.count);

    }

}

**Parent Component :** In the parent component , we need to listen to the “countChanged” event

The “countChanged” event is enclosed in Parentheses. It is then assigned to the method “countChangedHandler” in the component class. The syntax is similar to [Event Binding](https://www.tektutorialshub.com/angular/angular-data-binding/" \l "Event-Binding).

<child-component [count]=ClickCounter (countChanged)="countChangedHandler($event)"></child-component>`

countChangedHandler(count: number) {

**this**.ClickCounter = count;

    console.log(count);

  }

--------------------------

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

@Component({

  selector: 'app-root',

  template: `

        <h1>Welcome to {{title}}!</h1>

        <p> current count **is** {{ClickCounter}} </p>

        <child-component [count]=Counter (countChanged)="countChangedHandler($event)"></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Component Interaction';

  Counter = 5;

  countChangedHandler(count: number) {

**this**.Counter = count;

    console.log(count);

  }

}

**Parent uses a @ViewChild() to get reference to the Child Component :**

Injecting an instance of the child component into the parent as a @ViewChild is the another technique used by the parent to access the property and method of the child component

The @ViewChild decorator takes the name of the component/directive as its input. It is then used to decorate a property. The Angular then injects the reference of the component to the Property

**Child Component :** There is no change in the child component.

#### **Parent Component :**

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

import { ChildComponent } from './child.component';

@Component({

  selector: 'app-root',

  template: `

        <h1>{{title}}</h1>

        <p> current count **is** {{child.count}} </p>

        <button (click)="increment()">Increment</button>

        <button (click)="decrement()">decrement</button>

        <child-component></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Parent calls an @ViewChild()';

  @ViewChild(ChildComponent) child: ChildComponent;

  increment() {

**this**.child.increment();

  }

  decrement() {

**this**.child.decrement();

  }

**Parent uses local variable to access the Child in Template :** Parent Template can access the child component properties and methods by creating the template reference variable.

#### **Child Component :**

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

@Component({

    selector: 'child-component',

    template: `<h2>Child Component</h2>

               current count **is** {{ count }}

    `

})

export **class** ChildComponent {

    count = 0;

     increment() {

**this**.count++;

      }

    decrement() {

**this**.count--;

    }

}

We have removed the input, output & eventemiitter.

Our component is now have property count and two methods to increment and decrement it

**Parent component :**

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

@Component({

  selector: 'app-root',

  template: `

        <h1>{{title}}!</h1>

        <p> current count **is** {{child.count}} </p>

        <button (click)="child.increment()">Increment</button>

        <button (click)="child.decrement()">decrement</button>

        <child-component #child></child-component>` ,

  styleUrls: ['./app.component.css']

})

export **class** AppComponent {

  title = 'Parent interacts with child via local variable';

}

We have created a local variable, #child, on the tag <child-component>. The “child” is called template reference variable, which now represents the child component

The Template Reference variable is created, when you use #<varibaleName> and attach it to a DOM element. You can then, use the variable to reference the DOM element in your Template

The local variable approach is simple and easy. But it is limited because the parent-child wiring must be done entirely within the parent template. The parent component itself has no access to the child.

You can’t use the local variable technique if an instance of the parent component class must read or write child component values or must call child component methods.

**Component Life Cycle Hook :**