@Input and @Output:

@Input is used to define an input property to achieve component property binding. @Input decorator binds a property within one component (child component) to receive a value from another component (parent component). This is one way communication from parent to child. The component property should be annotated with @Input decorator to act as input property. It can be annotated at any type of property such as number, string, array or user defined class. To use alias for the binding property name we need to assign an alias name as @Input(alias).

Find the use of @Input with string data type.

@Input()

ctMsg : string;

Now find array data type with @Input decorator. Here we are aliasing the property name. In the component property binding, alias name ctArray will be used.

@Input('ctArray')

myctArray : Array<string>

Now find @Input decorator with a property of user defined class type.

@Input('stdLeader')

myStdLeader : Student;

@Output is used to define output property to achieve custom event binding. @Output decorator binds a property of a component to send data from one component (child component) to calling component (parent component). This is one way communication from child to parent component. @Output binds a property of the type of angular EventEmitter class. This property name becomes custom event name for calling component.

Find the @Output decorator using aliasing.

@Output('addStudentEvent')

addStdEvent = new EventEmitter<Student>();

In the above code snippet addStudentEvent will become custom event name. Now find @Output decorator without aliasing.

@Output()

sendMsgEvent = new EventEmitter<string>();

Here sendMsgEvent will be custom event name.

### Aliasing Input and Output properties

@Input and @Output decorate input and output properties. @Input can alias input property name and @Output can alias output property name.   
**1. Aliasing input property using @Input**  
To alias input property use an alias as @Input(alias). Find code snippet.

<child-two [stdLeader] = "stdLeaderObj"> </child-two>

Here we are doing property binding. stdLeaderObj is the property of parent component. stdLeader is the alias of child component property. Now find the below ode snippet.

@Input('stdLeader')

myStdLeader : Student;

What we achieve that stdLeader is the alias of myStdLeader property.   
**2. Aliasing output property using @Output**  
To alias output property use alias as @Output(alias). Find the custom event binding as below.

<child-two (addNumberEvent) = "printSum($event)" > </child-two>

Here addNumberEvent is a custom event name. When this event will invoke, printSum() method will be executed. Now find the code snippet.

@Output('addNumberEvent')

addNumEvent = new EventEmitter<number>();

What we achieve here is that addNumberEvent is the alias of addNumEvent.

### Component Property Binding using @Input

Find the steps for component property binding using @Input decorator step by step.   
1. In the parent component, first create a property. Here we are creating a property of our class Student type.

stdLeaderObj = new Student('Narendra', 'Modi');

2. Create a custom element in parent component that is a selector of one of our child component. Here we will perform component property binding.

<child-two [stdLeader] = "stdLeaderObj"> </child-two>

3. Use @Input decorator to declare child component property as an input property that will receive value from parent using component property binding. Here we are using aliasing for property name.

@Input('stdLeader')

myStdLeader : Student;

4. Now we are ready to fetch values from input component property in our child component.

{{myStdLeader.fname +' '+ myStdLeader.lname}}

### Custom Event Binding using @Output and EventEmitter

Here we will discuss custom event binding using @Output decorator step by step.   
1. Create text box using element property binding in child component. input event is fired when there is any change in text box. $event.target.value fetches the current value of text box entered by user.

<div>

First Number :<input (input)="num1=$event.target.value" /> <br/>

Second Number:<input (input)="num2=$event.target.value" /> <br/>

<br/> <button (click)="addNumber()">Add Number</button>

</div>

2. Find the method created in child component that will be fired when click event is invoked on click of button from above (step-1) code snippet. emit() is the method of EventEmitter class that emits event payload.

addNumber() {

this.addNumEvent.emit(parseInt(this.num1) + parseInt(this.num2));

}

3. In the child component, create an instance of EventEmitter annotated by @Output decorator. This instance will work as custom event name. Here we are using aliasing for custom event name.

@Output('addNumberEvent')

addNumEvent = new EventEmitter<number>();

4. Now we are performing custom event binding. The custom event addNumberEvent will be invoked in parent component when emit() method is invoked from child component. The event payload is accessed by $event object.

<child-two (addNumberEvent) = "printSum($event)" > </child-two>

5. The event addNumberEvent will call printSum() method.

printSum(res) {

this.sum = res;

}

@input example:

<b> {{ctMsg}}</b>

export class ChildOneComponent {

@Input()

ctMsg : string;

}

<child-one

[ctMsg]="cityMsg">

</child-one>

export class ParentComponent {

//Property for child component one

cityMsg = 'Indian City Names';

@output example:

<div>First Number :<input (input)="num1=$event.target.value" /> <br/>

Second Number:<input (input)="num2=$event.target.value" /> <br/>

<br/> <button (click)="addNumber()">Add Number</button></div>

export class ChildTwoComponent {

num1 : '';

num2 : '';

@Output('addNumberEvent')

addNumEvent = new EventEmitter<number>();

addNumber() {

this.addNumEvent.emit(parseInt(this.num1) + parseInt(this.num2));

}

}

<child-two

(addNumberEvent) = "printSum($event)" ></child-two>

<p>Sum: {{sum}}</p>

export class ParentComponent {

sum = '';

printSum(res) {

this.sum = res;

}}

## What is @ViewChild()?

 @ViewChild() provides the instance of another component or directive in a parent component and parent component can access the methods and properties of that component or directive. @ViewChild() can be used for component communication.

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

@Component({

selector: 'app-number',

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

})

export class NumberComponent {

message:string ='';

count:number = 0;

increaseByOne() {

this.count = this.count + 1;

this.message = "Counter: " + this.count;

}

decreaseByOne() {

this.count = this.count - 1;

this.message = "Counter: " + this.count;

}

}

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

import { NumberComponent } from './number.component';

@Component({

selector: 'app-number-parent',

templateUrl: './number-parent.component.html'

})

export class NumberParentComponent {

@ViewChild(NumberComponent)

private numberComponent: NumberComponent;

increase() {

this.numberComponent.increaseByOne();

}

decrease() {

this.numberComponent.decreaseByOne();

}

}

**number-parent.component.html**

<h3>@ViewChild using Component</h3>

Number Example:

<button type="button" (click)="increase()">Increase</button>

<button type="button" (click)="decrease()">Decrease</button>

<br/><br/>

<app-number></app-number>

What is the ng-template?

<ng-template> is an angular element for rendering HTML. It is never displayed directly. It can be displayed using structural directive, ViewContainerRef etc. Suppose we have following code in our HTML template.

<ng-template>

<p>Hello World!</p>

</ng-template>

When the code runs then the code written inside <ng-template> will not be displayed but there will be a comment.

<!---->

<h3>ng-template with ngFor</h3>

<ng-template ngFor let-person [ngForOf]= "allPersons" let-i="index">

<p>{{i + 1}}. {{person.name}} : {{person.age}} </p>

</ng-template>

let declares template input variable. The properties of NgFor such as index, first, last, even, odd can be assigned to a variable using let within the <ng-template>. Find the sample code.

<ng-template ngFor let-item [ngForOf]="items" let-i="index" let-o="odd" let-e="even" let-f="first" let-l="last" [ngForTrackBy]="trackByFn" >

---

</ng-template>

<button type="button" (click)="onToggle2()">Toggle</button>

<div \*ngIf="toggleFlag2; else msgElseBlock" >

<div>Hello World!</div>

</div>

<ng-template #msgElseBlock>

<div>Else Block: Hello World! </div>

</ng-template>

## What is Template Reference Variable?

A template reference variable is a reference to a DOM element or directive within a template. Using template reference variable we access the values of DOM element properties. Template reference variable is declared using **#**

Using **#**  
Find the example.

<input type="text" #myVar>

Here **myVar** will be a template reference variable.

### Template Reference Variable using Input Text Box

Here we will discuss template reference variable using input text box. Template reference variable is a variable using which we can access DOM properties. In our example we are using following DOM properties of input box.   
  
1. placeholder   
2. type   
3. value Now find the code snippet.

<input type="text" #mobile placeholder="Enter Mobile Number">

In the above input text box **#mobile** is a template reference variable. To fetch DOM properties, we do as follows.   
**mobile.placeholder**: It will give **placeholder** of our text box if we have specified.   
**mobile.value**: It will give **value** of our text box.   
**mobile.type**: It will give **type** of input element. In our example type is **text**.

### Template Reference Variable with NgForm

form (ngSubmit)="onSubmitPersonForm(myForm)" #myForm="ngForm">

<input name="name" required [(ngModel)]="person.pname">

<button type="submit" [disabled]="!myForm.form.valid">Submit</button>

</form>

Here we are using template reference variable for ngForm as #myForm="ngForm"

## Dynamic Component Loader?

Generally a component is loaded using component selector in component template that is identified at Angular compile time. Component can also be loaded dynamically at runtime using ComponentFactory, ComponentFactoryResolver and ViewContainerRef. Those components which need to be loaded dynamically should also be configured in entryComponents metadata of @NgModule decorator in module file. To load a dynamic component in a template we need an insert location and to get it we need ViewContainerRef of a decorator or a component.

entryComponents: [

ArticleComponent,

TechnologyComponent

],

### ComponentFactory and ComponentFactoryResolver

ComponentFactory is used to create instance of components. ComponentFactoryResolver resolves a ComponentFactory for a specific component. It is used as follows.

let componentFactory = this.componentFactoryResolver

.resolveComponentFactory(component);

## Interpolation Expression HTML ?

Interpolation is represented using double-curly braces. Using Interpolation we evaluate expressions. The component properties, mathematical calculation etc are executed within interpolation. It is also used for data binding. Interpolation can be used within HTML tag. It also assigns the value to tag attributes.

Interpolation executes expression

Sum of 20 + 30 is {{20 + 30}}

And it prints as

Sum of 20 + 30 is 50

Interpolation accesses values of component properties. Find some examples.

{{title}}

{{primeMinister.name.lname}}

Interpolation can also be used to assign the values to tag attributes.

<img src="{{imageUrl}}">

## Angular lifecycle hooks

## The Order of Execution of Life Cycle Hooks:

When the Component is Created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
6. AfterViewInit
7. AfterViewChecked
8. ngOnDestroy

When the Component with Child Component is created

1. OnChanges
2. OnInit
3. DoCheck
4. AfterContentInit
5. AfterContentChecked
   1. Child Component -> OnChanges
   2. Child Component -> OnInit
   3. Child Component -> DoCheck
   4. Child Component -> AfterContentInit
   5. Child Component -> AfterContentChecked
   6. Child Component -> AfterViewInit
   7. Child Component -> AfterViewChecked
6. AfterViewInit
7. AfterViewChecked

After The Component is Created

1. OnChanges
2. DoCheck
3. AfterContentChecked
4. AfterViewChecked

The OnChanges hook is fired only if there is a input property defined in the component and it changes. Otherwise it will never fire.

### OnInit

OnInit is interface. It has a method ngOnInit(). It is called after data-bound properties of component/directive are initialized. ngOnInit() is called only once. In the lifecycle sequence, ngOnInit() is called just after first ngOnChanges() call. OnInit can be implemented by component, directive, pipe etc. ngOnInit() can be used for following purposes.   
**1.** Perform complex initialization in ngOnInit() and not in constructor.   
**2.** If we need to fetch data then it should be done in ngOnInit() and not in constructor so that we should not worry while initializing component. A constructor should perform only local variable initialization.   
  
For the example a component will implement OnInit as following.

@Component({

---

})

export class CounterComponent implements OnInit {

ngOnInit() {

---

}

---

}

### OnDestroy

OnDestroy interface is a lifecycle hook. It has a method ngOnDestroy(). It is called for cleanup logic when a component, directive, pipe or service is destroyed. ngOnDestroy() is called just before component/directive is about to be destroyed by Angular. It can be used for following purposes.   
1. Stop interval timers.   
2. Unsubscribe Observables.   
3. Detach event handlers.   
4. Free resources that will not be garbage collected automatically.   
5. Unregister all callbacks.   
  
If we don't perform the above task in ngOnDestroy() then memory leaks are possible. For the example a component will use OnDestroy as following.

@Component({

---

})

export class CounterComponent implements OnDestroy {

---

ngOnDestroy() {

---

}

}

## What is data-bound properities?

When you have a component

@Component({

selector: 'my-component'

})

class MyComponent {

@Input() name:string;

ngOnChanges(changes) {

}

ngOnInit() {

}

}

you can use it like

<my-component [name]="somePropInParent"></my-component>

This make name a data-bound property.

When the value of somePropInParent was changed, Angulars change detection updates name and calls ngOnChanges()

After ngOnChanges() was called the first time, ngOnInit() is called once, to indicate that initial bindings ([name]="somePropInParent") were resolved and applied.

**Property Binding:**

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

@Component({

selector: 'my-app',

template: `

<h1 [innerHtml]='fullName'></h1>

`

})

**export** **class** **AppComponent** {

fullName: string = 'Martin Luther King Jr';

}

## OnChanges + SimpleChanges :

OnChanges is an interface and has a method declaration i.e ngOnChanges() . In parent-child component, the child component declares @Input() property to get values from parent component. Whenever parent component changes the value of properties used in child component decorated with @Input() then the method ngOnChanges() created in child component runs automatically. The method ngOnChanges() uses SimpleChanges as an argument that gives new and previous value of input values after changes. In case of input user object data type, ngOnChanges() is called only ,when the reference of object is changed in parent component. If we change only values of properties of an input user object then ngOnChanges() method will not run.

### SimpleChange Class and SimpleChanges Interface

SimpleChange class represents a basic change from a previous to new value. It has following class members. Suppose there is a change in input value, then following property can be used to detect changes.   
**previousValue**: Keeps previous value of input property.   
**currentValue**: Keeps current value of input property.   
**isFirstChange()**: Boolean value that tells whether the new value is the first value assigned.   
  
SimpleChanges is the interface that represents the changes object for all input property. SimpleChanges has the key as input property names and values are the instances of SimpleChange class.

### OnChanges:

 OnChanges is an interface that has a method declaration as follows.

ngOnChanges(changes: SimpleChanges)

It has the argument as SimpleChanges that is used to get new and previous values of input property.

Convert a JavaScript object into a string with JSON.stringify().

### OnChanges with @Input() Primitive Type

Suppose we have following primitive type decorated with @Input() in a child component.

@Input() message: string;

Now whenever parent component changes value in any of its property that has been used in child component, then in the child component the ngOnChanges() method runs. It works for any primitive data type such as string, number etc.

### OnChanges with @Input() User Object Type

Suppose we have user object data type decorated with @Input() in child component.

@Input() employee: Employee;

In child component when we have a user object as an @Input() data-bound property, then ngOnChanges() method is called only when the reference of the object is changed by parent component. Reference of the object can be changed by assigning new object to it. It means if we change value of property of object in parent component, then ngOnChanges() method will not be called in child component because reference is not changed.

onFormSubmit(empForm: NgForm) {

let name = empForm.controls['name'].value;

let age = empForm.controls['age'].value;

this.emp = new Employee(name, age); // Reference of the object can be changed by assigning new object to it

}

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

export class EmployeeComponent implements OnChanges {

@Input() employee: Employee;

@Input() message: string;

ngOnChanges(changes: SimpleChanges) {

for (let propName in changes) {

let change = changes[propName];

let curVal = JSON.stringify(change.currentValue);

let prevVal = JSON.stringify(change.previousValue);

let changeLog = `${propName}: currentValue = ${curVal}, previousValue = ${prevVal}`;

if (propName === 'message') {

this.allMsgChangeLogs.push(changeLog);

} else if (propName === 'employee') {

this.allEmployeeChangeLogs.push(changeLog);

}

}

* When component is initialized, Angular invokes ngOnInit
* When a component’s input properties change, Angular invokes ngOnChanges
* When a component is destroyed, Angular invokes ngOnDestroy

## ngOnInit

This hook is called when the component is created for the first time. This hook is called after the constructor and first ngOnChanges hook.

This is a perfect place where you want to add any initialisation logic for your component.

Note that ngOnChanges hook is fired before ngOnInit. Which means all the input properties are available to use when the ngOnInit is hook is called

This hook is fired only once

This hook is fired before any of the child directive properties are initialized.

## ngDoCheck

The Angular ngOnChanges hook does not detect all the changes made to the input properties. It Only detects when the Input Property is a primitive type or  reference to the Input property changes

This hook is provided so as to Implement custom change detection, whenever Angular fails to detect the changes made to Input properties.

## When ngDoCheck is called?

This hook is called after every change detection cycle no matter where the change has occurred.

## How to watch for changes an array/object on Angular ?

There are two types of differs, that angular provides

1. key-value differs
2. iterable differs

### key-value differs

The KeyValueDiffers service is a differ that tracks changes made to an object over time and also expose an API to react to these changes.

Key-value differs should be used for dictionary-like structures, and it works at the key level. This differ will identify changes when a new key is added, when a key removed and when the value of a key changed.

### Iterable differs

Iterable differs service is used when we have a list-like structure and we’re only interested in  
knowing things that were added or removed from that list.

It will detect if the elements are added/removed from the array. This will not detect if the changes are done to the elements of array.

To do that, you need to create a separate key value differ for the each element.

## Property binding:

 Property binding is a one-way data binding from data source to view target. Property binding is performed with component property, HTML element and angular directives. Component property binding is used for communication between parent and child component because using this binding we can send property values from parent to child component. In element property binding the DOM property of HTML element can be assigned with a value of component property. In directive property binding we can assign component property values to angular 2 directives.

**[(target)] = "source"**is two-way binding where **[]** is to set value from source to target and **()** is to set value from target to source  
Component property binding is performed as below.

<my-msg prefixMsg= "Website name is " [siteName] = "website.name"> </my-msg>

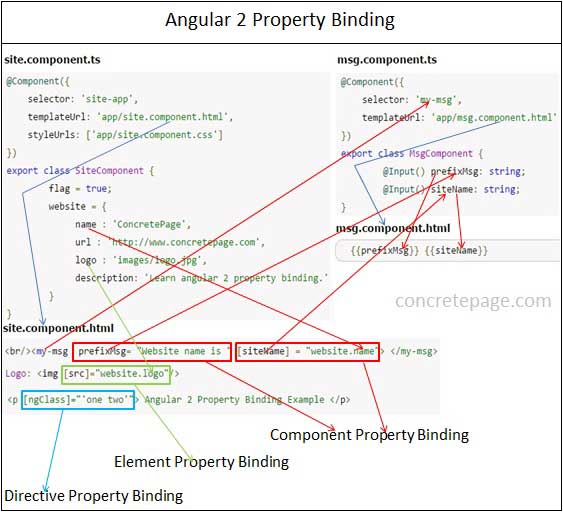
Element property binding is performed as below.

<a [href]="website.url" [textContent]="website.name"> </a>

Directive property binding is performed as below.

<p [ngClass]="'one two'"> Angular 2 Property Binding Example </p>

Angular framework ensures content security while displaying data. If we try to display data with <script> tag then it will not allow. Angular filters the data before display. Angular calls such type of coding not only a HTML but HTML Plus because it is more powerful. Now find the complete example of angular 2 property binding step by step using TypeScript.



.one {

color: green;

}

.two {

font-size: 20px;

}

Property binding target will use the below syntax.   
1. Bracket []   
2. bind- prefix   
3. Interpolation {{expression}}   
We can choose any of the above syntax for property binding that fits to our readability point of view.

Logo: <img [src]="website.logo"/>

<br/>Logo: <img bind-src="website.logo"/>

<br/>Logo: <img src="{{website.logo}}"/>

## Event Binding:

Event binding is a data binding from an element to a component. In the event binding, target will be an event name. Target event is enclosed within parenthesis **( )** . Target event is written to the left side of equal sign. Target events can be click, change, mouseover, mouseout, keydown, keyup etc. we can create custom events.

To access current changes by user, we use event object $event. It will be DOM event object if target event is a native DOM element event. It has properties as target and target.value.

Event binding using parenthesis **( )** is achieved as follows.

<button (click)="isValid=true">True</button>

And event binding using **on-** keyword is achieved as follows.

<button on-click="isValid=true">True</button>

Call component method on event binding.

<input (change) = "changeText($event.target.value)">

### Input Event Binding

input event is a DOM event which is fired synchronously when the value of <input> or <textarea> element is changed. Here we will bind the component property with input event. We will use event object $event to get the current value entered by user. For the example find a component property.

msg1 = 'Hello World';

For the initial value of our <input> element we are using property binding and assigning component property value. For event binding we need to enclose event name by parenthesis as **(input)** . Now find input event binding using **( )**.

<input [value]="msg1" (input)="msg1=$event.target.value">

We can also use **on-** keyword as follows.

<input [value]="msg1" on-input="msg1=$event.target.value">

Achieve same goal by calling a component method using event binding. Create a method in component.

setMsg(data:string) {

this.msg1 = data;

}

Use input event to call the method.

<input [value]="msg1" (input)="setMsg($event.target.value)">

Here we are passing the current input data entered by user. If we print the message as

{{msg1}}

Then we will observe that whenever we write any text in <input> element, at the same time value of msg1 will also be changed. Here $event gives the current value entered by user. We access it by using   
$event.target.value  
In our code we are assigning its value to msg1 as

msg1=$event.target.value

### Change Event Binding

Here we will discuss change event binding using <input> element. On value change we will call a method which we have defined in our component as follows.

changeText(mytext:string) {

this.msg3 = mytext;

}

Now find the <input> element which is using change event binding.   
Using parenthesis **( )**.

<input (change) = "changeText($event.target.value)">

Using **on-** keyword.

<input on-change = "changeText($event.target.value)">

For the demo we are printing data using innerHTML in <p> element as follows.

<p [innerHTML] = "msg3"> </p>

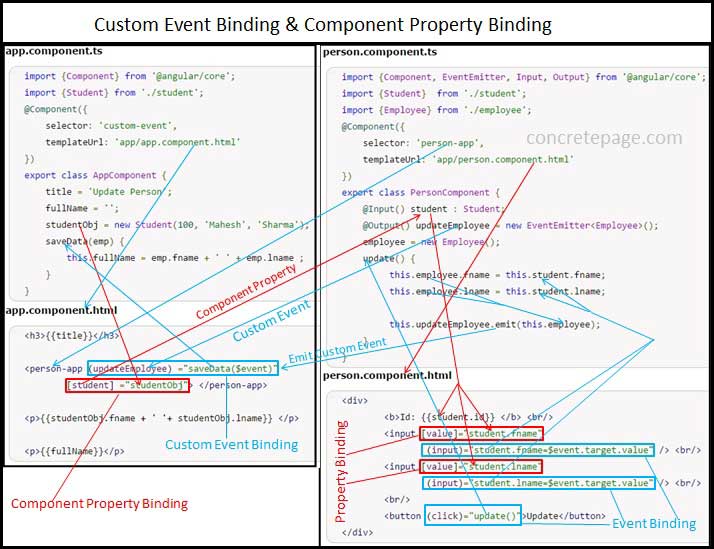
Now use interpolation.

<p> {{msg3}} </p>

When user changes text value then after value change completion, changeText() method will be called and current user input value will be passed to the method that will be assigned to a component property msg3. And hence we will observe value change within <p> element for every value change in <input> element.

## Custom Event Binding+ EventEmitter

Angular framework provides event binding using in-built event as well as custom event. Event binding is achieved using parenthesis **().**When using parenthesis **()**, we need to write our event name inside it. Custom events are the EventEmitter instances. To create a custom event we need to create an instance of EventEmitter annotated by @Output(). Then this instance calls emit() method to emit payload which can be received by event object $event.  To send data from parent to child component, property binding will be used and to get data in parent component from child component, custom event binding will be used.



### EventEmitter

EventEmitter is a class in angular framework. It has emit() method that emits custom events. We can use EventEmitter in custom event binding. To achieve it first we need to import it in our component file as given below.

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

And then initialize it using @Output decorator as follows.

@Output() updateEmployee = new EventEmitter<Employee>();

Here Employee is our TypeScript class. @Output() defines an output variable. updateEmployee will be a **custom event** name. Using emit() method of EventEmitter class we emits Employee object to parent component in custom event binding as follows.

this.updateEmployee.emit(this.employee);

The object emitted by emit() method can be accessed using event object $event.

### Create Custom Event

Suppose we want to create a custom event named as updateEmployee. So first of all we need to instantiate an instance of EventEmitter in the variable updateEmployee as below.

@Output() updateEmployee = new EventEmitter<Employee>();

We need to annotate it with @Output that declares it as output variable. Custom event binding is used in parent child component communication. Here we will send data from child component to parent component. To send data we need to use emit() method as below.

this.updateEmployee.emit(this.employee);

Now we are ready to use variable updateEmployee as custom event. Event binding can be done using parenthesis **()**or **on-** keyword. Find event binding using **()**.here person-app in parent html.

<person-app (updateEmployee) ="saveData($event)"

[student] ="studentObj"> </person-app>

Now we will use **on-** keyword for updateEmployee event binding.

<person-app on-updateEmployee ="saveData($event)"

[student] ="studentObj"> </person-app>

In the above code snippet [student] ="studentObj" is working as a property binding to send data from parent component to child component.

## Two-Way Data Binding + NgModel :

Using two-way binding we can display a data property as well as update that property when user makes changes. We can achieve it in component element and HTML element both. Two-way binding uses the syntax as **[()].**

Two-way binding uses the syntax of property binding and event binding together. Property binding uses the syntax as bracket **[]** or **bind-** and event binding uses the syntax as parenthesis **()** or **on-** and these bindings are considered as one-way binding. Two-way binding works in both direction setting the value and fetching the value.  In component property binding and custom event binding we create our own property name and event name as **target** in binding. So we can follow the naming pattern. But in HTML element there are in-built names as **target** in binding such as value in property binding and input in event binding. So here the role of NgModel directive comes into the picture to work as bridge that enables two-way binding to HTML elements. It provides the required name pattern of **target** as ngModel in property binding and ngModelChange in event binding.

1. **[(target)] = "source"**is two-way binding where **[]** is to set value from source to target and **()** is to set value from target to source. For example find the code snippet.

<msg-app [(cdMsg)] ="msg"> </msg-app>

**2.[(ngModel)] = "source"** is a two-way binding using NgModel directive. We will use **[(ngModel)]** in HTML element where we set a specific element property and listen for an element change event . We will use two-way binding with NgModel in text box and select box in our example. **[(ngModel)]** can set only data-bound property. To use NgModel we need to import FormsModule and add it to imports attribute of @NgModule in our module file. Now for the example of two-way data binding using NgModel find the code snippet.

<input [(ngModel)] ="myMsg"/>

## What is the directive?

The Angular directive helps us to manipulate the DOM. You can change the appearance, behaviour of a DOM element using the Directives. They help you to extend HTMLThere are three kinds of directives in Angular:

1. Component Directive
2. Structural directives
3. Attribute directives



Component directive is used to create HTML template. Attribute directive changes the appearance and behavior of DOM element. Structural directive changes the DOM layout by adding and removing DOM elements.

Angular also provides built-in directives. The built-in attribute directives are NgStyle, NgClass etc. These directives change the appearance and behavior of HTML elements. The built-in structural directives are NgFor and NgIf etc. These directives can add and remove HTML elements from the DOM layout.

Custom directives are created using following syntax.

@Directive({

selector: '[cpDir]'

})

export class CPDirective {

}

The directive name is cpDir here. It should be enclosed within bracket **[]**. We can keep directive name as we want but it should be started with your company name or any other keyword but not with Angular keyword such as ng. To behave our directive like attribute directive, we can use ElementRef to change appearance. To listen event we can use @HostListener() decorator. To behave our directive like structural directive, we can use TemplateRef and ViewContainerRef.

## Custom Attribute Directives

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

@Directive({

selector: '[cpDefaultTheme]'

})

export class CPDefaultThemeDirective implements AfterViewInit {

constructor(private elRef: ElementRef) {

}

ngAfterViewInit(): void {

this.elRef.nativeElement.style.color = 'blue';

this.elRef.nativeElement.style.fontSize = '20px';

}

}

Use ElementRef class to access DOM to change host element appearance.   
Use @Input() decorator to accept user input in our custom directive.   
Use @HostListener() decorator to listen events in custom attribute directive

AfterViewInit is the lifecycle hook that is called after a component view has been fully initialized. To use AfterViewInit, our class will implement it and override its method ngAfterViewInit().

<p cpDefaultTheme> cpDefaultTheme Directive Demo</p>

The text of the <p> element will be blue with font size 20px.

## Custom Structural Directives

We will create a structural directive that will add a layout in DOM for a true condition otherwise it will delete it from DOM.

import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({

selector: '[cpIf]'

})

export class CpIfDirective {

constructor( private templateRef: TemplateRef<any>,

private viewContainer: ViewContainerRef) { }

@Input() set cpIf(condition: boolean) {

if (condition) {

this.viewContainer.createEmbeddedView(this.templateRef);

} else {

this.viewContainer.clear();

}

}

}

Custom directive selector name: cpIf   
Example to use our custom directive:

<div \*cpIf="showCpIf">

<b>Hello World!</b>

</div>

<ng-template [cpIf]="!showCpIf">

<div>

<b>Not Available</b>

</div>

</ng-template>

When showCpIf is true then **Hello World!** message will be shown otherwise the message will be **Not Available** .

**a.** Using directive with \* prefix in host element.

<div \*cpIf="showCpIf">

<b>Hello cpIf Directive.</b>

</div>

**b.** Using directive with ng-template.

<ng-template [cpIf]="showCpIf">

<div>

<b>Hello cpIf Directive.</b>

</div>

</ng-template>

## TemplateRef

It represents an embedded template that can be used to instantiate embedded views.

<ng-template #msg>

   Welcome to you.<br/>

   Happy learning!

</ng-template>

export class CpMsgComponent implements  AfterViewInit {

    @ViewChild('msg')

    private msgTempRef : TemplateRef<any>

   ----

}

## ViewContainerRef

It represents a container where one or more views can be attached. We can use its methods such as createEmbeddedView() and createComponent()

import { Directive, TemplateRef, ViewContainerRef, Input } from '@angular/core';

@Directive({

     selector: '[cpIf]'

})

export class CpIfDirective {

constructor( private templateRef: TemplateRef<any>,

             private viewContainer: ViewContainerRef) { }

@Input() set cpIf(condition: boolean) {

   if (condition) {

  this.viewContainer.createEmbeddedView(this.templateRef);

   } else {

  this.viewContainer.clear();

  }

}

}

## NgIf

 NgIf is used with HTML elements as well as component elements.   
NgIf is used with HTML elements as follows.

<div \*ngIf="isValid"> Data is valid. </div>

NgIf is used with component elements as follows.

<my-msg \*ngIf="emp1" [pname] = "emp1.name"> </my-msg>

isValid = true;

emp1 = new Employee(100, 'Nilesh');

b>NgIf with HTML Elements </b><br/>

<p \*ngIf="isValid">

Data is valid.

</p>

<p \*ngIf="!isValid">

Data is not valid.

</p>

<br/><b>NgIf with Component Elements</b><br/>

<my-msg \*ngIf="emp1" [pname] = "emp1.name"> </my-msg>

<div \*ngFor="let id of ids">

Id is {{id}}

<div \*ngIf="id%2 == 0">

<div [ngClass]="'one'">Even Number</div>

</div>

<div \*ngIf="id%2 == 1">

<div [ngClass]="'two'">Odd Number</div>

</div>

</div>

<ng-template> is used by structural directive such as NgIf.

<h3>Using NgIf with Then and Else</h3>

<b>Example-1:</b><br/><br/>

<div>

<input type="radio" name="rad3" (click)= "changeValue(true)"> True

<input type="radio" name="rad3" (click)= "changeValue(false)"> False

</div>

<div \*ngIf="isValid; then thenBlock; else elseBlock"> </div>

<ng-template #thenBlock>

<div> <b>Data is valid.</b> </div>

</ng-template>

<ng-template #elseBlock>

<div> <b>Data is invalid.</b> </div>

</ng-template>

## Ngswitch

NgSwitch is used as property binding such as [ngSwitch] with bracket **[ ]**

<ul [ngSwitch]="person">

<li \*ngSwitchCase="'Mohan'">Hello Mohan</li>

<li \*ngSwitchCase="'Sohan'">Hello Sohan</li>

<li \*ngSwitchCase="'Vijay'">Hello Vijay</li>

<li \*ngSwitchDefault>Bye Bye</li>

</ul>

<div \*ngFor="let id of ids">

Id is {{id}}

<div ngSwitch="{{id%2}}">

<div \*ngSwitchCase="'0'" [ngClass]="'one'">I am Even.</div>

<div \*ngSwitchCase="'1'" [ngClass]="'two'">I am Odd.</div>

<div \*ngSwitchDefault>Nothing Found.</div>

</div>

</div>

**ngFor**

It is not a type-safe

ngFor directive instantiates a template once per item from iterate

It will be works like as collections

**ngForOf**

It is a type Safe

It will be works like as generics

**Note**The ngFor and ngForOf are actually the selectors of the NgForOf directive and it is not two distinct things

### NgForOf with HTML Elements

NgForOf directive is used with HTML elements as following.

<li \*ngFor="let item of items; index as i; even as isEven; odd as isOdd; first as isFirst; last as isLast; trackBy: trackByFn">

------

</li>

### NgForOf with <ng-template>

NgForOf directive is used with <ng-template> as following.

<ng-template ngFor let-item [ngForOf]="items" let-i="index" let-isEven="even" let-isOdd="odd" let-isFirst="first" let-isLast="last" [ngForTrackBy]="trackByFn">

<li> ------ </li>

</ng-template>

## NgClass

 It is used to add and remove CSS classes on an HTML element. We can bind several CSS classes to NgClass simultaneously that can be added or removed. There are different ways to bind CSS classes to NgClass that are using string, array and object. CSS classes are assigned to NgClass as property binding using bracket i.e [ngClass] as attribute of any HTML element .

<p [ngClass]="'one'">

Using NgClass with String.

</p>

In case we want to add more than one CSS classes then add it separated by space.

<p [ngClass]="'one two'">

Using NgClass with String.

</p>

<p [ngStyle]="{'color': 'purple',

               'font-size': '20px',

               'font-weight': 'bold'}">

     Multiple styles

</p>

### Using object from Controller

class StyleClass {

   'color': string= 'blue';

   'font-size.px': number= 20;

   'font-weight': string= 'bold';

}

And in controller initialize the class

styleClass: StyleClass = new StyleClass();

Then you can refer it in your template as shown below

<div [ngStyle]="styleClass">size & Color</div>

## Angular Pipes

 transform data to display in our template.  where we may have to change the appearance of the data before presenting it the user.

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

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>`

})

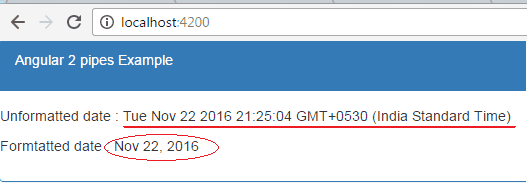
export class AppComponent

{

    title: string = 'pipe Example' ;

    toDate: Date = new Date();

}



### Passing arguments

We can also pass optional arguments to the pipe. The arguments are added to the pipe using a **colon (:)** sign followed by the value of the argument. If there are multiple arguments separate each of them with the **colon (:)**.

{{toDate | date:'medium'}}

## 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 Date pipe is then passed to the **uppercase pipe**

toDate | date | uppercase

Some of the important pipes are **Date Pipe**, **Uppercase Pipe**, **LowercasePipe**, **Number Pipe/Decimal Pipe**, **Currency Pipe**, and **Percent Pipe** etc

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

We need to import the Pipe & PipeTransform libraries from Angular 2. These libraries are part of the Angular 2 Core

@pipe({

    name: 'tempConverter'

})

export class TempConverterPipe implements PipeTransform {

    transform(value: number, unit: string) {

return;

    }

}

Every pipe we built must implement the **PipeTransform interface.It** has one method known as the transform. This interface takes the value being piped as the first argument. It takes the variable number of optional arguments of any type. It returns the final transformed data.

