**Data binding:**

1. String interpolation → {{}}

2. property binding → [xxx]=”xx”

3. event binding → (xxx)=”xx”

4. two way binding → [(xxx)]=”xx”

**Directives:**

Built-in directives:

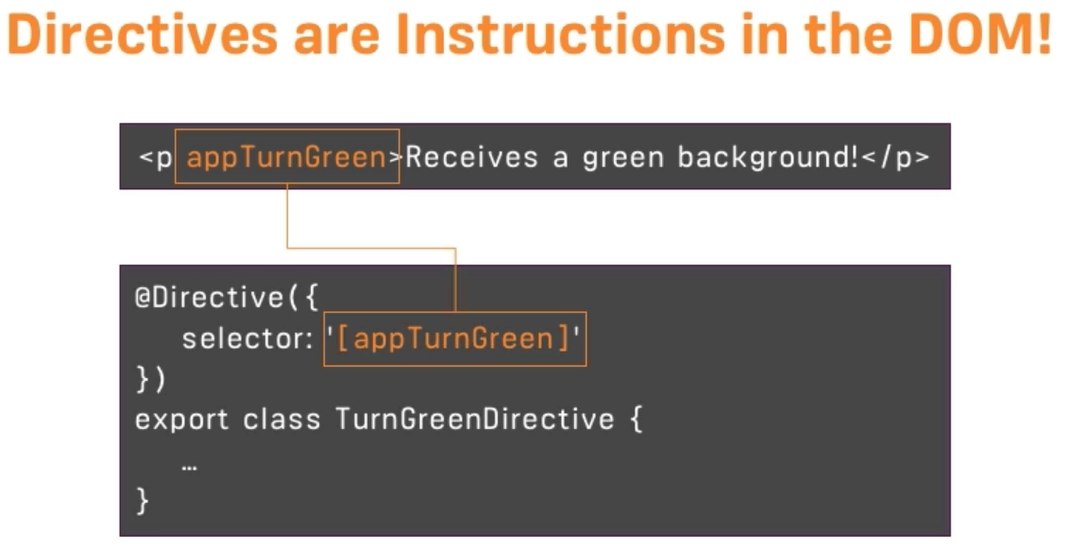
1. \*ngIf =”xx”*; else xx*

2. ngStyle // dynamically assign style eg, : [ngStyle]=”{backgroundColor: xx}”

3. ngClass // dynamically assign css classes eg: [ngClass]=”{css-class: xxx === xx}”

4. \*ngFor=”let xx of xxx*;* *let i = index*”

Custom directives :



**Components and databinding**

Binding a custom properties:

@Input() xx // decorator

**View Encapsulation:**

Defines the properties applied on components should be specific to component or not.

ShadowDom and Emulated are same, but native uses ShadowDom Technology(not supported by all browsers)

@Component({

---

---

encapsulation: ViewEncapsulation.Emulated //None, ShadowDom

}) |

(Default)

**ViewChild**

getting accesss to template and dom

@ViewChild(‘localRefName/component’,{static: true}) xx: ElementRef;

xx.nativeElement.value

**\*ngFor :-**

use to display list of items, manipulates dom by adding attributes dynamically.

Eg.

<ul>

<li \*ngFor=”let xx of xxs”> {{xx}}</li>

</ul>

**Services:** // ng g s serviceName

Logic should be implemented in services & not in component

We do not have any decorator for services.

We use here @Injectable(), if our service is dependent on another class. We donot use this in components because @Component internally uses @Injectable.

In angular.module.ts, we need to register service if we are using dependency injection.

Like:

providers: [XxxService, ...]

**Property Binding**

[xxx]=”xx”

works from component to dom, not the other way(one way binding).

**Attribute Binding**

sometimes when binding, the html and dom are not mapped(like in below eg.) hence we use this

<tr>

<td [attr.colspan]=”xx”></td>

</tr>

**To add bootstrap in application**

1. npm install bootstrap --save

|

adds bootstrap as dependency in package.json

2. styles.css → import “~bootstrap/dist/css/bootstrap.css”

**Class Binding**

[class.active]=”xx” //property binding

**Style Binding**

[style.background]=”isActive ? ‘red’:’blue’ ” //property binding

**Event Binding**

(click)=”onSave($event)”

….

...

onSave(){ console.log(“xx”, $event)}

To stop event bubbling i.e. it will not trigger/propogate to another event use:

$event.stopPropagation()

**Event Filtring**

(keyup.enter)=”onKeyUp()”

…

onKeyUp(){console.log(“Enter was pressed”)}

**Template variable**

<input (keyup.enter)=”onKeyUp()”/>

…

onKeyUp(){console.log($event.target.value)}

*can be written as:*

<input #email (keyup.enter)=”onKeyUp(email.value)”/>

…

onKeyUp(email){console.log(email)}

**Two way binding:**

1. app.module.ts → import { FormsModule } from ‘@angular/forms’

imports: [FormsModule]

2. <input [(ngModel)] =”email” (keyup.emter)=”onKeyUp()”/>

export class xxx{

email = “xx”

onKeyUp(){ console.log(this.email) }

}

**Pipes**

Used to format date, built in pipes → Uppercase, Lowercase, Decimal(number ), Currency, Percent

{{Course.xx | uppercase | lowercase }}

{{Course.xx | number }}

{{Course.xx | number: ‘2.1(min)-1(max)’ }}

|

Describes the number of digits before and after decimal

{{Course.xx | currency:’IND’: true : ‘3.2-2’ }}

|

Display currency symbol

{{Course.xx | date:’shortDate’ }}

**Custom Pipes**

1. create file xx.pipe.ts

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

@Pipe({

name:’summary’

})

export class SummaryPipe implements PipeTransform {

transform(value: any, args?:any){

if(!value) return null

let yyy = (args) ? args : y

return value.someoperation(yyy);

}

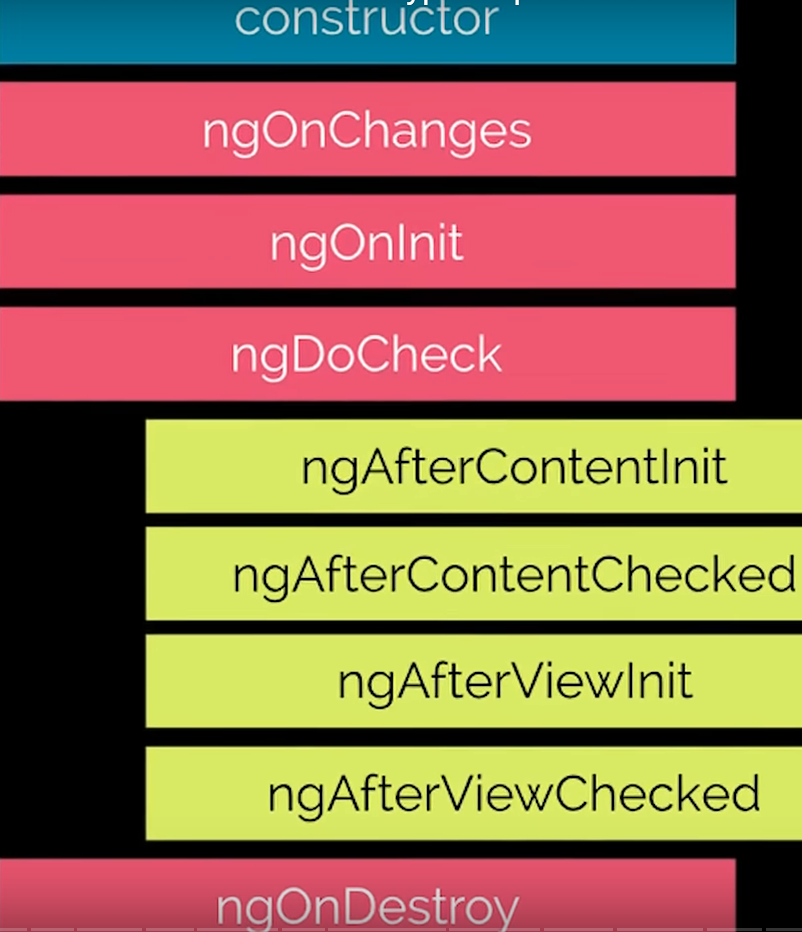
}

2. app.module.ts →

declarations: [xxPipe]

3. in Html → {{xxx | xx: yy}}

**Angular Life cycle Hooks order:**

****

Four ways to communicate between angular components:

1. Binding(@Input & @Output)

2. Reference(@ViewChild & @ContentChild)

3. Provider (service)

4. Template Outlet

**Observables**

→ Comes under rxjs package,

→ There are observables which keeps on emitting even though we are not intrested.

Eg:

import {interval} from ‘rxjs’;

---

ngOnInit(){

interval(1000).subscribe(count =>{console.log(count)})

}

To avoid this memory leak we should unsubscribe the observables in which we are not intrested.

Eg:

import {interval, Subscription} from ‘rxjs’;

---

private obsSubscription: Subscription

ngOnInit(){

this.obsSubscription = interval(1000).subscribe(count =>{console.log(count)})

}

ngOnDestroy(){

this.obsSubscription.unsubscribe();;

}

→ We do not do this for the observables (like params) provided by angular as angular takes care of it.

**Building a custom observable:**

Above observable can be built like:

import {interval, Subscription, *Observable*} from ‘rxjs’;

---

private obsSubscription: Subscription

ngOnInit(){

const customIntervalObserval = Observable.create(observer => {

let count =0

setInterval(() => {

observer.next(count)

count++

}1000)

})

this.obsSubscription = customIntervalObserval.subscribe(count =>{console.log(count)})

}

ngOnDestroy(){

this.obsSubscription.unsubscribe();;

}

**Handling & completing Observables:**

Handling means hanndling the errors/exceptions

Completing means completing the observable process. Note: not all observable ment to be completed

ngOnInit(){

const customIntervalObserval = Observable.create(observer => {

let count =0

setInterval(() => {

observer.next(count)

if(count === 2){observer.complete()}

if(count > 3) {observer.error(new Error(‘count is greater than 3’))}

count++

}1000)

})

*this.obsSubscription = customIntervalObserval.subscribe(*

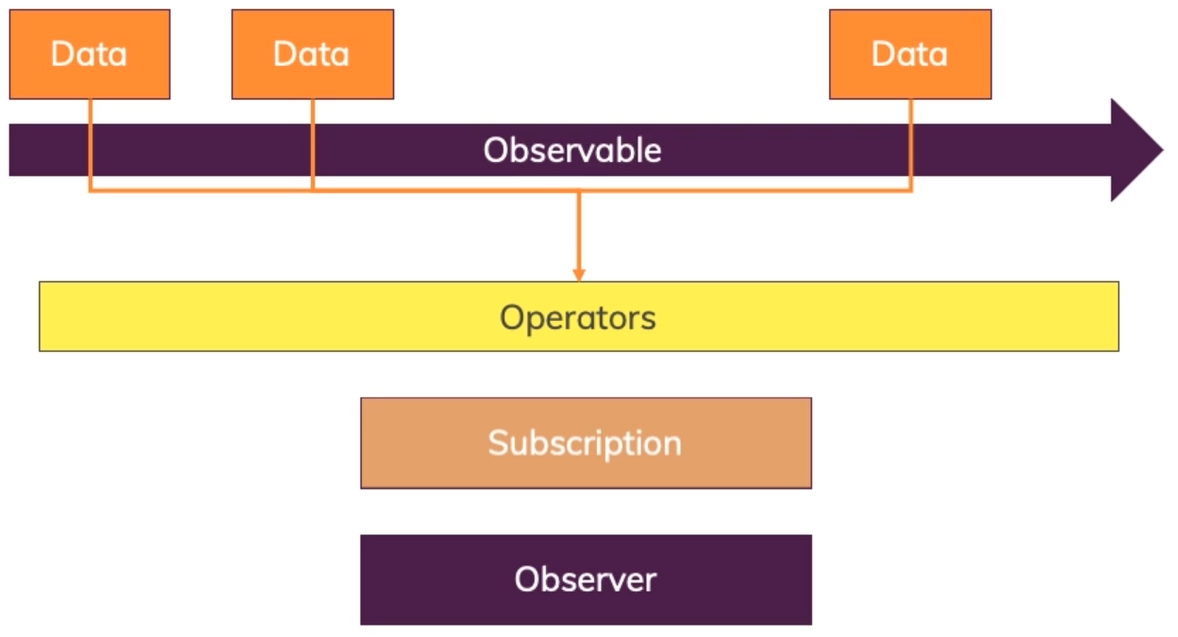
*count =>{console.log(count)},*

error =>{console.log(error)},

() => {console.log(‘completed – here we can do cleanup If needed’)}

*)*

***Operators:***

******

*import {filter, map} from ‘rxjs/operators’*

*this.obsSubscription = customIntervalObserval*

*.pipe(*

*filter(count => {return count>0}), // returns true or false*

*map((count: number) => {return ‘Round: ’+count})*

*)*

*.subscribe(*

*count =>{console.log(count)},*

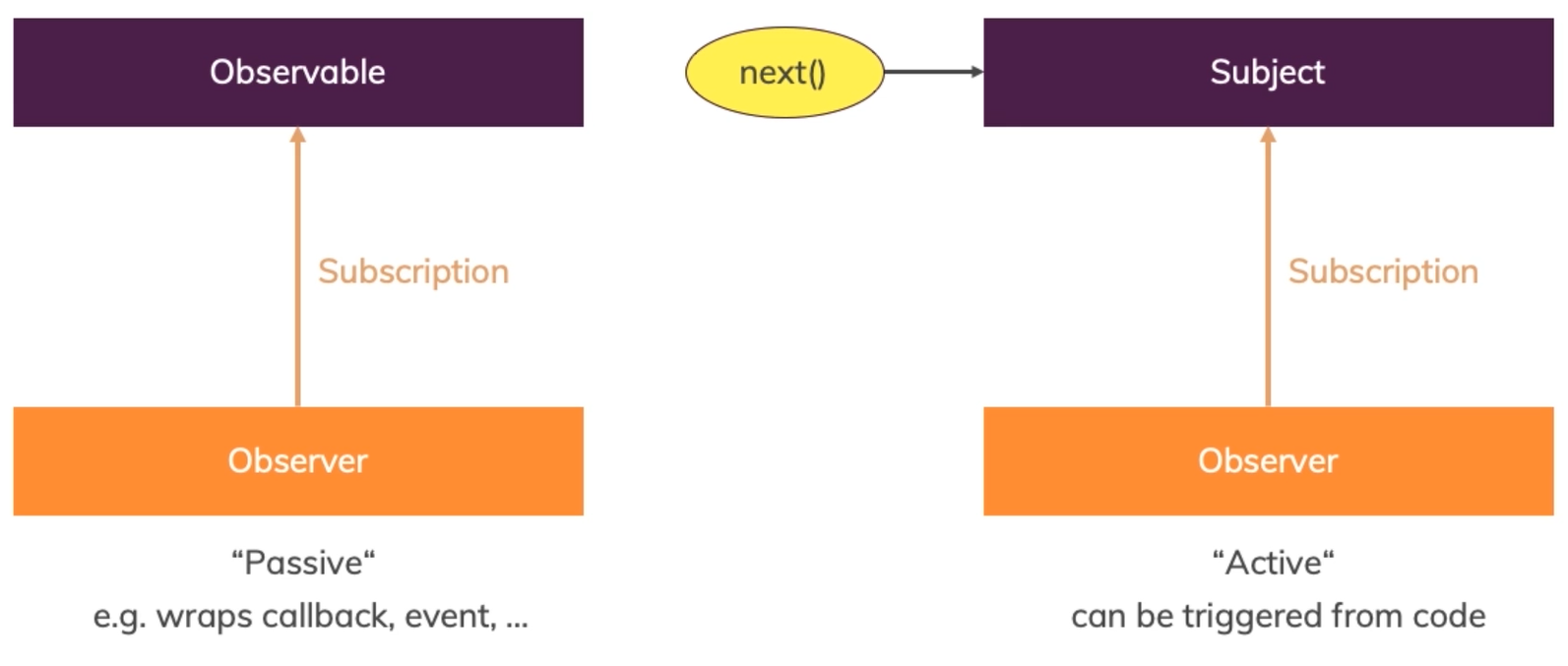
error =>{console.log(error)},

() => {console.log(‘completed – here we can do cleanup If needed’)}

*)*

***Subject***

***(a better event emitter)***

******

*import {Subject} from ‘rxjs’*

*Subjects are the replacement for event emitters i.e. instead of new EventEmitter<T>() use new Subject<T>() and next() instead of emit()*

*Do not use subject when using @Output()*

*use subject to communicate across components through services where we are subscribe to event*

*Like observable we need to destroy/ unsubscribe to subject in ngOnDestroy.*