Important Points

[Link 2](#_Toc525829731)

[Component 2](#_Toc525829732)

[Module 2](#_Toc525829733)

[Binding 2](#_Toc525829734)

[Data Binding 3](#_Toc525829735)

[Event Binding 4](#_Toc525829736)

[Directives / @HostListener and @HostBinding 4](#_Toc525829737)

[Pipes 5](#_Toc525829738)

[Routing 5](#_Toc525829739)

[HTTP 5](#_Toc525829740)

[Services 6](#_Toc525829741)

[Sharing Data between Angular Components 6](#_Toc525829742)

[Constructor vs ngoninit 6](#_Toc525829743)

[Angular Life Cycle Hook 6](#_Toc525829744)

[Memory Leak 7](#_Toc525829745)

[Lazy Loading 7](#_Toc525829746)

[Design Pattern 7](#_Toc525829747)

[Observables 7](#_Toc525829748)

[RxJS 8](#_Toc525829749)

[Reactive Programming 8](#_Toc525829750)

[Functional Programming 8](#_Toc525829751)

[Arrow Function 8](#_Toc525829752)

[Map 9](#_Toc525829753)

[Filter 9](#_Toc525829754)

[Mutable and Immutable 9](#_Toc525829755)

[Imperative vs Declarative 9](#_Toc525829756)

[Closure 9](#_Toc525829757)

[Interface 10](#_Toc525829758)

<https://medium.com/javascript-scene/master-the-javascript-interview-what-is-functional-programming-7f218c68b3a0>

# Link

<https://www.tutorialspoint.com/angular4/index.htm>

# Component

Components are basic building block in an Angular application. Components are defined using the @Component decorator. A component has a selector, template, style and other properties, using which it specifies the metadata required to process the component.

@Component({

selector: 'app-new-cmp',

templateUrl: './new-cmp.component.html',

styleUrls: ['./new-cmp.component.css']

})

# Module

Module in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.

To define module, we can use the NgModule. When you create a new project the ngmodule is created in the app.module.ts file by default.

@NgModule({

declarations: [

AppComponent

],

imports: [

BrowserModule

],

providers: [],

bootstrap: [AppComponent]

})

# Binding

[] – Property Binding / One Way Data Binding

() – Event Binding

[()] – Two way Data Binding

myMsg = 'Hello World!';

Use ngModel for two-way binding

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

If we want to use NgModel as property binding and event binding separately then we need to use ngModel and ngModelChange as follows.

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

# Data Binding

We use curly braces for data binding - {{}}; this process is called interpolation.

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

title = 'Angular 4 Project!';

//array of months.

months = ["January", "February", "March", "April",

"May", "June", "July", "August", "September",

"October", "November", "December"];

isavailable = false;

}

<div style="text-align:center">

<h1>

Welcome to {{title}}.

</h1>

</div>

<div> Months :

<select>

<option \*ngFor="let i of months">{{i}}</option>

</select>

</div>

<br/>

<div>

<span \*ngIf="isavailable; else condition1">Condition is valid.</span>

<ng-template #condition1>Condition is invalid</ng-template>

</div>

# Event Binding

<button (click) = "myClickFunction($event)">Click Me</button>

myClickFunction(event) {

alert("Button is clicked");

console.log(event);

}

# Directives / @HostListener and @HostBinding

Link: <https://dzone.com/articles/what-are-hostbinding-and-hostlistener-in-angular>

There are three types of directives in Angular:

1. Component
2. Attribute Directive
3. Structural Directive

The basic difference between a component and a directive is that a component has a template, whereas an attribute or structural directive does not have a template.

**Components**

Directives with a template.

**Structural Directives**

Change the DOM layout by adding and removing DOM elements. A structure directive basically deals with manipulating the dom elements. Structural directives have a \* sign before the directive. For example, \*ngIf and \*ngFor.

**Attribute Directives**

Change the appearance or behaviour of an element, component, or another directive. Attribute directives deal with changing the look and behaviour of the dom element.

ng g directive nameofthedirective

e.g

ng g directive changeText

# Pipes

Link: <https://www.tutorialspoint.com/angular4/angular4_pipes.htm>

The | character is used to transform data.

<b>{{title | uppercase}}</b><br/>

<b>{{title | lowercase}}</b>

# Routing

Link: <https://code.tutsplus.com/tutorials/beginners-guide-to-angular-4-routing--cms-29676>

When a user enters a web application or website, routing is their means of navigating throughout the application. To change from one view to another, the user clicks on the available links on a page.

# HTTP

Link: <https://code.tutsplus.com/tutorials/beginners-guide-to-angular-4-http--cms-29677>

Angular HttpClient provides a simplified API calls from the Angular application.

# Services

Link: <https://code.tutsplus.com/tutorials/beginners-guide-to-angular-4-services--cms-29675>

In an Angular application, components deal with presenting the data to the view. Fetching data for the components to display is handled by Angular services.

# Sharing Data between Angular Components

Link: <https://angularfirebase.com/lessons/sharing-data-between-angular-components-four-methods/>

@Input() – Parent to child

@ViewChild() – Child to parent

@Output() and EventEmitter – Child to parent

Observable() – Using service

# Constructor vs ngoninit

|  |  |
| --- | --- |
| constructor | ngoninit |
| The Constructor is part of ES6. It is default method of class. | ngOnInit is part of Angular life cycle hook |
| constructor will be called first | ngOninit will be called later after constructor method  Constructor()->ngOnChanges()->ngOnInit() |
| Constructor should be used for dependency injection  constructor(private http: Http, private customService: CustomService) {} |  |
| @Input() properties are undefined inside the constructor | @Input() properties are available inside ngOnInit |

# Angular Life Cycle Hook

Link: <https://angular.io/guide/lifecycle-hooks>

A component has a lifecycle managed by Angular.

|  |  |
| --- | --- |
| Hook | Description |
| ngOnChanges() | Called when Angular sets/resets data-bound input properties. The method receives a SimpleChanges object of current and previous property values. |
| ngOnInit() | Initialize the directive/component after Angular first displays the data-bound properties and sets the directive/component's input properties. |
| ngDoCheck() | Detect and act upon changes that Angular can't or won't detect on its own.  Called during every change detection run, immediately after ngOnChanges() and ngOnInit(). |
| ngOnDestroy() | Cleanup just before Angular destroys the directive/component. Unsubscribe Observables and detach event handlers to avoid memory leaks.  Called just before Angular destroys the directive/component. |

# Memory Leak

A failure in a program to release discarded (unwanted) memory, causing impaired (decreased) performance or failure. A memory leak reduces the performance of the computer by reducing the amount of available memory. Eventually, in the worst case, too much of the available memory may become allocated and all or part of the system or device stops working correctly, the application fails, or the system slows down.

Once you subscribe to one, it will keep working until you unsubscribe, even if you navigate to another view. We must unsubscribe before Angular destroys the component. Failure to do so could create a memory leak.

msgs$ = httpService.listenToServer().subscribe(

msg => {this.serverMsgs.push(msg); console.log(msg)}

);

ngOnDestroy(){ this.msgs$.unsubscribe(); }

# Lazy Loading

Lazy loading is a design pattern commonly used in computer programming to defer initialization of an object until the point at which it is needed.

**Angular Module Loading: Eager, Lazy and Preloading**

Link: <https://www.concretepage.com/angular-2/angular-module-loading-eager-lazy-and-preloading>

# Design Pattern

Design Pattern is a general repeatable solution to a commonly occurring problem in software design.

# Observables

Observables provide support for passing messages between publishers and subscribers in your application.

Eg:

**Service**

import { BehaviorSubject } from 'rxjs/BehaviorSubject';

In Constructor add

this.radioBindSource = new BehaviorSubject('NoData');

this.currentRadioBind = this.radioBindSource.asObservable();

Create function

bindRadioBtnVal(bindRadioVal: any) {

this.radioBindSource.next(bindRadioVal);

}

**Component1 (publisher)**

this.\_pageDataService.bindRadioBtnVal(dataBinding);

**Component 2 (subscriber)**

this.\_pageDataService.currentRadioBind.subscribe(data => {

if (data !== "NoData" && data.rdKey && data.rdVal) {

let arrRDKeys = data.rdKey.toString().split(',');

let arrRDVal = data.rdVal.toString().split(',');

let keyValList = [];

for (let x = 0; x < arrRDKeys.length; x++) {

keyValList.push({ 'name': arrRDVal[x], 'value': arrRDVal[x] });

}

\_self.rdKeyValList = keyValList;

}

});

# RxJS

RxJS (Reactive Extensions for JavaScript) is a library for reactive programming using observables.

# Reactive Programming

Reactive programming (asynchronous dataflow programming) is about writing code that define how to react to changes: user input, data coming from a stream, a change in the state of a system, etc.

# Functional Programming

Functional programming (often abbreviated FP) is the process of building software by composing pure functions, avoiding shared state, mutable data, and side-effects. Functional programming is declarative rather than imperative, and application state flows through pure functions.

# Arrow Function

In TypeScript, everything that comes after the **:** is the type information.

Eg: dummyNum: boolean

=> defines a function type where the arguments are to the left of the => and the return type is on the right.

argument => {argument.length};

will convert to

function (argument) { argument.length; };

# Map

The map() method creates a new array with the results of calling a function for every array element.

The map() method calls the provided function once for each element in an array, in order.

**Note:** map() does not execute the function for array elements without values.

**Note:** map() does not change the original array.

.map((res : Response) => res.json())

# Filter

Filter allows us to filter an array, and return an array containing only the matching items.

var booksByStoreID = books.filter(book => book.store\_id === this.store.id)

# Mutable and Immutable

A mutable object can be changed after it's created, and an immutable object can't.

In JavaScript, only objects and arrays are mutable. String and Numbers are Immutable.

You can make a variable name point to a new value, but the previous value is still held in memory.

# Imperative vs Declarative

|  |  |
| --- | --- |
| **Imperative** | **Declarative** |
| Imperative code focuses on creating statements that change program states by creating algorithms that tell the computer how to do things. | Declarative code focuses on building logic of software without actually describing its flow. You are saying what without adding how |
| Eg: C, C++, C#, PHP, Java | Eg: HTML, XML, CSS, SQL |
| Typically your code will make use of conditional statements, loops and class inheritance. | For example with HTML you use <img src="./image.jpg" /> to tell browser to display an image and you don’t care how it does that. |

# Closure

A closure is an inner function that has access to outer function’s variables in addition to it’s own variable and global variables. In simple term a closure is a function inside a function.

function employee() {

let employee\_dept = 'IT';

return {

getDept: function () {

return employee\_dept;

},

setDept: function (new\_dept) {

employee\_dept = new\_dept;

}

}

}

let emp1 = employee();

console.log(emp1); // {getDept: ƒ, setDept: ƒ}

console.log(emp1.getDept()); // IT

emp1.setDept('Account');

console.log(emp1.getDept()); // Account

# Interface

An interface is a group of related properties and methods that describe an object, but neither provides implementation nor initialisation for them. An interface will not generate any code, it is only used by Typescript for type checking during development. Interfaces are only at compile time. This allows only you to check that the expected data received follows a particular structure.

this.http.get('...')

.map(res => <Product[]>res.json());

Example of interface:

export interface IProduct {

ProductNumber: number;

ProductName: string;

ProductDescription: string;

}