Angular Complete Notes (Simple Language with Syntax and Interview Prep)

1. What is Angular?

Angular is a front-end web framework built using TypeScript. It is used to build Single Page Applications (SPAs), where the page doesn't reload while navigating.

A **Single Page Application (SPA)** is a **web application** that **loads a single HTML page** and dynamically **updates the content without reloading the entire page** as the user interacts with it.

1. Angular Architecture:

* **Modules**: A way to group components, directives, pipes, and services. Every app has a root module AppModule.
* **Components**: Control views (UI). Decorated with @Component.
* **Templates**: HTML with Angular syntax to render UI.
* **Directives**: Instructions in templates (e.g. \*ngIf, \*ngFor).
* **Services**: Business logic shared across components. Decorated with @Injectable().
* **Dependency Injection**: Angular injects services into components automatically.

1. Angular CLI:

* Used to generate components, services, modules, etc.

ng new app-name

ng serve

ng generate component my-component

ng build --prod

1. Components:

* Defined with @Component decorator.

@Component({

selector: 'app-hello',

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

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

})

export class HelloComponent implements OnInit {

ngOnInit() {}

}

**Lifecycle Hooks:**

* ngOnInit(): Called after component is initialized.
* ngOnDestroy(): Called just before component is destroyed.

1. Data Binding:

{{ title }} <!-- Interpolation -->

[inputValue]="data" <!-- Property binding -->

(click)="handleClick()" <!-- Event binding -->

[(ngModel)]="data" <!-- Two-way binding -->

1. Directives:

* **Structural**: Change structure (e.g., \*ngIf, \*ngFor).

<div \*ngIf="isLoggedIn">Welcome</div>

<li \*ngFor="let user of users">{{ user.name }}</li>

* **Attribute**: Change appearance (e.g., ngClass, ngStyle).

<p [ngStyle]="{color: 'red'}">Red text</p>

1. Pipes:

* Used to transform data in templates.

{{ dateValue | date:'short' }}

{{ amount | currency:'INR' }}

* Custom Pipe:

@Pipe({ name: 'customPipe' })

export class CustomPipe implements PipeTransform {

transform(value: string): string {

return value.toUpperCase();

}

}

1. Services & Dependency Injection:

@Injectable({ providedIn: 'root' })

export class UserService {

getUser() {}

}

constructor(private userService: UserService) {}

1. Routing & Navigation:

const routes: Routes = [

{ path: 'home', component: HomeComponent },

{ path: 'about', component: AboutComponent }

];

* Use <router-outlet> in template.
* Navigation:

<a routerLink="/home">Home</a>

this.router.navigate(['/home']);

1. Forms:

* **Template-Driven Forms:**

<form #f="ngForm">

<input name="username" ngModel required>

</form>

* **Reactive Forms:**

this.form = new FormGroup({

name: new FormControl('', Validators.required)

});

<form [formGroup]="form">

<input formControlName="name">

</form>

1. HTTP Client:

constructor(private http: HttpClient) {}

getData() {

this.http.get('https://api.example.com').subscribe(data => console.log(data));

}

1. Angular Material:

ng add @angular/material

Import required modules like:

import { MatButtonModule } from '@angular/material/button';

Use in template:

<button mat-raised-button>Click</button>

1. Lifecycle Hooks:

* ngOnInit() – Initialization logic.
* ngAfterViewInit() – When view is initialized.
* ngOnDestroy() – Clean up logic.

1. RxJS & Observables:

import { Observable } from 'rxjs';

let obs$ = new Observable(observer => {

observer.next('data');

});

obs$.subscribe(data => console.log(data));

**Operators:** map, filter, tap, switchMap

1. Standalone Components (Angular 14+):

@Component({

standalone: true,

selector: 'app-standalone',

template: `<h1>Hello</h1>`

})

export class StandaloneComponent {}

1. Signals (Angular 16+):

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

const counter = signal(0);

counter.set(1);

counter.update(n => n + 1);

1. Build & Deployment:

ng build --output-path docs --base-href /your-repo/

Host it on GitHub Pages or Netlify.

1. Testing:

* **Unit Testing**: Jasmine + Karma

it('should return sum', () => {

expect(service.add(2, 3)).toEqual(5);

});

* **E2E Testing**: Cypress

1. Best Practices:

* Use OnPush change detection.
* Use trackBy in \*ngFor for performance.
* Use shared and core modules properly.
* Unsubscribe from Observables.
* Lazy load large modules.

These notes now include definitions, syntax, and elaboration for interview preparation.

**🔹 ng-content**

* Used for **content projection** — lets you insert external (parent) content into a component’s template.
* Enables **reusable** and **flexible** component design.

✅ Example:

html

<!-- Parent -->

<app-card>

<p>Hello World</p>

</app-card>

<!-- Child Template -->

<div class="card">

<ng-content></ng-content>

</div>

**🔹 @ViewChild**

* Used to **access a single element, directive, or child component** from the template.
* Returns the **first matching element**.
* Commonly used to manipulate DOM or access child component methods.

✅ Example:

Ts

@ViewChild('myDiv') divRef!: ElementRef;

ngAfterViewInit() {

console.log(this.divRef.nativeElement.innerText);

}

html

<div #myDiv>Hello</div>

**🔹 @ViewChildren**

* Used to **access multiple elements or components** in the template.
* Returns a **QueryList**, which can be iterated over.
* Useful when working with repeated elements or dynamic components.

✅ Example:

ts

@ViewChildren('item') items!: QueryList<ElementRef>;

ngAfterViewInit() {

this.items.forEach(item => console.log(item.nativeElement.innerText));

}

html

<div #item>Item 1</div>

<div #item>Item 2</div>

<div #item>Item 3</div>

**ngModel vs [(ngModel)]**

* **ngModel**: One-way binding from component to view. Sets the input value but does **not** update the component when the input changes.
* **[(ngModel)]**: Two-way binding (banana in a box syntax). Synchronizes the input value **both ways**—updating the component property when input changes, and updating input when component changes.

**Example:**

html

<!-- One-way binding -->

<input [ngModel]="name" />

<!-- Two-way binding -->

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

**\*ngIf**

* Used for **conditional rendering** of a single template block.
* Adds or removes elements from the DOM based on a boolean condition.
* Example:

html

<div \*ngIf="isLoggedIn">Welcome back!</div>

* Only **one condition** checked (true/false).

**\*ngSwitch**

* Used for **conditional rendering** among multiple possible views.
* Works like a switch-case statement.
* Example:

html

<div [ngSwitch]="userRole">

<div \*ngSwitchCase="'admin'">Admin Panel</div>

<div \*ngSwitchCase="'user'">User Dashboard</div>

<div \*ngSwitchDefault>Guest View</div>

</div>

* Allows multiple cases and a default.

**Summary**

| **Directive** | **Usage** |  | **Condition Type** | **DOM Manipulation** |
| --- | --- | --- | --- | --- |
| \*ngIf | Show/hide one block |  | Boolean (true/false) | Adds/removes element |
| \*ngSwitch | Choose one among many blocks |  | Match a value against cases | Renders matched case only |

**Observable vs Promise**

| **Feature** | **Observable** | **Promise** |
| --- | --- | --- |
| **Data type** | Stream of multiple values over time | Single future value |
| **Lazy or eager** | Lazy — does not execute until subscribed | Eager — starts immediately |
| **Cancellable** | Yes — can unsubscribe | No — cannot be cancelled |
| **Operators support** | Yes — supports map, filter, retry, etc. | No — only .then(), .catch() |
| **Multiple values** | Can emit multiple values | Resolves once with single value |
| **Usage** | Best for multiple async events, streams (like user input, websockets) | Best for single async operation (like HTTP request) |

**Calling a Promise**

**typescript**

**// Function returning a Promise**

**function getPromiseData(): Promise<string> {**

**return new Promise((resolve) => {**

**setTimeout(() => resolve('Promise data received!'), 1000);**

**});**

**}**

**// Calling the Promise**

**getPromiseData()**

**.then(data => {**

**console.log(data); // Output after 1 second: Promise data received!**

**})**

**.catch(error => {**

**console.error(error);**

**});**

**Calling an Observable**

**typescript**

**import { Observable } from 'rxjs';**

**// Function returning an Observable**

**function getObservableData(): Observable<string> {**

**return new Observable(observer => {**

**setTimeout(() => {**

**observer.next('Observable data received!');**

**observer.complete();**

**}, 1000);**

**});**

**}**

**// Calling the Observable**

**getObservableData().subscribe({**

**next: data => console.log(data), // Output after 1 second: Observable data received!**

**error: err => console.error(err),**

**complete: () => console.log('Observable completed')**

**});**

| **Context** | **Syntax** |
| --- | --- |
| **Inside a class** | **methodName() { ... }** |
| **Standalone function** | **function functionName() { ... }** |

**Structural Directive**

* Changes **DOM structure** (adds/removes elements)
* Uses \* syntax (e.g., \*ngIf, \*ngFor)

**Attribute Directive**

* Changes **appearance or behavior** of existing elements
* No \* syntax, used as normal attributes (e.g., ngClass, ngStyle)

✅ **Structural Directive Example (\*ngIf)**

<div \*ngIf="isLoggedIn">Welcome, user!</div>

Shows the <div> only if isLoggedIn is true.

✅ **Attribute Directive Example (ngClass)**

<div [ngClass]="{'active': isActive}">Click me</div>

Applies the active CSS class if isActive is true.

**✅ BehaviorSubject**

* **Stores the latest value**.
* On new subscription: emits the **most recent value** immediately.
* Requires an **initial value**.

ts

const subject = new BehaviorSubject<number>(0); // initial value = 0

subject.next(1);

subject.subscribe(val => console.log(val)); // Output: 1 (latest value)

**✅ ReplaySubject**

* **Stores multiple previous values** (you decide how many).
* On new subscription: emits the **last N values** (defined by buffer size).
* **No initial value required**.

ts

const subject = new ReplaySubject<number>(2); // buffer size = 2

subject.next(1);

subject.next(2);

subject.next(3);

subject.subscribe(val => console.log(val)); // Output: 2, 3 (last 2 values)

**🔍 Summary Table:**

| **Feature** | **BehaviorSubject** | **ReplaySubject** |
| --- | --- | --- |
| Initial value | Required | Not required |
| Latest value | Emits latest only | Emits last N values |
| Use case | Current state or data | Playback of past values/events |

**✅ Subject – No Memory**

* You start saying answers: 1, 2, 3
* A student joins **after** you already said 1, 2
* He **only hears** 3 (misses past answers)

ts

const subject = new Subject<number>();

subject.next(1); // student not yet joined

subject.next(2); // student not yet joined

subject.subscribe(val => console.log('Student got:', val)); // joins late

subject.next(3); // ✅ Student hears: 3

🧠 Subject = **no memory**, gives data **only from the point of subscription**.

**✅ ReplaySubject – Remembers Past**

* Same teacher says 1, 2, 3
* A student joins **after** all are said
* ReplaySubject **remembers** past answers
* So the student **hears all**: 1, 2, 3

ts

const replaySubject = new ReplaySubject<number>(3); // remember last 3

replaySubject.next(1);

replaySubject.next(2);

replaySubject.next(3);

replaySubject.subscribe(val => console.log('Student got:', val));

// ✅ Output: 1, 2, 3

🧠 ReplaySubject = **remembers old values**, gives all to late joiners.

**Summary in One Line:**

| **Type** | **Remembers Past?** | **Late Subscribers Get Old Values?** |
| --- | --- | --- |
| Subject | ❌ No | ❌ No |
| ReplaySubject | ✅ Yes | ✅ Yes |

**Component vs Directive** in Angular:

**✅ Component**

* A **special type of directive** with a **template**.
* Defines a **UI block** (HTML + logic).
* Always has a **selector**, **template**, and **styles**.

ts

@Component({

selector: 'app-hello',

template: `<p>Hello Component</p>`

})

export class HelloComponent {}

🧠 Use when you want to **create UI elements** or views.

**✅ Directive**

* **Adds behavior** to an existing DOM element.
* Does **not have a template**.
* Two types:
  + **Attribute directive** (e.g., ngClass, custom highlight)
  + **Structural directive** (e.g., \*ngIf, \*ngFor)

ts

@Directive({

selector: '[appHighlight]'

})

export class HighlightDirective {

constructor(el: ElementRef) {

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

}

}

🧠 Use when you want to **change behavior or appearance** of an element.

**🔍 Summary Table:**

| **Feature** | **Component** | **Directive** |
| --- | --- | --- |
| Template | ✅ Yes | ❌ No |
| Purpose | Build UI | Add behavior to existing elements |
| Example | @Component({ ... }) | @Directive({ ... }) |

**Lazy Loading vs Eager Loading** in Angular:

**✅ Eager Loading**

* Loads **all modules at app startup**.
* Used for **core or frequently used features**.
* Modules are bundled and loaded **immediately**.

ts

// app.module.ts

import { DashboardModule } from './dashboard/dashboard.module';

@NgModule({

imports: [DashboardModule]

})

🧠 **Pros**: Simple, faster routing  
⚠️ **Cons**: Increases initial load time

**✅ Lazy Loading**

* Loads modules **only when needed** (on route access).
* Used for **rarely visited or large modules**.
* Configured in **routing** using loadChildren.

ts

// app-routing.module.ts

const routes: Routes = [

{

path: 'dashboard',

loadChildren: () =>

import('./dashboard/dashboard.module').then(m => m.DashboardModule)

}

];

🧠 **Pros**: Faster initial load, better performance  
⚠️ **Cons**: Slight delay on first route visit

**🔍 Summary Table:**

| **Feature** | **Eager Loading** | **Lazy Loading** |
| --- | --- | --- |
| Load Time | At application start | When route is accessed |
| Performance | Slower startup | Faster startup |
| Setup | Directly imported in AppModule | Uses loadChildren in routing |
| Use Case | Core/small modules | Large/rarely used modules |

**routerLink vs href in Angular:**

**✅ routerLink**

* **Angular-specific**.
* Uses Angular's **Router** to change pages **without reloading**.
* Keeps the app **single-page**.

html

<a routerLink="/home">Go to Home</a>

✅ Best for **internal navigation** in Angular apps.

**✅ href**

* **Standard HTML** attribute.
* Causes **full page reload**.
* Can be used for **external or internal links**.

html

<a href="/home">Go to Home</a>

⚠️ Not recommended for internal routing in Angular apps.

**🔍 Summary Table:**

| **Feature** | **routerLink** | **href** |
| --- | --- | --- |
| App reload | ❌ No | ✅ Yes (full reload) |
| SPA navigation | ✅ Yes | ❌ No |
| Usage | Internal Angular routes | External or non-SPA links |

**ActivatedRoute vs Router in Angular:**

**✅ ActivatedRoute**

* Gives **info about the current route**.
* Used to get:
  + Route parameters (params)
  + Query parameters (queryParams)
  + Fragment, data, etc.

ts

constructor(private route: ActivatedRoute) {

this.route.params.subscribe(params => {

console.log(params['id']);

});

}

📌 **Use when you want to read current route info.**

**✅ Router**

* Used to **navigate programmatically**.
* Also gives access to route events.

ts

constructor(private router: Router) {

this.router.navigate(['/home']);

}

📌 **Use when you want to change the route via code.**

**🔍 Summary Table:**

| **Feature** | **ActivatedRoute** | **Router** |
| --- | --- | --- |
| Purpose | Read info about current route | Navigate to a different route |
| Example Use | route.params, route.queryParams | router.navigate(['/path']) |
| Use Case | Get route data | Change the route |

**✅ HttpClient**

* The **official Angular service** for making HTTP requests.
* Comes from @angular/common/http.
* Supports **GET, POST, PUT, DELETE**, interceptors, typed responses, and observables.
* Recommended for all HTTP operations.

ts

import { HttpClient } from '@angular/common/http';

constructor(private http: HttpClient) {}

this.http.get('/api/data').subscribe(data => console.log(data));

**What are HTTP Interceptors?**

* Interceptors are **middleware** for HTTP requests and responses.
* They **intercept** every HTTP request before it’s sent and every response before it’s processed.
* Used to:
  + Add headers (e.g., auth tokens)
  + Handle errors globally
  + Log requests/responses
  + Modify request/response data

**What is an interceptor?**

* It’s a **special class** that runs **before every HTTP request** and **after every HTTP response**.
* You can **change or add things** to the request or response.

**Angular lifecycle hooks** :

**1. ngOnChanges**

* **When?** Called **when input properties change** (data passed from parent to child).
* **What?** Detect changes and react.
* **Example:**

ts

ngOnChanges(changes: SimpleChanges) {

console.log('Input changed:', changes);

}

**2. ngOnInit**

* **When?** Called **once after the first ngOnChanges**, when the component initializes.
* **What?** Used for component setup, fetch data here.
* **Example:**

Ts

ngOnInit() {

console.log('Component initialized');

}

**3. ngDoCheck**

* **When?** Called **on every change detection cycle** (very often).
* **What?** Detect and act on custom changes.
* **Example:**

ts

ngDoCheck() {

console.log('Change detection running');

}

**4. ngAfterContentInit**

* **When?** Called **once after content (ng-content) has been projected** into the component.
* **What?** Used when you want to act after content projection.
* **Example:**

ts

ngAfterContentInit() {

console.log('Content projected');

}

**5. ngAfterContentChecked**

* **When?** Called **after every check of projected content**.
* **What?** React after content changes.
* **Example:**

ts

ngAfterContentChecked() {

console.log('Content checked');

}

**6. ngAfterViewInit**

* **When?** Called **once after the component’s view (and child views) are initialized**.
* **What?** Used to access/view child components or DOM elements.
* **Example:**

ts

ngAfterViewInit() {

console.log('View initialized');

}

**7. ngAfterViewChecked**

* **When?** Called **after every check of the component’s view and child views**.
* **What?** Detect/view changes in your component’s view.
* **Example:**

ts

ngAfterViewChecked() {

console.log('View checked');

}

**8. ngOnDestroy**

* **When?** Called **just before the component is destroyed**.
* **What?** Clean up resources (unsubscribe, timers).
* **Example:**

ts

ngOnDestroy() {

console.log('Component destroyed');

}

**Summary Table:**

| **Hook** | **When Called** | **Purpose** |
| --- | --- | --- |
| ngOnChanges | When input properties change | React to input changes |
| ngOnInit | After first ngOnChanges | Component setup |
| ngDoCheck | Every change detection cycle | Custom change detection |
| ngAfterContentInit | After projected content loaded | Content projection init |
| ngAfterContentChecked | After every check of projected content | Content projection checked |
| ngAfterViewInit | After component view initialized | Access child views or DOM |
| ngAfterViewChecked | After every check of views | View checked |
| ngOnDestroy | Before component is destroyed | Cleanup |

**Why use these hooks?**

* ngOnChanges — react to input changes (like username)
* ngOnInit — initialize component & start timer or API calls
* ngDoCheck — detect custom changes (rarely used)
* ngAfterContentInit — after projected content loaded
* ngAfterContentChecked — after projected content checked
* ngAfterViewInit — after component’s view is ready
* ngAfterViewChecked — after view checked
* ngOnDestroy — clean up (clear timers, unsubscribe)

| **Version** | **Feature** | **Why Important** | **Sample Concept** |
| --- | --- | --- | --- |
| 14 | Typed Reactive Forms | Type safety in forms | FormControl<string> |
| 14-15 | Standalone Components | Simpler component/module setup | standalone: true |
| 16 | Signals | New reactive primitive | signal(0) |
| 19 | Strict Typed Forms & Routing | Better dev experience, simpler routing | Typed forms + standalone routing |

Angular 13 was a big step toward a more modern Angular. It removed the older View Engine in favor of Ivy only, which improved build speed and bundle size. The Angular CLI got better with persistent caching by default, speeding up development. Dynamic component creation got simplified with a cleaner API, and forms had improved typing support. Overall, Angular 13 focused on performance, simplicity, and modernizing the framework.

**What’s new in Angular 13 (Simple version)**

1. **Only Ivy Rendering Now**

* Angular stopped using the old rendering system (View Engine).
* Now it only uses Ivy, which makes apps faster and smaller.

1. **Faster Builds**

* Building your app is quicker because Angular remembers previous work (build cache).

1. **Easier to Create Components Dynamically**

* You don’t need extra code to create components on the fly. It’s simpler now.

1. **Better Support for TypeScript 4.4**

* Angular works well with the newer TypeScript version, helping catch errors early.

1. **Improved Forms and Testing**

* Forms have better typing, making it easier to avoid mistakes.
* Tests run faster and clean up better.

1. **Updated Angular Material**

* The UI component library is updated to work smoothly with Angular 13.

**In short:** Angular 13 made Angular apps faster, smaller, and easier to build and test by removing old stuff and improving the tools.