# \*\*TypeScript and Angular Concepts\*\*

## \*\*1. Generics in TypeScript\*\*

### \*\*What Are Generics?\*\*

Generics in TypeScript allow you to create reusable and type-safe components by using placeholders for types. Generics ensure that functions, classes, and interfaces work with different data types while maintaining type safety.

### \*\*Syntax of Generics\*\*

The syntax for defining a generic type uses angle brackets `<T>`:

```typescript

function getData<T>(value: T): T {

return value;

}

console.log(getData<number>(100)); // 100

console.log(getData<string>("Hello")); // Hello

```

### \*\*Generic Interfaces\*\*

```typescript

interface ApiResponse<T> {

success: boolean;

status: number;

data: T;

}

const response: ApiResponse<string> = {

success: true,

status: 200,

data: "This is a string",

};

```

### \*\*Generic Classes\*\*

```typescript

class Box<T> {

item: T;

constructor(item: T) {

this.item = item;

}

}

const numberBox = new Box<number>(10);

const stringBox = new Box<string>("Apple");

```

### \*\*Generic Constraints\*\*

To restrict the types used in generics, we use constraints:

```typescript

function logLength<T extends { length: number }>(value: T): void {

console.log(value.length);

}

logLength("Hello"); // ✅ Allowed (string has length)

logLength([1, 2, 3]); // ✅ Allowed (array has length)

// logLength(100); ❌ Error: number does not have 'length'

```

## \*\*2. Type Locking in TypeScript\*\*

Type locking ensures that variables, parameters, and function returns use strictly defined types, preventing unintended type assignments.

### \*\*1️⃣ Using Explicit Types\*\*

```typescript

let age: number = 25;

age = "twenty-five"; // ❌ Error: Type 'string' is not assignable to type 'number'

```

### \*\*2️⃣ Using \*\*\*\*\`\`\*\*\*\* for Immutability\*\*

```typescript

const user = { name: "Alice", age: 30 };

user.age = 35; // ✅ Allowed

user = { name: "Bob", age: 40 }; // ❌ Error: Assignment to constant variable

```

### \*\*3️⃣ Using \*\*\*\*\`\`\*\*\*\* for Deep Type Locking\*\*

```typescript

const colors = ["red", "green", "blue"] as const;

colors[0] = "yellow"; // ❌ Error: Cannot assign to '0' because it is a read-only property.

```

### \*\*4️⃣ Using Type Assertions (\*\*\`\`\*\*)\*\*

```typescript

let someValue: unknown = "This is a string";

let strLength: number = (someValue as string).length;

```

## \*\*3. Dependency Injection in Angular\*\*

### \*\*What is Dependency Injection?\*\*

Dependency Injection (DI) is a design pattern where dependencies are injected into a class rather than being created inside the class. This promotes modularity and testability.

### \*\*Example of Dependency Injection in Angular\*\*

1. \*\*Create a Service\*\*

```typescript

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

@Injectable({

providedIn: 'root',

})

export class DataService {

getData() {

return 'Hello from DataService';

}

}

```

2. \*\*Inject the Service into a Component\*\*

```typescript

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

import { DataService } from './data.service';

@Component({

selector: 'app-root',

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

})

export class AppComponent {

message: string;

constructor(private dataService: DataService) {

this.message = this.dataService.getData();

}

}

```

## \*\*4. Different Types of Binding in Angular\*\*

Angular provides different types of data binding to communicate between components and templates.

### \*\*1️⃣ Property Binding (\*\*\`\`\*\*)\*\*

Property binding binds a property in the component to an element in the template.

```html

<input [value]="name">

```

```typescript

export class AppComponent {

name = "Angular";

}

```

### \*\*2️⃣ Event Binding (\*\*\`\`\*\*)\*\*

Event binding listens for user events like clicks and keystrokes.

```html

<button (click)="showMessage()">Click Me</button>

```

```typescript

showMessage() {

alert("Button Clicked!");

}

```

### \*\*3️⃣ Two-Way Binding (\*\*\`\`\*\*)\*\*

Two-way binding updates both the component and the template.

```html

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

<p>Hello {{ name }}</p>

```

```typescript

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

@Component({

selector: 'app-root',

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

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

})

export class AppComponent {

name: string = "";

}

```

> \*\*Note:\*\* To use `ngModel`, import `FormsModule` in `app.module.ts`.

### \*\*4️⃣ Attribute Binding (\*\*\`\`\*\*)\*\*

Attribute binding sets custom attributes dynamically.

```html

<a [attr.href]="link">Click Here</a>

```

```typescript

export class AppComponent {

link = "https://angular.io";

}

```

---

This document covers the key concepts of TypeScript generics, type locking, dependency injection, and Angular binding types. Mastering these topics will improve your Angular development skills significantly.

In Angular, **Dependency Injection (DI)** is a design pattern that allows you to inject services or dependencies into components, directives, pipes, or other services instead of creating them manually. This improves modularity, testability, and maintainability of the application.

### ****What is Dependency Injection in Angular?****

Dependency Injection in Angular is a way to **pass dependencies (such as services) into components** rather than creating them inside the component. Angular has a built-in **injector** that takes care of providing dependencies when required.

### ****How Dependency Injection Works****

1. **A service is created** using @Injectable() decorator.
2. **The service is registered** in the provider list (either in a module, component, or root level).
3. **Angular's DI system injects** the service wherever it is required.

### ****Example: Injecting a Service in Angular****

#### ****1. Create a Service****

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

@Injectable({

providedIn: 'root', // This makes the service available globally

})

export class DataService {

getData() {

return 'Hello from DataService';

}

}

#### ****2. Inject the Service into a Component****

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

import { DataService } from './data.service';

@Component({

selector: 'app-example',

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

})

export class ExampleComponent {

message: string;

constructor(private dataService: DataService) {

this.message = this.dataService.getData(); // Injecting service

}

}

### ****Benefits of Dependency Injection in Angular****

* **Code Reusability**: Services can be reused across different components.
* **Loose Coupling**: Components depend on abstractions rather than concrete implementations.
* **Improved Testing**: Easier to write unit tests by injecting mock dependencies.
* **Better Maintainability**: Centralized service logic makes the app more manageable.

Would you like an example with multiple services or a real-world scenario? 😊