1.Basics:

What is a Framework?

1. Foundational structures known as frameworks provide developers a head start in creating apps. They remove the requirement to start coding from scratch, freeing up developers to focus on resolving specific issues and adding unique features.
2. One web framework that is specifically meant for web apps is called Angular. It provides a set of pre-established guidelines and structures, streamlining the development process.This improves the effectiveness of developing web applications while also saving time.

- **What is client-server communication?**

Client-server communication is the fundamental mechanism that allows web applications to interact with servers. The process unfolds as follows:

1. Sending Requests: It commences with the client, typically a web browser, initiating an HTTP request. This request is a trigger for the server to begin processing.
2. Server Processing: The server receives the HTTP request and begins processing it. Operations such as retrieving data from a database or executing specific tasks are carried out at this stage.
3. Sending Responses: Following the server's processing, it sends back an HTTP response to the client, conveying requested data. This response also communicates the success or failure of the operation initiated by the client.

What is JavaScript?

JavaScript, a versatile programming language, boasts several key attributes:

- Dynamically Typed: JavaScript is dynamically typed, eliminating the need for explicit data type declarations. This flexibility facilitates quick and dynamic development, but developers must exercise caution to prevent potential runtime errors.
- Object-Oriented: JavaScript adopts an object-oriented paradigm, enabling developers to work seamlessly with objects, classes, and inheritance. This approach enhances code organization and reusability.
- Event-Driven: JavaScript is inherently event-driven, meaning it responds to events triggered by user actions (e.g., button clicks, mouse movements) or other components within the application. This responsiveness contributes to the creation of interactive and user-friendly web applications.
- Cross-Browser Compatible: JavaScript enjoys widespread support and is compatible with all major web browsers, including Chrome, Firefox, Safari, Edge, and others.

ANGULAR:

o This cross-browser compatibility ensures a consistent user experience across different platforms.

What is Typescript?

TypeScript is a superset of JavaScript, introducing static typing to enhance the development process. Here's a succinct overview:

- Static Typing: TypeScript employs static typing, allowing developers to explicitly define variable types, function parameters, and return types.
- The TypeScript compiler leverages this information to catch type-related errors during the development phase.
- This proactive approach enhances code quality and helps prevent potential runtime errors.

Error Prevention in Development:

- By catching type-related errors early in the development process, TypeScript provides a layer of error prevention. This is particularly valuable in larger-scale projects where the early detection of issues contributes to overall code stability.
- ntegration with Modern Web Frameworks: TypeScript is frequently integrated into larger-scale projects, especially those utilizing modern web frameworks like Angular. Its static typing features align well with the structured architecture of such frameworks, promoting a more robust and maintainable codebase.

What is Angular?

a. Definition:

Not a Programming Language:

o Angular is not a programming language but a comprehensive framework for building dynamic, single-page web applications (SPAs).

 Framework:

o As a framework, Angular provides a structured environment with predefined rules that make it easier for developers to efficiently write code.

b. Framework Characteristics:

 Structured Approach:

o Angular offers a structured approach to web development, enforcing guidelines and patterns for creating maintainable and scalable applications.

 Component-Based:

o Angular follows a component-based architecture, allowing developers to break down complex applications into smaller, reusable parts called components.

 Code Reusability:

o Components can be created once and reused across multiple projects, promoting code reusability and reducing development time.

c. Web Development:

 Building Websites:

o Angular is specifically designed for web development, providing tools and features that simplify the process of creating dynamic and interactive websites.

d. Single Page Web Applications (SPAs):

 Contrast with Multi-Page Applications:

o In a multi-page application, each user interaction triggers a request to the server, resulting in a full page reload.

o SPAs, on the other hand, load a single HTML page initially and dynamically update the content as the user interacts with the application.

 Advantages of SPAs:

o SPAs provide a smoother user experience by avoiding full page reloads, leading to faster navigation and improved performance.

e. Dynamic and Interactive:

 Content Updates:

o Angular enables the creation of dynamic and interactive websites where content can change dynamically without requiring the entire page to reload.

o Two-way data binding and real-time updates contribute to the dynamic nature of Angular applications.

What is SPA and why do we use single page applications?

a. Definition:

 Web Application Type:

o A Single Page Application (SPA) is a specific type of web application that interacts with the user by dynamically rewriting the current page, rather than loading entire new pages from the server.

b. Operational Mechanism:

 Initial Loading:

o In SPAs, the initial HTML, CSS, and JavaScript resources are loaded when the user first accesses the application.

 Subsequent Interactions:

o Unlike traditional multi-page applications, SPAs do not require a full page reload with each user interaction.

o Instead, subsequent interactions or changes in content are handled by dynamically updating the existing page through asynchronous requests.

c. Key Characteristics:

 Dynamically Rewriting Pages:

o SPAs dynamically rewrite the content of the current page based on user interactions, providing a seamless and continuous user experience.

 Asynchronous Requests:

o Dynamic updates are often achieved through asynchronous requests, commonly using technologies like AJAX (Asynchronous JavaScript and XML) or more

modern approaches like Fetch API.

d. Advantages of Single Page Applications:

 Faster User Experience:

o SPAs offer a faster user experience as the initial page load includes only essential resources, and subsequent interactions are handled without reloading the entire page.

- Smooth Navigation:

o Since SPAs avoid full page reloads, transitions between different views or sections of the application are smoother, providing a more fluid and responsive feel.

- Reduced Server Load:

o SPAs reduce the load on the server as only data, not entire HTML pages, needs to be transmitted in response to user interactions.

e. Technologies Used:

 JavaScript Frameworks:

o SPAs are often built using JavaScript frameworks like Angular, React, or Vue.js, which provide tools and features to facilitate the development of dynamic and interactive user interfaces.

f. Use Cases:

 Rich Web Applications:

o SPAs are particularly suitable for building rich web applications where a fluid and engaging user experience is essential.

o Common use cases include social media platforms, interactive dashboards, and real-time collaboration tools.

Practical applications of Angular?

- Customer Relationship Management (CRM) Systems

- Travel and Booking Platforms

- E-commerce Platforms

- Healthcare Applications

- Real-Time Chat Applications

History/Versions of Angular:

- AngularJS (Version 1.x):

o Release Date: Released in 2010.

o Key Features:

- Introduced by Google as a JavaScript-based open-source front-end framework.
- Utilized two-way data binding for automatic synchronization between the model and the view.
- Pioneered the concept of directives for creating reusable components.

- **Angular 2+:**
- Release Date: Angular 2 was released in 2016, followed by subsequent versions.
- Key Changes:

- A complete rewrite of AngularJS, Angular 2 introduced a component-based architecture.

- Shifted from JavaScript to TypeScript for enhanced developer experience and code maintainability.
- Embraced reactive programming with the introduction of RxJS.

- Current Version (Angular 17.0.0):
- Release Date: Released on November 8, 2023.
- Key Features:
- Continuation of Angular's evolution with enhancements, bug fixes, and new features.

- Ongoing commitment to providing a robust framework for modern web development.
- Improved performance, security, and support for the latest web standards.

Why we don’t use AngularJs and use only Angular?

- **Performance Limitations:**
- AngularJS:

o AngularJS faced performance challenges, especially with larger applications due to its digest cycle and two-way data binding mechanism.

- Angular (2+):

o Angular's architecture, built from the ground up, addresses performance issues by introducing a more efficient change detection mechanism.

- **Mobile Responsiveness:**

o AngularJS:

- While AngularJS supports mobile development, it may not provide the same level of optimization for mobile applications as newer Angular versions.

- Angular (2+): Angular versions beyond AngularJS are designed with mobile responsiveness in mind, catering to the growing demand for mobile-friendly web applications.

 TypeScript Integration:

 AngularJS:

 AngularJS primarily relies on JavaScript, which lacks the static typing benefits of TypeScript.

 Angular (2+):

 Angular's transition to TypeScript brings static typing to the forefront, enhancing code maintainability, reducing errors, and promoting better developer tooling.

*******************Stay consistent****************

2.Setup and Installation

1. IDE Visual Studio Code

Step 1: Download VS Code from its official website: https://code.visualstudio.com/download

Step 2: Carry out the steps shown in the video

2. For Node.JS

Step 1: Download Node.js through its official website: https://nodejs.org/en

(Download LTS latest version available)

Step 2: Check if installed or not

for that open your Command Prompt and run command: node –version

3. Install Angular JS

Step 1: Open a terminal or command prompt and run the following command to install AngularJS

globally on your machine: npm install -g @angular/cli

Step 2: Check if it is installed or not

for that open your Command Prompt and run command: ng --version

4. By default, on Windows client computers, the execution of PowerShell scripts is restricted. To enable the execution of PowerShell scripts, a necessary prerequisite for installing npm global binaries, you must adjust the execution policy as follows:

Step 1: First, you need to open the command prompt and run this command:

```
set-ExecutionPolicy RemoteSigned -Scope CurrentUser
```

Step 2: Now you have to run the second command on your system. This command is:

```
Get-ExecutionPolicy
```

Step 3: If you want to view their policy, you need to run this command in your command prompt:

```
Get-ExecutionPolicy -list
```

5. Create a workspace and running the Application

Step 1: Create a Workspace

For this run the command: ng new my-app

Step 2: Navigate to the Project Folder

Run the command: cd my-app

Step 3: Run the Application

Run the command: ng serve

****************Stay consistent***************

3.Angular Architecture

Explain Angular Architecture

Modules

- Application is organized into modules, which are containers for different parts of applications helps in organizing and separation within an application.

- Each module can have its own components, services, and other features.

Component

- These are basic building blocks of Angular applications.

- Each component consists of a TypeScript class that defines the component's behavior and an HTML template that defines its view.

Templates

- Templates are written in HTML and define the view of a component.

Directives

- Directives are markers on a DOM element that tell Angular to attach a behavior to it. Angular comes with built-in directives like ngif for conditional rendering and ngfor for iterating over lists.

Services

- Services are used for organizing and sharing code across components. They are singleton objects that can be injected into components, providing a way to encapsulate and share functionality.

Router

- Angular includes a router module that provides a way to navigate between different components and views in a single-page application.

Forms.

- Angular has a robust forms module that simplifies the handling of user input and form validation.

Angular CLI (Command Line Interface)

- Angular CLI is a command-line tool that helps developers’ scaffold, build, test, and deploy Angular applications.

It provides a set of commands that automate common development tasks, making it easier to manage and maintain Angular projects.

Project Scaffolding

- Angular CLI can be used to create the basic structure of an Angular application, including modules, components, services, and other necessary files.

Code Generation

- Angular CLI simplifies the process of generating code for components, services, modules, directives, and more. This reduces the boilerplate code developers need to write manually.

Build and Deployment

- CLI provides commands to build production-ready bundles of the application that can be easily deployed to servers or cloud platforms.

Angular Imports:

In TypeScript the import keyword is used to bring in functionalities from other files or modules.

The basic syntax is: import { Symbol } from 'module';

Angular provides a set of core modules that are commonly imported into other modules. For example, the @angular/core module includes essential features like decorators (Component, Ng Module), dependency injection, and other core functionalities.

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

In addition to importing Angular core modules, you'll often import custom modules and services created within your application.

```
import { MyService } from './my-service';
```

```
import { SharedModule } from '../shared/shared.module';
```

Angular imports play a crucial role in creating modular, organized, and reusable code. They allow you to bring in functionalities from other modules, making it easier to manage dependencies and build complex applications by assembling smaller, well-defined parts.

Angular Decorators and Selectors:

Decorators are special types of declarations in TypeScript that are used to modify the structure or behavior of classes or class members. Angular comes with a set of built-in decorators that are used to configure and enhance various elements in an Angular application.

- @Component: Used to define a component and its metadata.

- @Directive: Used to define a directive and its metadata.

- @Injectable: Used to define a service and its metadata for dependency injection.

- @NgModule: Used to define a module and its metadata.

The decorator function receives information about the decorated item and can modify its behavior, add metadata, or perform other tasks.

The @Component decorator is commonly used to define Angular components. It takes a metadata object as an argument, which provides information about the component.

- selector: Specifies the HTML selector for the component ('app-root' in this case).

- templateUrl: Specifies the external HTML template file for the component.

- styleUrls: Specifies an array of external style files for the component.

@Component decorator is the key decorator used in this code, providing metadata to define the behavior and appearance of the AppComponent.

Once the component is defined with a selector, you can use it in other templates by using its custom HTML tag.

```
<app-root></app-root>
```

Selectors should be unique within the application to avoid naming conflicts. They allow you to create reusable components that can be easily identified and included in different parts of the application.

Angular Classes

Classes as Blueprints:

Example:

Let's say you have a Person class. It has an age property and a celebrateBirthday method.

```
export class Person {
age: number = 25; //Properties are like pieces of information
celebrateBirthday(): void { //methods are like actions.
this.age++;
}
}
```

Angular Modules

An Angular module is like a container or a box that holds related pieces of your application together.

Why Do We Need Modules?

Imagine you are building a house.

Each room in the house has a specific purpose, like a kitchen for cooking, and a living room for relaxing.

Similarly, in Angular, modules help you organize and separate different parts of your application.

Benefits of Modules:

Organization

- Modules help you keep your code organized by grouping related parts together.

Reusability

- You can reuse modules in different parts of your application or even in other Angular projects.

Maintainability

- Smaller, focused modules are easier to maintain than a large, monolithic codebase.

Dependency Management

- Modules allow you to manage dependencies between different parts of your application.

Angular Components

They are responsible for encapsulating the logic and view related to a particular part of the application. Components are used to create modular and reusable pieces of the user interface, making it easier to manage and maintain complex applications.

In Angular, a component is a fundamental building block of a user interface (UI). Components are self-contained, reusable units that encapsulate the behavior and presentation of a part of the UI.

***************Stay Consistent**************

4.STANDALONE COMPONENTS:

Standalone components provide a simplified way to build Angular applications. Standalone components,

directives, and pipes aim to reduce the need for NgModules.

Components, directives, and pipes can now be marked as standalone: true.

Angular classes marked as standalone do not need to be declared in an NgModule (the Angular compiler will report an error if you try).

Standalone components specify their dependencies directly instead of getting them through NgModules.

For example, if Component 1 is a standalone component, it can directly import another standalone component Component 2.

VARIABLES

A variable is a named storage location (a memory location) that holds a value. Variables are used to store

and manage data within a program. Each variable has a unique identifier (its name) that allows developers

to reference and manipulate the stored value.

DATATYPES:-

Primitive Data Types:

- Primitive data types are simple and directly hold a value.
- The values of primitive types are stored directly in memory locations.
- Operations on primitive types are generally faster because they involve working directly with the values stored.

Number: Represents numeric values, including integers and floating-point numbers.

Example: let count: number = 42;

Boolean: Represents true or false values.

Example: let isActive: boolean = true;

String: Represents textual data enclosed in single or double quotes.

Example: let name: string = "ABC";

Null: A special value that represents the intentional absence of any object value. It is a way to explicitly say that a variable or object property has no value.

Example: let myVar: null = null;

Undefined: A value represents a variable that has been declared but not assigned a value. Like null, you can explicitly assign undefined to a variable or use it as a type. It is automatically assigned to variables

that have not been assigned a value.

Example: let myVar: undefined = undefined; string: Represents textual data.

Non-Primitive (Reference) Data Types:

- Non-primitive data types store references to memory locations where the data is stored.

- They are more complex data structures compared to primitives.

- Operations on non-primitive types involve accessing or modifying the data indirectly through references.

Array: Represents an ordered list of values of the same type or a combination of types.

Example: let numbers: number[] = [1, 2, 3];

Object: Represents a collection of key-value pairs where values can be of any data type.Keys act as

identifiers for values, providing a way to access or retrieve specific pieces of information within an object

Objects allow you to group related pieces of information together under a single variable.

For example, you might have an object representing a person with keys like name, age, and address

Example: let person: { name: string, age: number } = { name: "ABC", age: 25 };

Any: Any is a TypeScript data type that represents a variable that can hold values of any data type. It essentially turns off TypeScript's type checking for a particular variable, allowing it to be assigned any value without type enforcement.

```
let dynamicValue: any = 5;

dynamicValue = "Hello";

dynamicValue = true;
```

While any provides flexibility, it comes with the trade-off of losing the benefits of static type checking, which is one of TypeScript's main features. It's generally recommended to avoid using any, when possible, to maintain type safety.

Void: Represents the absence of a type. Often used as the return type of functions that do not return a value. When a function has a void return type, it means the function doesn't produce a meaningful result.

Example: function logMessage(): void { console.log("Hello, world!"); }

GLOBAL AND LOCAL VARIABLES:

- Global variables are variables declared outside of any function or block. They have global scope, meaning

they are accessible from any part of the code, including functions and blocks.

- Global variables are accessible throughout the entire program. They can be accessed and modified from

any function or block.

- The lifetime of a global variable extends throughout the entire program. It is created when the program

starts and remains in memory until the program finishes.

Local variables: - are variables declared within a function or block.

- They have local scope, meaning they are only accessible within the function or block where they are declared.
- Local variables are declared using var, let, or const within a function or block.

- Local variables are accessible only within the function or block where they are declared. They cannot be

accessed from outside that context.

- The lifetime of a local variable is limited to the execution context of the function or block. It is created when the function or block is entered and ceases to exist when the function or block exits.
- It's generally advisable to minimize the use of global variables to avoid unintended side effects and

potential conflicts. Local variables provide encapsulation and help in creating modular and maintainable

code.

What is var, let and const?

```
example() {

if (true) {

var a = 20;

console.log(a); // Output: 20
}

console.log(a); // Output: 20

}
```

In the above example,

- if you use var this works
- var is function-scoped. It does not have block-level scope if you use let and const it gives error on 2nd console
- It's block-scoped. It is only accessible within the block (or statement) where it's defined.

(var and let) variables override when we update their values but const values can’t be changed.

var:

var is function-scoped. It does not have block-level scope. If a variable is declared inside a block (like an if statement), it is accessible throughout the entire function.

You can reassign values to a variable declared with var.

let:

let is block-scoped. It is only accessible within the block (or statement) where it's defined.

You can reassign values to a variable declared with let.

const:

const is also block-scoped like let. It is only accessible within the block (or statement) where it's defined.

Variables declared with const cannot be reassigned. They are constant.

****************Stay Consistent**************

5.Operators

Arithmetic Operators: A mathematical function that takes two operands and calculates anything with them is called an arithmetic operator.

It performs the following calculations:

1. Addition (+): Adds two values.

```
let sum = 5 + 3; // sum is 8
```

2. Subtraction (-): Subtracts the right operand from the left operand.

```
let difference = 10 - 4; // difference is 6
```

3. Multiplication (*): Multiplies two values.

```
let product = 2 * 3; // product is 6
```

4. Division (/): Divides the left operand by the right operand.

```
let quotient = 8 / 2; // quotient is 4
```

5. Modulus (%): Returns the remainder of the division of the left operand by the right operand.

```
let remainder = 9 % 4; // remainder is 1
```

6. Increment (++), Decrement (–): Increases or decreases the value of a variable by 1.

```
let x = 5;

x++; // x is now 6

x–; // x is back to 5
```

Comparison Operators: Comparison operators are used for evaluations and comparisons of strings or integers. Unlike arithmetic expressions, comparison operator expressions do not return a numerical value.

Comparison expressions return either 1, which represents true, or 0, which represents false.

1. Less Than (<): Checks if the left operand is less than the right operand.

```
let isLess = 3 < 5; // isLess is true
```

2. Greater Than (>): Checks if the left operand is greater than the right operand.

```
let isGreater = 7 > 4; // isGreater is true
```

3. Less Than or Equal To (<=): Checks if the left operand is less than or equal to the right operand.

```
let isLessOrEqual = 5 <= 5; // isLessOrEqual is true
```

4. Greater Than or Equal To (>=): Checks if the left operand is greater than or equal to the right operand.

```
let isGreaterOrEqual = 8 >= 8; // isGreaterOrEqual is true
```

5. Not Equal (!==): Checks if the left operand is not equal to the right operand.

```
let notEqual = 2 !== 4; // notEqual is true
```

6. Equal (==): Checks if the left operand is equal to the right operand (type coercion may occur).

```
let isEqual = '2' == 2; // isEqual is true
```

Logical Operators

1. Logical AND (&&): Returns true if both operands are true.

```
let andResult = (5 > 3) && (2 < 4); // andResult is true
```

2. Logical OR (||): Returns true if at least one operand is true.

```
let orResult = (5 < 3) || (2 > 1); // orResult is true
```

3. Logical NOT (!): Returns the opposite boolean value of the operand.

```
let notResult = !(4 > 2); // notResult is false
```

Assignment Operators: A variable can be assigned a value using assignment operators. A value is the operand on the right side of the assignment operator, whereas a variable is on the left side. To avoid the compiler raising an error, the value on the right side needs to be of the same data-type as the variable on the left side.

1. Assignment (=): Assigns the value of the right operand to the left operand.

```
let x = 20; // x is assigned the value 20
```

2. Addition Assignment (+=): Adds the right operand to the left operand and assigns the result to the left operand.

```
let y = 5;

y += 3; // y is now 8
```

3. Subtraction Assignment (-=): Subtracts the right operand from the left operand and assigns the result to the left operand.

```
let z = 7;

z -= 2; // z is now 5
```

Ternary Operator: Unlike other operators, which typically utilize one or two operands, the ternary operator requires three operands. It offers a way to shorten a simple if else block.

1. Ternary (condition? true : false): If the condition is true, it returns the true expression; otherwise, it returns the false expression.

```
let age = 20;

let status = (age >= 18) ? 'Adult' : 'Minor';

// status is 'Adult' because age is 20
```

Bitwise Operators:

An operator that manipulates individual bits is referred to as a bitwise operator. It can be applied to bit patterns or binary numerals.

1. Bitwise AND (&): Performs a bitwise AND operation.

```
let result = 5 & 3; // result is 1 (binary: 101 & 011)
```

2. Bitwise OR (|): Performs a bitwise OR operation.

```
let result = 5 | 3; // result is 7 (binary: 101 | 011)
```

3. Bitwise XOR (^): Performs a bitwise XOR (exclusive OR) operation.

```
let result = 5 ^ 3; // result is 6 (binary: 101 ^ 011)
```

4. Bitwise NOT (~): Flips the bits of its operand.

```
let result = ~5; // result is -6 (bitwise NOT of 5)
```

5. Left Shift (<<): Shifts the bits of the left operand to the left by the number of positions specified by the right operand.

```
let result = 5 << 1; // result is 10 (binary: 101 << 1)
```

6. Right Shift (>>): Shifts the bits of the left operand to the right by the number of positions specified by the right operand.

```
let result = 5 >> 1; // result is 2 (binary: 101 >> 1)
```

LOOPS

IF-ELSE LOOP

The if-else statement is used for conditional execution. It allows you to execute a block of code if a certain condition is true and another block of code if the condition is false.

Syntax:

```
if (condition) {

// Code to be executed if the condition is true

} else {

// Code to be executed if the condition is false

}
```

Example:

```
let age: number = 20;
```

```
if (age >= 18) {

console.log('You are an adult.');

} else {

console.log('You are a minor.');

}
```

In this example, if the age is greater than or equal to 18, it prints "You are an adult." Otherwise, it prints

"You are a minor."

FOR LOOP

The for loop is used for iterating over a block of code multiple times. It consists of an initialization, a condition, and an iteration statement. The loop continues to execute the block of code as long as the condition is true.

Syntax:

```
for (initialization; condition; iteration) {

// Code to be executed in each iteration

}
```

Example:

```
for (let i = 0; i < 5; i++) {

console.log('Iteration:', i);

}
```

In this example, the loop initializes i to 0, executes the block of code as long as i is less than 5, and increments i in each iteration. It prints the message "Iteration: 0" to "Iteration: 4".

WHILE and DO- WHILE LOOP

while Loop:

```
let i = 0;
while (i < 10) {
console.log('Iteration:', i);
i++;
}
```

The loop initializes i to 0, executes the block of code as long as i is less than 10, and increments i in each iteration. It prints the message "Iteration: 0" to "Iteration: 10".

do-while Loop:

```
let i = 0;
do {
```

```
console.log('Iteration:', i);

i++;

} while (i < 2);
```

OUTPUT:

```
Iteration: 0

Iteration: 1
```

Here's the breakdown of how the code works:

- The loop initializes i to 0.
- It enters the loop and prints the current value of i.
- The loop increments i by 1.

- It checks the condition (i < 2). If the condition is true, it repeats the loop; otherwise, it exits the loop.

- The loop will iterate five times because it starts with i = 0 and increments i in each iteration, and the loop condition (i < 2) will eventually become false after the second iteration.

PRACTICE QUESTIONS:

OPERATORS---

Arithmetic Operations:

- Declare two numbers and check their sum, difference, product, and quotient. Comparison Operations:

 Compare two numbers and check whether the first number is greater than, equal to, or less than the second number.

Logical Operations:

 Declare two numbers and check if a number is less than 20 or greater than 40. Assignment Operations:

 Create a variable that uses the += operator to update a variable by adding 5 to its current value.

Ternary Operator:

 Declare two numbers and check if a number is positive or negative using the ternary operator. Bitwise Operators:

 Declare two numbers and perform bitwise AND, OR, and XOR operations on two numbers check the results.

IF_ELSE---

- Take a number and check whether it’s positive, negative or zero
- Declare 3 variables and check which one is the largest
- Declare a variable and check even or odd number

FOR LOOPS—

- Print numbers from 1 to 10
- Print the even or odd numbers from 1 to 20
- Calculate the sum of 5 digits using for loop

WHILE LOOPS—

- Using while loop check the count from 1 to 5

- Get the Sum of first 10 digits

***************Stay Consistent**************

6.Lazy Loading

Lazy Loading Routing:

Lazy loading is a technique used in web development to optimize the loading time of a web page by deferring the loading of certain resources until they are actually needed. In the context of routing in a web application, lazy loading refers to loading specific modules or components only when the user navigates to a particular route, rather than loading all modules and components at the initial page load.

Lazy loading routing is commonly used in Single Page Applications (SPAs) where the entire application is loaded initially, but certain parts of the application are loaded on-demand as the user interacts with the application.

Without Lazy Loading:

In a traditional setup, both the Home and About components would be loaded when the application starts, increasing the initial load time.

```
// Initialize the application

const app = new App({

routes: [

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

{ path: '/about-us', component: AboutUsComponent },
],
```

```
});
```

With Lazy Loading:

Lazy loading involves importing the component only when it's needed. In a lazy-loaded setup, the About Us component would be loaded only when the user navigates to the '/about-us' route.

When lazy loading a component, you typically use dynamic imports. Dynamic imports return a Promise, and the component is loaded when the Promise is resolved.

Note: If you see import() or something similar in your code, it's a good indication that lazy loading is being used.

```
{
```

```
path: 'about-us',

loadChildren: () => import('./features/about-us/about-us.module').then(m

=> m.AboutUsModule)

}
```

path: 'about-us':

This specifies the route path. In this case, it's 'about-us'. This means that when the user navigates to the 'about-us' route, the specified module (lazy-loaded) will be loaded and associated with this route.

loadChildren: () => import('./features/about-us/about-us.module').then(m => m.AboutUsModule):

This is the key part that enables lazy loading. Instead of directly importing the module at the time the application loads, it uses the loadChildren property to specify a function that will be called when the module is needed.

import('./features/about-us/about-us.module') is a dynamic import statement that returns a Promise. The specified module is not loaded immediately; it will be loaded asynchronously when the Promise is resolved.

The .then(m => m.AboutUsModule) part ensures that the module is loaded and the specific module class (AboutUsModule) is retrieved. This is essential because Angular expects a dynamically loaded module to have a certain structure, and this structure is defined by the module class.

In summary, when a user navigates to the 'about-us' route, the associated module (AboutUsModule) will be loaded asynchronously, reducing the initial bundle size and improving the application's loading performance. Lazy loading is especially useful in large applications were loading all modules at once would lead to slower initial load times.

Advantages of using Lazy Loading Routing

1. Reduced Initial Bundle Size

In a traditional Angular application, all modules are loaded at the initial startup, which can result in a larger bundle size.

2. Faster Initial Page Load

By loading only, the required modules on demand, the initial page load time is significantly improved. Users see the main part of the application quickly and only additional features are loaded as they navigate to specific routes.

3. Improved User Experience

Users experience faster load times when accessing your application, leading to a better overall user experience.

4. Code Splitting

Lazy loading enables code splitting, where different parts of your application are divided into separate chunks. These chunks are loaded dynamically as needed, allowing for better optimization and utilization of browser caching.

5. Route-Level Loading

The loadChildren property is used at the route level, specifying which module should be loaded when a particular route is accessed.

CLASS

A class is a fundamental building block that encapsulates data and behavior into a single unit. It serves as a blueprint or a template for creating objects.

Encapsulation is the bundling of data (attributes or properties) and methods that operate on the data into a single unit (i.e., a class). It helps in hiding the internal details of the class and exposing only what is necessary.

```
class Animal {
constructor(name) {
this.name = name;
}
makeSound() {
// Abstract method, to be implemented by subclasses
}
}
```

In this example, Animal is an abstract class. It defines a property name and an abstract method makeSound. Concrete subclasses (e.g., Dog, Cat) will implement the makeSound method with specific behavior.

OBJECTS

Object is a self-contained unit that consists of both data (often referred to as attributes or properties) and the methods (functions) that operate on that data. Objects are instances of classes and are created based on the blueprint provided by the class.

e.g. the Student class encapsulates the properties (name, age, grade) and behavior (displayInfo) into a single unit.

Reusability:

If you want to represent another student, you can create a new instance of the Student class with different values. The same class can be reused for different students.

Without Object

```
let studentName = "Alice";
let studentAge = 20;

let studentGrade = "A";

displayStudentInfo(name, age, grade)

{

console.log(`Student: ${name}, Age: ${age}, Grade: ${grade}`);

}

displayStudentInfo(studentName, studentAge, studentGrade);
```

In this non-object-oriented approach, you have separate variables for different pieces of information about a student, and a function takes these variables as parameters to display the information.

With Object

Now, let's use objects to represent a student. We'll create a Student object that encapsulates the properties and behavior related to a student.

```
class Student {
constructor(name, age, grade) {
this.name = name;
this.age = age;
this.grade = grade;
}
displayInfo() {
console.log(`Student: ${this.name}, Age: ${this.age}, Grade:
${this.grade}`);
}

}

const alice = new Student("Alice", 20, "A");

alice.displayInfo();

const john = new Student("John", 24, "B");

john.displayInfo();
```

Now, you can easily manage a list of students by creating multiple instances of the Student class and

calling the displayInfo method for each student.

PROPERTIES

Properties are variables declared within a class, defining the data that an object created from the class will hold. Each object created from the class will have its own set of these properties.

These are the attributes or variables associated with an object, representing its state. Properties store information about the object.

```
// Define a simple class
class Person {
constructor(name, age) {
this.name = name;
this.age = age;
}
greet() {
console.log(`Hello, my name is ${this.name} and I am ${this.age} years
old.`);
}

}

// Create an object (instance) of the Person class

const person1 = new Person('John', 25);

// Accessing properties and calling methods

console.log(person1.name); // Output: John

person1.greet(); // Output: Hello, my name is John and I am 25

years old.
```

METHODS

- A method is a block of code associated with an object or a class, which performs a specific action or provides a service.

- They represent the behavior or actions that objects of the class can perform.

- Methods operate on the data (properties) of an object and can interact with other objects.

- Methods can be reused across different parts of the program or even in different programs.

- This enhances code maintainability and reduces redundancy.

Without Parameters and Arguments:

```
getInfo(){

console.log("Information Logged!!")

}
```

Parameters and Arguments:

- Parameters are variables declared in the method's signature.

- Arguments are values passed to the method when it is called.

- Parameters receive the values of the corresponding arguments during method invocation.

```
addNumbers(a, b) {
return a + b;
}

const result = addNumbers(3, 4); // Here, 3 and 4 are arguments passed to
the addNumbers method.
console.log(result); // Output: 7
```

Return Statement:

- A method may return a value using the return statement.

- The returned value can be assigned to a variable or used in expressions.

```
multiply(a, b) {

return a * b;

}

const product = multiply(5, 6);

console.log(product); // Output: 30
```

NAMING CONVENTIONS

1. Method Naming: Use camelCase for method names.

e.g. getUserInfo()

2. Event Handlers: Prefix event handler methods with "on".

e.g. onClick()

3. Variable Naming: Use camelCase for variable names.

e.g. let userName: string = "John";

4. Constants: Use uppercase letters and underscores for constant variables.

e.g. const MAX_COUNT: number = 10;

5. Boolean Variables: Prefix boolean variables with "is" or "has".

e.g. let isActive: boolean = true;

6. Parameter Naming: Use camelCase for parameter names

e.g. calculateSum(number1: number, number2: number): number {

```
// method implementation

}
```

********************Stay Consistent*******************

7.Template Driven Form & Reactive Form:

In template-driven forms, the form structure and behavior are primarily defined in the template (HTML) rather than in the component class (TypeScript). This approach is more declarative and relies on directives in the template to create and manage the form.

Automatic Form Creation:

The form is automatically created based on the structure of the HTML template. Angular inspects the template and generates the necessary form controls and bindings.

ngModel directive:

It’s used for two-way data binding between the form controls and the component properties.

Implicit Validation:

- Template-driven forms provide implicit validation through directives like required, minlength, maxlength, etc. These validators are applied directly in the template.

Less Boilerplate Code:

- Compared to reactive forms, template-driven forms generally result in less boilerplate code in the component class. The form logic is often handled directly in the HTML template.

Event Handling:

- Event handling, such as form submissions, is typically done by calling methods in the component class from the template.

HTML

```
<!-- app.component.html -->

<div class="form-container">

<h2>Template-Driven Form</h2>

<form #userForm="ngForm">

<div>

<label for="name">Name:</label>
```

```
<input type="text" id="name" name="name" ngModel>
<label for="email">Email:</label>
<input type="email" id="email" name="email" ngModel>
<button (click)="onSubmit(userForm)">Submit</button>
</div>
</form>
</div>
```

CSS

```
/* app.component.css */
```

```css
.form-container {
max-width: 400px;
margin: auto;
padding: 20px;
border: 1px solid #ccc;
border-radius: 5px;
box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
}
input {
width: 100%;
padding: 8px;
```

```
box-sizing: border-box;
border: 1px solid #ccc;
border-radius: 4px;
outline: none;
}
button {
background-color: #4caf50;
color: white;
padding: 10px 15px;
border: none;
border-radius: 4px;
```

```
cursor: pointer;

}
```

TS File

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

import { FormsModule } from '@angular/forms';

@Component({

selector: 'app-port-test2',

standalone: true,

imports: [FormsModule],

templateUrl: './port-test2.component.html',

styleUrl: './port-test2.component.css'
})

export class PortTest2Component {

onSubmit(form: any): void {

// Displaying form values in the console for this example

console.log('Form submitted:', form.value);

}

}
```

REACTIVE FORM:

Reactive Forms provide a more programmatic and flexible way to work with forms in Angular. Unlike Template-Driven Forms, Reactive Forms are driven by the underlying form model defined in the component class (TypeScript).

FormBuilder

- It’s a service in Angular that provides syntactic sugar and convenience methods for creating instances of FormGroup and FormControl.

- It simplifies the process of defining and managing form controls, making the code cleaner and more readable.

- It is typically injected into a component through its constructor.

FormGroup

- It’s a class in Angular that represents a group of form controls. It can contain one or more FormControl instances.

FormControlName

- It’s a directive used in the template to bind a form control defined in the FormGroup to an input field.

- It works in conjunction with formGroup to establish a connection between the form control in the component class and the corresponding input field in the template.

Programmatic Form Creation:

- Reactive Forms involve creating and managing form controls and group programmatically in the component class using the FormBuilder service

Reactive Forms introduce three main building blocks

o FormControl: Represents a single input control.

o FormGroup: Represents a group of form controls.

o FormArray: Represents an array of form controls.

- These building blocks provide a more modular and reusable way to structure your forms.

Explicit Validation

- Validation is explicit and defined in the component class using Validators.

Validators are functions that take a form control as an argument and return an error object if the validation fails.

Dynamic Forms

- Reactive Forms are well-suited for dynamic forms where the form structure and validation logic can change based on user interaction or external factors.

HTML

```
<div class="form-container">

<h2>Reactive Form</h2>
```

```
<form [formGroup]="userForm">
<div>
<label for="name">Name:</label>
<input type="text" id="name" formControlName="name">
<div *ngIf="userForm.get('name')?.hasError('required')
&&userForm.get('name').touched">Name is required</div>
<label for="email">Email:</label>
<input type="email" id="email" formControlName="email">
<div *ngIf="userForm.get('email')?.hasError('email') &&
userForm.get('email')?.touched">
Please enter a valid email address.
```

```
</div>

<button (click)="onSubmit()">Submit</button>

</div>

</form>

</div>
```

CSS

```
/* app.component.css */

.form-container {

max-width: 400px;

margin: auto;
```

```
padding: 20px;
border: 1px solid #ccc;
border-radius: 5px;
box-shadow: 0 2px 5px rgba(0, 0, 0, 0.1);
}
input {
width: 100%;
padding: 8px;
box-sizing: border-box;

border: 1px solid #ccc;
```

```
border-radius: 4px;
outline: none;
}
button {
background-color: #4caf50;
color: white;
padding: 10px 15px;
border: none;
border-radius: 4px;
cursor: pointer;
}
```

TS File

```
import { CommonModule } from '@angular/common';

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

import { FormGroup, FormBuilder, Validators, FormsModule,

ReactiveFormsModule } from '@angular/forms';

import { log } from 'console';

@Component({

selector: 'app-about-us',

standalone: true,

imports: [FormsModule,CommonModule,ReactiveFormsModule],
templateUrl: './about-us.component.html',
styleUrl: './about-us.component.css'
})
export class AboutUsComponent {
userForm: any;
constructor(private fb: FormBuilder) {}

ngOnInit(){
this.userForm = this.fb.group({
name: [", [Validators.required]],
email: [", [Validators.required, Validators.email]],
});

}



onSubmit() {

if (this.userForm.valid) {

console.log('Form submitted:', this.userForm.value);

}

}

}
```

ngOnInIt

ngOnInit is a lifecycle hook in Angular that is called after Angular has initialized all the data-bound properties of a directive. It's one of the Angular lifecycle hooks that allows you to perform actions when a component is being initialized.

Lifecycle Hooks

Angular components go through a series of lifecycle stages, and Angular provides hooks at specific points in this lifecycle where you can execute custom logic. These hooks are methods that are called at certain points during the creation, update, and destruction of a component. ngOnInit is one such hook.

Initialization Phase

ngOnInit is specifically related to the initialization phase of a component. After Angular has created a component and set its input properties, it calls ngOnInit. This is a good place to perform any initialization logic for your component.

Use Cases for ngOnInit

- Initializing Properties: You can use ngOnInit to set default values for properties.
- Service Calls: If you need to fetch data from a service when the component is created, you can make the service call in ngOnInit.
- Component Setup: Any setup or configuration that needs to happen once when the component is initialized can be done in ngOnInit.

***************Stay Consistent**************

8.Data Binding

DATA BINDING

Data binding is a concept in software development that establishes a connection between the application's user interface (UI) and the underlying data.

It simplifies the process of updating the UI when the data changes and vice versa. There are two main types of data binding: one-way data binding and two-way

data binding.

One-way Data Binding:

- In one-way data binding, the data flows in a single direction, from the data source to the UI or vice versa.

- Changes in the data source are reflected in the UI, but changes in the UI do not affect the data source.

- This is commonly used when you want to display data in the UI but don't necessarily need to update the data based on user interactions.

```
<h2>{{message}}</h2>
```

Two-way Data Binding:

- In two-way data binding, changes in the UI automatically update the data source, and changes in the data source automatically update the UI.

- This is useful in scenarios where you want to keep the UI and data in sync bidirectionally, such as form input fields.

In Template-Driven Forms:

```
<form #userForm="ngForm">

<label for="name">Name:</label>

<input type="text" id="name" name="name" ngModel required>

<label for="email">Email:</label>

<input type="email" id="email" name="email" ngModel required>
```

```
<button (click)="onSubmit(userForm)">Submit</button>

</form>
```

Two-way Binding using ngModel:

- The ngModel directive is used for two-way data binding.
- The [(ngModel)] syntax is applied to the <input> fields for "name" and "email."
- This establishes a connection between the UI and the corresponding properties in the form model.

In Reactive Forms

```
<div class="form-container">

<h2>Reactive Form</h2>
```

```
<form [formGroup]="userForm">

<div>



<label for="name">Name:</label>

<input type="text" id="name" formControlName="name">

<label for="email">Email:</label>

<input type="email" id="email" formControlName="email">

<button (click)="onSubmit()">Submit</button>

</div>

</form>

</div>
```

Two-way Binding using formControlName:

The <input> fields are bound to specific form controls using formControlName. This establishes a two-way binding between the input fields and the corresponding properties in the userForm FormGroup.

Templates and Directives:

Templates

Definition: A template in Angular is a declarative way of defining the structure of the UI (User Interface) for a component. It is an HTML file that includes Angular-specific syntax and expressions.

Purpose: Templates allow you to define the layout and structure of your application's UI. They include placeholders for data binding, which allows you to display dynamic content and respond to user interactions.

Example:

```
<!-- Example of an Angular template -->
```

```
<h1>{{ title }}</h1>

<p>{{ description }}</p>

<button (click)="handleClick()">Click me</button>
```

Directives

Definition: Directives in Angular are markers on a DOM (Document Object Model) element that tell Angular to attach a specific behavior to that element or transform it in a certain way.

Purpose: They are a way to extend, transform, and manipulate the DOM and its behavior within Angular applications. Directives can be classified into two main types: structural directives and attribute directives.

Structural Directives

Structural directives are responsible for manipulating the structure of the DOM by adding, removing, or replacing elements. They are prefixed with an asterisk (*) in the template syntax.

Angular provides built-in directives, such as ngIf, ngFor, and ngSwitch, which are used for conditional rendering, looping through arrays, and switching between multiple views, respectively.

Additionally, you can also create custom directives to encapsulate and reuse behavior across components.

Host elements

A host element refers to the element to which a directive is applied. When you use a directive in an Angular template, you apply it to a specific element, and that element is considered the host element for the directive.

In the below example for *ngIf, The <div> element is the host element for this directive because that's where the directive is applied. The directive might manipulate the behavior or appearance of this <div> element based on its implementation.

Here are a few commonly used structural directives in Angular:

1. ngIf

Usage: *ngIf="condition"

Description: Conditionally adds or removes the host element and its children from the DOM based on the specified condition.

```
<!-- Example of using *ngIf directive -->
```

HTML--

```
<div *ngIf="isUserLoggedIn">
<p>Welcome, {{ username }}!</p>
</div>
```

TS--

```
isUserLoggedIn:boolean=true;
```

2. ngFor:

Usage: *ngFor="let item of items"

Description: Iterates over a collection (e.g., array) and repeats the host element for each item in the collection. It is used for rendering lists of items dynamically.

<!-- Example of using *ngFor directive –>

HTML--

```
<ul>

<li *ngFor="let item of items">{{ item }}</li>

</ul>
```

TS–

```
items: string[] = ['Item 1', 'Item 2', 'Item 3'];
```

3. ngSwitch

Usage: <div [ngSwitch]="expression"> ... </div>

Description: Conditionally renders content based on the value of an expression. It's similar to a switch statement in programming languages.

```
<!-- Example of using *ngSwitch directive -->

HTML--

<div [ngSwitch]="userRole">

<div *ngSwitchCase="'admin'">

<p>Welcome, Admin!</p>

</div>

<div *ngSwitchCase="'user'">

<p>Welcome, User!</p>
```

```
</div>

<div *ngSwitchDefault>

<p>Welcome, Guest!</p>

</div>

</div>
```

TS--

```
userRole: string = 'guest';
```

Event Handling

Event handling in Angular involves responding to user interactions, such as clicks, key presses, or form submissions.

```
<!-- Example of using Event Handling -->
```

HTML--

```
<button (click)="onSubmit()">Submit</button>
```

TS--

```
onSubmit() {

console.log("Form Submitted",this.userForm.value);

}
```

Explaination:--

(click) is an Angular event binding syntax. It associates the onSubmit() method from the component class with the click event of the "Submit" button.

***************Stay Consistent**************

9.Component Lifecycle Hooks & Pipes:

These are lifecycle hooks in Angular, which are methods that Angular calls at specific points in the lifecycle of a component or directive.

Let's go through each one with a brief explanation,

First, here's the sequence of lifecycle hooks when a component is created:

- Constructor - the constructor of the component class is called first

1. ngOnChanges

If the component has input/output bindings, ngOnChanges is called next.

This hook is called before ngOnInit and provides information about the changes in the component's input/output property when the binding value changes.

```
ngOnChanges(changes: SimpleChanges) {

// React to input binding changes

if (changes['myInput']) {

console.log('Input changed:',

changes['myInput'].currentValue);

}
}
```

2. ngOnInit

After ngOnChanges, the ngOnInit lifecycle hook is called.

This is called once after the component is initialized. Ideal for initializing component properties.

```
ngOnInit() {

// Initialization logic after ngOnChanges

console.log('Component initialized');

}
```

3. ngDoCheck

This hook is called after ngOnInit.

It provides an opportunity for the developer to implement custom change detection logic. Called during every change detection cycle.

```
ngDoCheck() {

// Custom change detection logic

console.log('Custom change detection');

}
```

4. ngAfterContentInit

Called after the component's content has been initialized.

This hook is often used when you need to perform initialization logic related to content projection.

```
ngAfterContentInit() {
```

```
// Initialization logic after content is initialized

console.log('Content initialized');

}
```

5. ngAfterContentChecked

It is called after every check of the component's content.

It provides an opportunity to perform logic after the content has been checked during the change detection cycle.

```
ngAfterContentChecked() {

// Logic after checking component content

console.log('Content checked');
}
```

6. ngAfterViewInit

Called once after the component's view and child views have been initialized.

Useful for performing operations that require the view to be ready.

```
ngAfterViewInit() {

// Initialization logic after views are initialized

console.log('Views initialized');

}
```

7. ngAfterViewChecked

It is called after every check of the views of the component. It allows you to perform logic after the views have been checked during the change detection cycle.

```
ngAfterViewChecked() {

// Logic after checking component views

console.log('Views checked');

}
```

8. ngOnDestroy

This hook is called just before the directive (or component) is destroyed.

It provides an opportunity to perform cleanup logic, such as unsubscribing from observables or releasing resources.

The ngOnDestroy lifecycle hook won't be automatically called for a component unless it is actually destroyed. Components in Angular are typically destroyed when their associated views are removed from the DOM or when their parent component is destroyed.

Here are a few scenarios where the ngOnDestroy hook is typically called:

Routing Navigation:

If the component is associated with a route, it will be destroyed when the user navigates away from that route.

*ngIf or ngIf Directives

If the component is conditionally rendered using *ngIf or ngIf directive, it will be destroyed when the condition becomes false.

```
ngOnDestroy() {

// Cleanup logic before component destruction
console.log('Component destroyed');

}
```

By logging messages in each of these hooks, you can observe the sequence of calls and understand when each hook is executed during the component lifecycle. This can be very helpful for debugging and optimizing your components.

Output:----

```
Component initialized

Custom change detection

Content initialized

Content checked

Views initialized

Views checked

Custom change detection
```

```
Content checked

Views checked

Custom change detection

Content checked

Views checked
```

The repeated logs of "Custom change detection," "Content checked," and "Views checked" indicate that the change detection cycle is being triggered multiple times, which is normal behavior in Angular.

PIPES

In Angular, pipes are a way to transform data in the template before displaying it.

They are similar to filters in other frameworks and can be used for various tasks, such as formatting dates, converting text to uppercase, or filtering lists. Angular provides several built-in pipes, and you can also create custom pipes for your specific needs. Here are a few examples of built-in Angular pipes,

1. Uppercase and Lowercase: The uppercase and lowercase pipes transform the input string to uppercase or lowercase, respectively.

```
<!-- Convert text to uppercase -->

<p>{{ 'hello world' | uppercase }}</p>

<!-- Convert text to lowercase -->

<p>{{ 'HELLO WORLD' | lowercase }}</p>
```

Output:

```
HELLO WORLD
```

hello world

2. Date Pipe: The date pipe formats a date based on the provided format. In this example,

the 'short' format is used

```
<!-- Format a date -->

<p>{{ today | date:'short' }}</p>
```

Output: (Assuming today is a Date object)

```
MM/DD/YYYY
```

3. Percent Pipe: The percent pipe multiplies the input by 100 and appends a percentage sign.

```
<!-- Format a number as a percentage -->
```

```
<p>{{ 0.25 | percent }}</p>
```

Output:

```
25%
```

4. Json Pipe: The json pipe converts an object to its JSON string representation. It's useful for debugging or displaying structured data.

```
<!-- Display an object as JSON string -->

<p>{{ object | json }}</p>
```

Useful for debugging or displaying structured data.

Output:

```
{"key": "value", "anotherKey": 42}
```

5.)Keyvalue Pipe: The keyvalue pipe iterates over the key-value pairs of an object,

allowing you to display them in a template.

```
<!-- Iterate over key-value pairs of an object -->

<div *ngFor="let entry of myObject | keyvalue">

{{ entry.key }}: {{ entry.value }}

</div>
```

Output:

```
key: value

anotherKey: 42
```

6. Slice Pipe: The slice pipe extracts a portion of a string or array. In this example, it extracts characters from index 0 to 6.

```
<!-- Slice a string or array -->

<p>{{ 'Angular is awesome' | slice:0:7 }}</p>
```

Output:

```
Angular
```

7. Title Case Pipe: The titlecase pipe converts the input string to title case (capitalizing the first letter of each word).

```
<!-- Convert a string to title case –>

<p>{{ 'hello world' | titlecase }}</p>
```

Output:

Hello World

******************Stay Consistent*****************

10.Event Binding

Definition:-

Event binding is a mechanism in Angular that allows you to capture and respond to events triggered by user interactions, such as clicks, keystrokes, mouse movements, etc. It provides a way to execute custom code when certain events occur in the application.

Syntax: In Angular, event binding is accomplished by using a set of parentheses () in the template to bind a method or expression to a specific event.

Eg:-

HTML-

```
<button type="submit" (click)="submit()">Submit</button>
```

TS-

```
submit() {

console.log("SUBMIITEDDDDDDD");

}
```

In this example, the (click) event binding is used to bind the submit() method to the click event of the button. When the button is clicked, the submit() method is executed, logging "SUBMITTED" to the console.

EVENT FILTERING:

Definition: Event filtering involves selectively handling or filtering events based on certain conditions within the event handler. It allows you to evaluate conditions before allowing the default behavior associated with the event to take place.

For eg:- you might want to perform an action only if certain keys are pressed, a specific element is targeted, or some other criteria are met.

Event filtering typically involves adding conditions inside the function that handles the event to determine whether to proceed with the default action or not based on certain criteria.

Eg:-

HTML–

```
<button type="submit" (click)="handleSubmit($event)">Submit</button>
```

TS–

```
handleSubmit(event: any) {

console.log("EVENTTTTTTTT", event);

console.log("EVENTTTTTTTT", event.altKey);
if (event.altKey) {
console.log("ALTTT key is pressed");
} else {
console.log("Normal");
}
}
```

Explanation:

- The ($event) syntax in the template is used to pass the event object to the corresponding method in the component.

- The event object contains information about the event, such as which key was pressed, the type of event, etc.

- Inside the event handler method (handleSubmit), you can access properties of the event object to perform conditional actions based on certain criteria.

- In this case, it checks whether the Alt key is pressed and logs different messages accordingly. In this example, the (click) event is bound to the handleSubmit($event) method. The $event is a special variable that captures the event object. Inside the handleSubmit method, it checks if the Alt key is pressed

(event.altKey). If the Alt key is pressed, it logs "ALT key is pressed," otherwise, it logs "Normal."

********************Stay Consistent*******************

11.OnChange

Definition: The (change) event is triggered when the value of an input element, such as a dropdown (select) or an input field (input), changes. It typically occurs after the user has selected a different option in a dropdown or entered text into an input field and then moved the focus away from that element.

Eg:-

HTML-

```
<label for="gender" class="asterisk">Gender</label>

<select id="gender" formControlName="gender"

placeholder="Please select Gender" (change)="onGenderChange($event)">

<option value="" disabled selected>Select Gender</option>

<option value="male">Male</option>

<option value="female">Female</option>

<option value="other">Other</option>

</select>
```

TS-

```
selectedGender: any;

onGenderChange(event: any) {

console.log("onGenderChange", event);

console.log("onGenderChange target", event.target);

console.log("onGenderChange value", event.target.value);

this.selectedGender = event.target.value

}
```

Explanation: The (change) event is attached to the select element, and it triggers the onGenderChange method when the user selects a different option.

Now, then selectedGender is a property in the component that is used to store the selected gender.

The onGenderChange method is called when the (change) event occurs. Now, let’s see how this onChange works here:–

- event is the event object itself.

- event.target refers to the DOM element that triggered the event, in this case, the select element.

- event.target.value retrieves the selected value from the select element, which corresponds to the chosen gender.

Finally, the selected gender value is assigned to the selectedGender property.

SUBSCRIBE VALUE CHANGES

Definition:

valueChanges:--

- In Angular's Reactive Forms, each form control has a property called valueChanges.

- valueChanges is an observable that emits an event every time the value of the associated form control changes.

- It provides a way to observe and react to changes in real-time.

subscribe:--

- The subscribe method is part of the Observable pattern in JavaScript and is used to listen to events emitted by an observable.

- In the context of Angular's Reactive Forms, when you call subscribe on the valueChanges

observable of a form control, you're essentially saying, "I want to be notified whenever the value of this form control changes."

Eg:-

HTML–

```
<label for="gender" class="asterisk">Gender</label>

<select id="gender" formControlName="gender"

placeholder="Please select Gender">
```

```
<option value="" disabled selected>Select Gender</option>

<option value="male">Male</option>

<option value="female">Female</option>

<option value="other">Other</option>

</select>
```

TS–

```
ngOnInit() {

this.medicalForm.get('gender').valueChanges.subscribe((value:string)=>

{

this.selectedGender=value;

console.log("selectedGender",this.selectedGender);
})

}
```

Explanation:

this.medicalForm.get('gender'):

- medicalForm is an instance of the Angular FormGroup class, representing a form in the component.
- .get('gender') is used to retrieve the form control named 'gender' from the medicalForm.

valueChanges.subscribe(callback):

- valueChanges returns an observable that you can subscribe to.
- When you subscribe to valueChanges, you provide a callback function (callback) that will be executed whenever the value of the form control changes.

- The callback function receives the new value as a parameter, allowing you to perform actions based on the updated value.

.subscribe((value: string) => {...}):

- .subscribe is used to subscribe to the changes emitted by the valueChanges observable.

- The callback function (value: string) => {...} is executed whenever the value changes.

- value is the new value of the 'gender' form control, and it is expected to be a string.

Inside the callback function, this.selectedGender = value; assigns the new value to the selectedGender property in the component.

ngOnInit method sets up a subscription to changes in the 'gender' form control using the valueChanges observable. Whenever the value of the 'gender' control changes, the callback function is executed, updating the selectedGender property and logging the new value to the console.

****************Stay Consistent***************

12.TimeOuts

Definition:

Timeouts in Angular refer to a mechanism where a function or a piece of code is scheduled to run after a specified amount of time has passed. It allows you to delay the execution of a specific task.

Eg:-

TS-

```
constructor(){

this.startTimeout();

}
```

```
startTimeout(){

setTimeout(() =>{

console.log("Timeeoutttt completeeddd!!!!");

},5000);

}
```

Explanation:

In the example, the startTimeout method is called in the constructor, and it uses setTimeout to log a message after a delay of 5000 milliseconds (5 seconds). This is useful for scenarios where you want to execute code after a certain delay.

INTERVALS

Definition:

Intervals involve repeatedly executing a function or a block of code at specified time intervals. It is similar to timeouts but differs in that it continues executing the provided function at regular intervals until explicitly stopped.

Eg:-

TS–

```
constructor(){

this.startInterval();

}

startInterval(){

setInterval(()=>{

console.log("Set intervall completed");

},2000);

}
```

Explanation:

- The startInterval method uses setInterval to repeatedly execute the provided function (logging a message) every 2000 milliseconds (2 seconds).

- The function provided to setInterval will be invoked at regular intervals until the application is closed or the interval is explicitly cleared.

To stop the interval and exit the loop, you need to use the clearInterval function, passing in the interval ID returned by setInterval. Here's how you can modify your code to include a way to stop the interval:

TS–

```
intervalId: any; // Store the interval ID

constructor() {

this.startInterval();

}
```

```
startInterval() {

this.intervalId = setInterval(() => {

console.log("Set interval completed");

}, 2000);

}

stopInterval() {

clearInterval(this.intervalId); // Clear the interval using the stored

ID

console.log("Interval stopped");

}
```

Explanation:

In this example, I've added a variable intervalId to store the ID returned by setInterval. The stopInterval method can then be called to clear the interval and stop the execution of the provided function. Now, whenever you want to exit the loop, you can call the stopInterval method. For example, you might

call it based on some condition or user action.

CHANGE DETECTOR REF

Definition:

ChangeDetectorRef is an Angular service that allows you to manually trigger change detection. When you update data outside Angular's normal lifecycle (e.g., through third-party libraries or asynchronous operations), Angular might not be aware of the changes. Calling detectChanges ensures that Angular

checks for changes and updates the view accordingly.

Eg:-

HTML–

```
<div>{{message}}</div>

<button (click)="updateMessage()">Update</button>
```

TS–

```
updateMessage(){

this.message="Updated message"

this.cdr.detectChanges();

}
```

Explanation:

- The HTML template displays a message using the Angular interpolation syntax ({{message}}) and provides a button triggering the updateMessage method.

- The updateMessage method updates the value of message and then calls cdr.detectChanges().

It's important to note that while ChangeDetectorRef provides a way to manually trigger change detection,

it's generally recommended to rely on Angular's automatic change detection mechanism as much as

possible, as it is optimized for performance.

************Stay Consistent***********

13.Interfaces & Enums

Definition:

An interfaceis a way to define a contract for the structure of an object. It declares a set of properties and their types, but it doesn't provide an implementation. It's used to specify what properties an object must have without actually defining how those properties will be implemented.

By enforcing this structure, you make it clear what properties are expected for a valid Product object, providing type safety and making the code more maintainable.

If you attempt to assign an object to the array of products that does not adhere to the structure defined by the Product interface, TypeScript will raise a compile-time error. TypeScript interfaces are used for static type checking, and they help catch potential issues during the development phase.

Eg:-

TS-

 Outside the class:--

```
interface Product {

id: number;

name: string;

price: number;

quantity: number;

}
```

▯ Inside the class:–

```
product: Product[] = [
{
id: 1,
name: 'ABC',
price: 5,
quantity: 10,
},
]
```

Explanation:

The Product interface defines a contract for objects that should have properties like id, name, price, and quantity, each with a specific data type.

Inside your Component class, an array of products is declared using the Product interface. This ensures that each product in the array adheres to the structure defined by the interface.

If an object doesn't match the structure defined in the interface, TypeScript will raise a compilation error.

ENUMS

Definition:

Enums, short for enumerations, are a feature in many programming languages, including TypeScript. In TypeScript, enums allow you to define a set of named constant values. These values can then be used in your code to represent a finite set of distinct options or states. Each of these members is assigned an automatic numeric value starting from 0. Enums are often used when you have a fixed set of values that a variable can take. For example, you might use an enum to represent the possible directions in a game, the days of the week, or the status of an operation.

Eg:-

TS–

- Outside the class:--

```
enum Direction{
Up,
Down,
Left,
Right
}
```

- Inside the class:–

```
enums(){
let direction:Direction;
direction=Direction.Left;
console.log("direction",direction);

}
```

Explanation:

The Direction enum is defined with four values: Up, Down, Left, and Right. By default, each value is assigned a numeric index starting from 0.

Suppose ,if you declare enum Direction with Up = 5, it sets Up to 5, and subsequent enum members are assigned incremental values. Down will be 6, Left will be 7, and Right will be 8. Inside your Component class, a variable direction of type Direction is declared and assigned the value Direction.Left. Enums help make code more readable by providing meaningful names for numeric values.

By defining interfaces and enums separately, you make them reusable across multiple classes.

*********Stay Consistent********

14.Angular Animations:

For using this firstly enter the command in your command prompt

```
npm i @angular/animations
```

Definition: Angular animations are a feature in the Angular framework that allows you to create dynamic and visually appealing effects in your web applications. These animations are built on top of the Web Animations API and provide a powerful way to manipulate the presentation of HTML elements during different states or transitions.

Here are the key concepts in Angular animations:

 Animation Trigger

An animation is defined by an animation trigger, which is created using the trigger function. The trigger has a name and contains one or more states, transitions, and associated styles.

Eg:-

```
trigger('myAnimation', [

// states, transitions, and styles

])
```

▯ States

States represent different configurations or appearances of an element. States are defined using the state function and include a set of styles that should be applied when the element is in that state.

Eg:-

```
state('start', style({ opacity: 1, transform: 'scale(1)' }))
```

- Transitions

Transitions define how an element moves from one state to another. They are created using the transition function, specifying the start and end states, as well as the animation that should be applied during the transition.

Eg:-

```
transition('start => end', animate('500ms ease-in'))
```

- Styles

Styles represent the CSS properties and values that should be applied to an element in a particular state. Styles are defined using the style function.

Eg:-

```
style({ backgroundColor: 'red', color: 'white' })
```

Below are some of the types of how angular animations takes place:

Single Animation

This property is used to define animations for the component. It includes an array with a single animation trigger named myAnimation. The trigger specifies two states (start and end) and transitions between them with associated styles and timing.

currentState: This is a variable in the component that keeps track of the current state of the animation. It is initially set to 'start'.

Eg:-

HTML–

```
<div [@myAnimation] = "currentState" (click)="toggleState()">

Click me to toggle Animation

</div>
```

TS-

```
animations:[
trigger('myAnimation',[
state('start',style({
transform:'scale(1)',
backgroundColor:'red'
})),
state('end',style({
transform:'scale(1.5)',
backgroundColor:'blue'
})),
transition('start => end',

animate('500ms ease-in')),

transition('end => start',

animate('500ms ease-out'))

])

]

currentState:string='start';

toggleState(){

this.currentState = this.currentState === 'start'?'end' :'start';

}
```

Explanation

[@myAnimation]="currentState": This is an Angular animation binding. It binds the myAnimation animation trigger to the current state (currentState variable) of the component. As the state changes, the associated animation will be triggered.

Group Animation

This property is used to define animations for the component. It includes an array with a single animation trigger named combinedAnimations. The trigger specifies two states ('start' and 'end') and transitions between them with a group of two animations for each transition.

currentState: This is a variable in the component that keeps track of the current state of the animation. It is initially set to 'start'.

Eg:-

HTML –

```
<div [@combinedAnimations]="currentState"></div>
```

```
<button (click)="toggleAnimation()">Toggle Button</button>
```

TS–

```
animations:[
trigger('combinedAnimations',[
state('start',style({
opacity:1,
transform:'translateX(0)'
})),
state('end',style({
opacity:0,
transform:'translateX(100px)'
})),
transition('start => end',group([
animate('500ms ease-out', style({ transform:
'scale(1.5)', backgroundColor: 'blue' })), // First animation
animate('300ms 200ms ease-in', style({ opacity: 0 })), // Second
animation with delay
])),
transition('end => start', group([
animate('500ms ease-out', style({ transform: 'translateX(0)' })),
// Reverse the first animation
animate('300ms ease-in', style({ opacity: 1 })), // Reverse the
second animation
])),
])

]

currentState:string='start';

toggleAnimation(){

this.currentState = this.currentState === 'start'?'end' :'start';

}
```

Explanation

[@combinedAnimations]="currentState": This is an Angular animation binding. It binds the combinedAnimations animation trigger to the current state (currentState variable) of the component. As the state changes, the associated combined animation will be triggered.

*********Stay Consistent********

15.Error handeling:

Error handling is crucial to manage and respond to unexpected issues that may occur during the execution of a program.By handling errors, you can ensure that your application doesn't crash or stop unexpectedly when it encounters unexpected situations. Instead, it can degrade gracefully, providing a better user experience. One of the common ways to handle errors is by using the try, catch, and finally blocks.

try block: In this block, you place the code that might throw an exception. The try block is the region of code where you expect an error to occur.

catch block: If an exception is thrown in the try block, the control flow jumps to the corresponding catch

block. The catch block contains the code that will handle the exception. You can define a variable (often named error or err) that will hold information about the thrown error.

finally block (optional): This block is executed regardless of whether an exception occurred or not. It is commonly used for cleanup code that must be executed no matter what.

Eg:-

TS-

```
constructor(){

this.errorhandling()

}

sampleError() {

let x: number = null!;
```

```
console.log("Sampple", x);
throw new Error("")
}
errorhandling() {
try {
let result = this.sampleError();
console.log("Result", result);
} catch (error: any) {
console.error("Type error", error.message)
} finally {
console.log("Finally");
}

}
```

Explanation

- sampleError()

It is a method that is throwing an error intentionally. It declares a variable x with a type annotation of number, but assigns it the value null!. The ! here is the non-null assertion operator, essentially telling TypeScript to treat null as a valid value for x. Then it logs the value of x and throws an empty Error.

- errorhandling() is a method that contains a try-catch-finally block. Here's how it works:

Try Block: It attempts to execute the code inside. It calls this.sampleError() and assigns its result to the result variable.

Catch Block: If an exception is thrown inside the try block, it catches the exception. The catch

block logs an error message to the console, specifically mentioning it as a "Type error." The error

variable in the catch block is of type any, meaning it can be of any type.

Finally Block: This block is always executed, whether an exception is thrown or not. It logs

"Finally" to the console.

In summary, when an instance of this class is created, the constructor calls errorhandling().

errorhandling() in turn calls sampleError() which intentionally throws an error. The catch block in

errorhandling() catches the error, logs a message, and the finally block is executed regardless of whether

an error occurred or not.

METADATA

In Angular, metadata plays a crucial role in defining and configuring various aspects of components, directives, modules, and services. Metadata is essentially additional information that describes how a particular class should be processed or behave in the Angular application.

Angular uses decorators to attach metadata to classes. A decorator is a special kind of declaration that can be attached to a class declaration, method, accessor, property, or parameter. Decorators are denoted by the @ symbol followed by the decorator's name.

Eg:-

```
@Component({

//Metadata ☒

selector: 'app-error-handling',

standalone: true,
imports: [CommonModule],

templateUrl: './error-handling.component.html',

styleUrl: './error-handling.component.scss'

})
```

*********Stay Consistent********

16.Service/DI/Constructor:

SERVICES

- In Angular, services are a fundamental building block of the framework. They are used to encapsulate and provide a specific functionality or set of functionalities that can be shared across multiple components, directives, or other services within an Angular application.

- Services have a lifecycle managed by Angular. When provided at the root level (providedIn: 'root'), a service is a singleton, meaning there is only one instance of the service in the entire application.
- Angular services are typically created as TypeScript classes. You use the @Injectable decorator to mark a class as a service it’s not always required, but it's good practice to use it.
- Services play a crucial role in promoting modularity, code organization, and reusability. Services are used to encapsulate and manage functionality that doesn't belong to a specific component, such as data fetching, business logic, or communication with a backend server.
- Generateservices using the ng generate service service-name command.

DEPENDENCY INJECTION

- Dependency injection in Angular allows you to declare the dependencies of a class (like services) in its constructor, and Angular's dependency injection system will provide instances of those dependencies when the class is instantiated.
- Angular's dependency injection system is used to inject services into the components or other services that need them.When a component or another service requests a dependency (like a service) in its constructor, Angular provides the instance of that dependency.
- This makes the code more modular, testable, and maintainable, as dependencies can be easily swapped or mocked during testing. The @Injectable decorator is an essential part of this mechanism, allowing Angular to understand how to create and manage instances of services.

Eg:-

User.service.ts–

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

@Injectable({

providedIn: 'root'

})

export class UserService {

users:any=[

'user1',

'user2',
'user3'
]
constructor() { }
getData(){
return 'Data from Service'
}
```

```
getUsers(){

return this.users;

}

}
```

HTML–

```
<p>{{data}}</p>

<h2>Users</h2>

<ul>

<li *ngFor="let user of users">

{{users}}
```

```
</li>

</ul>
```

TS-

```
data:any;

users:any;

constructor(public userService:UserService) {

this.data = this.userService.getData();

console.log("Result",this.data);

this.users = this.userService.getUsers();
console.log("Userss---",this.users);

}
```

Explanation:

 Class Properties:

o data: It is a property to store the result of calling the getData() method from the UserService.

o users: It is a property to store the result of calling the getUsers() method from the UserService.

ANGULAR

Lecture 18

3

- Constructor:

o The constructor takes an instance of UserService as a parameter (public userService: UserService), indicating that an instance of the UserService will be injected into this component.

o Inside the constructor, it calls getData() and getUsers() methods from the injected UserService.

o The result of getData() is assigned to the data property, and the result of getUsers() is assigned to the users property.

o Finally it logs the results to the console.

***Stay Consistent**

17.Asynchronous Operations, Async/Await, Promise:

ASYNCHRONOUS OPERATION

 In programming, operations can be broadly categorized as synchronous or asynchronous.

Synchronous operations block the execution of code until the operation is complete, whereas

asynchronous operations allow the program to continue executing while waiting for the operation to finish.

 Asynchronous operations are common in scenarios like fetching data from a server, reading files, or handling user input. In traditional synchronous code, these operations could lead to blocking the entire program, resulting in poor user experience and performance.

ASYNC / AWAITS

 async Function:

o An async function is a function that always returns a promise.

o It can contain the await keyword, signaling that the function will work with promises and asynchronous code.

 await Keyword:

o The await keyword is used inside an async function to pause the execution of the function until the promise is resolved.

o It can be applied to any promise-like object, including promises returned by asynchronous functions or methods.

 Error Handling with try/catch:

o try/catch blocks can be used to handle errors in an async function, providing a structured way to handle both resolved values and errors.

PROMISE

A Promise is an object representing the eventual completion or failure of an asynchronous operation. It is a container for a value, which may be available now, or in the future, or never.

The key characteristics of a Promise are:

- Pending: The initial state of a Promise. The operation represented by the Promise is ongoing, and the final result is not yet known.
- Fulfilled (Resolved): The Promise successfully completed, and a result value is available.
- Rejected: The Promise encountered an error or was unable to fulfill the operation, and an error reason is available.

ANGULAR

Lecture 19

2

Creating a Promise

A Promise is created using the Promise constructor, which takes a single argument, a function called the executor. The executor function is passed two functions: resolve and reject. These functions are used to signal the successful completion or failure of the asynchronous operation.

Eg:-

User.service.ts–

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

@Injectable({
```

```
providedIn: 'root'

})

export class UserService {

fetchData(): Promise<string> {

return new Promise((resolve, reject) => {

const success = true;

setTimeout(() => {

if (success) {

resolve('Data from resolve---')

} else {

reject('Error while fetching---rejected')
}

}, 1000)

})

}

}
```

HTML–

```
<h2>{{this.fetchData}}</h2>
```

TS-

```
async ngOnInit() {
```

```
this.fetchData = await this.userService.fetchData();

console.log("Fetcheddd data-----",this.fetchData);

}
```

Explanation:

- Here, fetchData is is a method that returns a Promise<string>. This function is designed to simulate an asynchronous operation, such as an HTTP request, using setTimeout to introduce a delay of 1000 milliseconds (1 second).

 Inside the Promise constructor, there are two parameters: resolve and reject. resolve is a function used to fulfill the promise with a successful result, and reject is a function used to reject the promise with an error.

 The success variable is used to simulate whether the asynchronous operation was successful. If success is true, the promise is resolved with the string 'Data from Service'. Otherwise, it is rejected with the string 'Error fetching data'.

 async ngOnInit(): This method is marked as async, indicating that it contains asynchronous operations.

o async is used to define a function as asynchronous, allowing the use of await within it.

o await is used to pause the execution of the function until the awaited promise is settled

(resolved or rejected).

***Stay Consistent**

18.Http Requests, RXJS, Observable, Subscribe & Observables:

HTTP REQUESTS

In Angular, when you make HTTP requests, such as fetching data from a server, the response is often

represented as an observable. An observable is an object that emits a sequence of values over time, and it can be used to handle asynchronous operations like HTTP requests. Observables are part of the Reactive Extensions for JavaScript (RxJS) library.

RXJS

RxJS, which stands for Reactive Extensions for JavaScript, is a library that provides a set of tools for reactive programming in JavaScript. It is based on the principles of the ReactiveX library and extends the concept of observables, allowing you to work with asynchronous data streams and events.Here are some key concepts and features of RxJS:

Observables,Operators,Subscription,Error Handling ,etc

OBSERVABLE

In Angular, observables are often used to handle asynchronous operations like HTTP requests or user interactions. Observable is the result of the HTTP request.

- When you make an HTTP request, it returns an observable representing the stream of data that will come from the server.
- You subscribe to this observable to listen for data changes or handle errors.

Imagine a Stream

Observable as a Stream: Think of an observable as a stream of events or data over time. Just like a river that flows continuously, an observable emits vales or events continuously.

Data Over Time

Values Over Time: Imagine you're tracking the temperature every hour. The temperature at each hour is a value in the stream of temperatures over time.

- At 9 AM: 20°C

- At 10 AM: 22°C

- At 11 AM: 25°C

This series of temperature values over time forms a stream, and an observable is like this stream of data.

SUBSCRIBE AND OBSERVERS

Subscribe

ANGULAR

Lecture 20

2

Listening to Changes: Now, imagine you want to know the temperature changes. You subscribe to this stream of temperature data. Whenever the temperature changes, you get notified.

- You subscribe at 9 AM.
- You receive 20°C.
- You subscribe at 10 AM.
- You receive 22°C.

You are "subscribed" to the observable stream, and you get updates whenever there's a new value.

Observer:

The observer is created through the subscribe method. The three functions passed to subscribe represent the observer:

- The subscribe method is used to subscribe to the observable. It takes an object as an argument, following the observer pattern.
- The next property is a function that will be called when the observable emits a new value. In this case, it logs the received data to the console.
- The error property is a function that will be called if there is an error during the observable's execution. It logs the error to the console.
- The complete property is a function that will be called when the observable completes. It logs a message indicating that the request has completed. This function is optional, and you don't have to include it.

Eg:-

User.service.ts–

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

@Injectable({

providedIn: 'root'

})

export class UserService {

observeData() {

let request = of('Dataa from my side') //Return Observable–stream of

data

request.subscribe({

//observer
```

```
next: (data) => {

console.log("Subscribed data---", data);

},

error: (error) => {

console.log("Subscription Error--", error);

},

complete: () => {

//optional

console.log("Request complete");
```

ANGULAR

```
3

},

})

}

}
```

HTML–

```
<h2>{{this.observeData}}</h2>
```

TS-

```
async ngOnInit() {



this.observeData = await this.userService.observeData();

console.log("Observe data-----",this.observeData);

}
```

Explanation

Observable Creation:

 of is an RxJS creation operator used to create an observable that emits a specific value or set of values.

 In this case, of('Data from my side') creates an observable that emits the string 'Data from my side'.

Subscription:

- The subscribe method is used to subscribe to the observable and start listening for emitted values.

- It takes an object (the observer) with three properties: next, error, and complete.

Observer Pattern:

- The observer pattern is employed to handle different aspects of the observable's lifecycle.

- next is a function that gets called when a new value is emitted.

- error is a function that gets called when an error occurs during the observable's execution.

- complete is a function that gets called when the observable completes. It is optional and not always needed.

Result

When you call observeData(), it creates an observable using of('Data from my side'). The observable emits the specified data, and the observer's next function logs the data to the console. If there were any errors during the process, they would be logged in the error function. The complete function (optional) logs a message when the observable completes.

***Stay Consistent**

20.Patch Values & Set Values:

In Angular, "patchValue" and "setValue" are methods used to update form controls within reactive forms. Reactive forms are a way to manage form data using observable streams and immutability. Let's dive into examples for both methods with detailed explanations.

PATCH VALUES

It allows you to update only a subset of form controls within a FormGroup or a FormControl. You provide an object containing the values you want to update.

It's useful when you want to update specific parts of the form without affecting the entire form. If the provided object contains only a subset of the form controls, it will update only those controls, leaving the others unchanged. It's more flexible when you don't need to provide values for all form controls.

Eg:-

```
ngOnInit() {

this.initializeForm();

// Assume this is the data fetched from API

const formData = {
```

```
name: 'John Doe',
age: 30,
gender: 'male',
address: '123 Main St',
diagnosis: 'Headache',
contact: '1234567890',
email: 'john@example.com',
personalHistory: 'None',
familyHistory: 'None',
painScore: '5'
};
```

```
// Use patchValue to update only specific fields
this.medicalForm.patchValue({
name: formData.name,
age: formData.age,
gender: formData.gender,
address: formData.address,
diagnosis: formData.diagnosis,
contact: formData.contact,
email: formData.email
});
}
```

SET VALUES

It's used to update the entire form with new values. You provide an object containing values for all form controls within the FormGroup. It requires a complete set of values for all controls in the form. If any control is not provided a value, it will be reset to its default state. It's useful when you have a complete set of values and want to update the entire form. It enforces that you provide values for all controls, ensuring the form is in a valid state after the update.

Eg:-

```
ngOnInit() {

this.initializeForm();

// Assume this is the data fetched from API

const formData = {

name: 'John Doe',
age: 30,
gender: 'male',
address: '123 Main St',
diagnosis: 'Headache',
contact: '1234567890',
email: 'john@example.com',
personalHistory: 'None',
familyHistory: 'None',
painScore: '5'
};
// Use setValue to update the entire form
```

```
this.medicalForm.setValue(formData);

}
```

Summary

So, the key difference here is that when updating a single form control: patchValue() allows selective updates, while setValue() requires a complete set of values for all controls within the form. Use patchValue() when you want to update specific parts of the form, and use setValue() when you have a complete set of values for the entire form.

Real World Cases:

 User Profile Forms: Users may update specific details such as their email address or phone number without altering other profile information.

 Order Forms: When editing an order, only certain fields like shipping address or payment method may need updating, while other details remain intact.

- Multi-Step Forms: In multi-step forms, each step may update a subset of the form data until the final submission, where all values are set before processing the form.