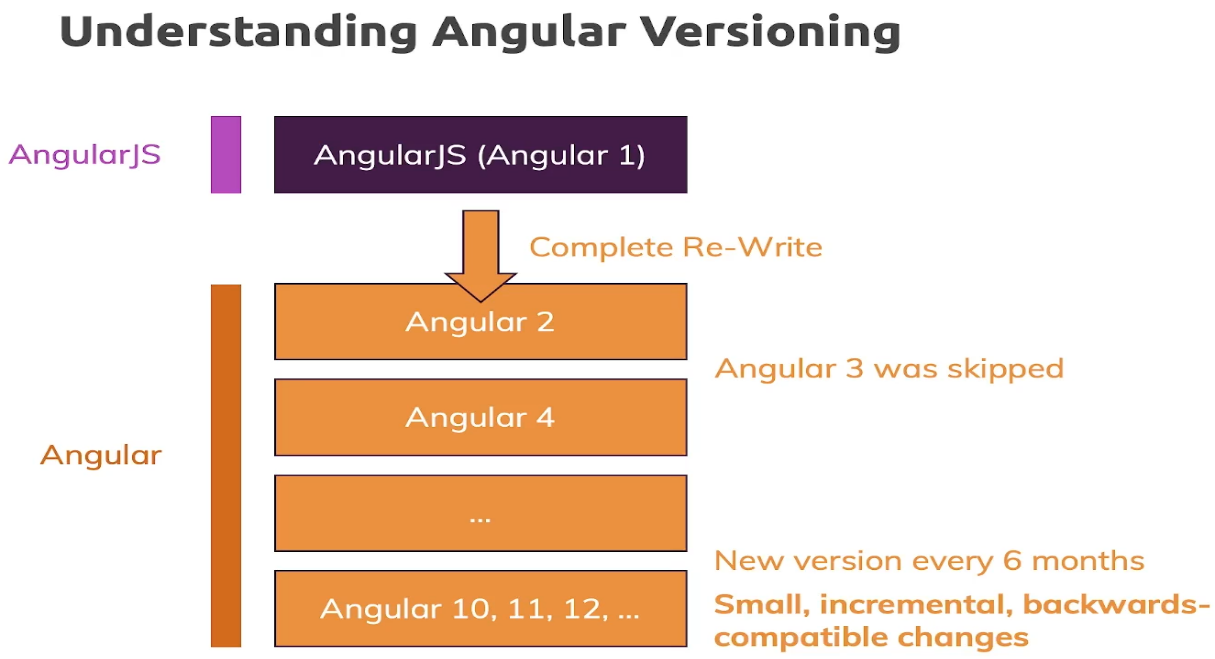
* Angular is a **JavaScript Framework** which allows you to create reactive **Single-Page-Applications** (SPAs).



Commands:

1. Creating angular app

ng new app\_name

1. Generating Component

ng generate component component\_name (or)

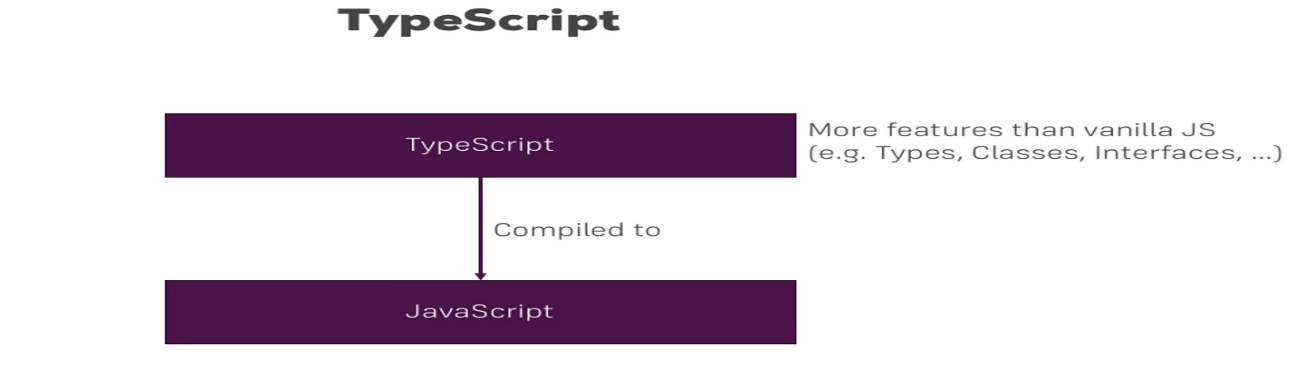
ng g c component\_name

1. To Run Angular app

ng serve

**What is TypeScript?**

* TypeScript is a syntactic superset of JavaScript which adds **static typing**.
* This basically means that TypeScript adds syntax on top of JavaScript, allowing developers to add **types.**
* TypeScript allows specifying the types of data being passed around within the code and can report errors when the types don't match.
* TypeScript uses compile time type checking. Which means it checks if the specified types match before running the code, not while running the code.
* A common way to use TypeScript is to use the official TypeScript compiler, which transpiles TypeScript code into JavaScript.
* TypeScript is transpiled into JavaScript using a compiler.



**How angular app gets loaded and started?**

* Angular is a framework which allows us to create "Single Page Applications", and here the **index.html** is the single page which was provided by the server.

**index.html file**

<!doctype html>

<html lang="en">

<head>

  <meta charset="utf-8">

  <title>FirstApp</title>

  <base href="/">

  <meta name="viewport" content="width=device-width, initial-scale=1">

  <link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

  <app-root></app-root>

</body>

</html>

The above code looks like a normal HTML code and here the <title> tag shows the same title in the browser as the app's title. **But the <body> code is different from normal HTML code**. Here, you see **"<app-root>"** tag which is provided by the CLI. We can say that, whenever we create a project from CLI, by default, one component is created, i.e., "app component".

**app.component.ts file**

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

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  name = 'Max';

}

**“app.component.ts”** is a typescript file. Now in the above code selector has **<app-root>** tag which is same as shown in body of **index.html** file. This information is required the Angular to place this part into an index.html file with the template of the component.

The template of the component is "./app.component.html". So, angular includes this part into the body of the index.html file.

**How does Angular trigger?**

* Whenever ng-serve builds the application, it creates "bundles" and automatically adds these to index.html file at runtime. So, from these bundles, the first code must be executed from **"main.ts"** file.
* **"main.ts"** file is the main file from where the execution of an Angular application starts.

**main.ts file**

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

import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';

import { environment } from './environments/environment';

if (environment.production) {

  enableProdMode();

}

platformBrowserDynamic().bootstrapModule(AppModule)

  .catch(err => console.error(err));

Here, the **bootstrapModule**() method starts the Angular application. It refers to AppModule, which looks into the app folders. You can see in the "app.module.ts" file that a bootstrap array which is basically a list of all the components analyzes the index.html file.

**app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

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

import { AppComponent } from './app.component';

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    FormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

Now, you can see that the Angular application gets loaded as:

**main.ts  -->   app.Module.ts  -->  app.component.ts  -->  index.html  -->  app.component.html**

So Angular gets started, this *main.ts* file gets started, there we bootstrap an Angular application and we pass this **‘App module’** as an argument. In this module, we tell Angular, hey, there is this app component which you know when you try to start yourself and Angular now analyzes the app component, reads the set up we pass here and therefore knows this selector, **app-root** and now Angular is able to handle app-root in the *index.html* file and it knows all right this is the selector I know, you told me that I should know it because it was listed in the bootstrap array in the app module. So now I know that here I should insert the app component and the app component happens to have some HTML code, a template attached to it which is this h3 tag and this is how the Angular application starts here, this is why we see what we see.

# ****Purpose of NgModule****

*NgModules are Angular’s way to group blocks of code that have related domain, workflow, or capabilities. They describe how the application fits together.*

Modules are similar to a namespace in C#. They organize dependencies for both the compiler and dependency injection. They also help with organization and readability of the application code. Lastly, modules can be lazy loaded when they don’t need to be readily available on startup which improves the speed of the application.

Below is an example of the root NgModule. All Angular applications must have at least a root module, usually called AppModule. The app is launched by [bootstrapping](https://angular.io/guide/bootstrapping)the root module.

The bare bones root NgModule generated by Angular CLI

# Structure of NgModule

Every NgModule contains the @NgModule tag. The @[NgModule](https://angular.io/api/core/NgModule) decorator identifies AppModule as an [NgModule](https://angular.io/api/core/NgModule" \t "_blank) class. @[NgModule](https://angular.io/api/core/NgModule) takes a metadata object that tells Angular how to compile and launch the application. It includes the following key pieces:

1. The Declarations Array
2. The Imports Array
3. The Providers Array
4. The Bootstrap Array

## 1. The Declarations Array

Your own components and pipes go here.

Declarations are views, or classes that display data — including components, pipes, and directives.

## 2. The Imports Array

Modules, both external and internally created, get imported. Some examples of external modules include CommonModule (for \*ngFor and \*ngIf directives) and NgBootstrap modules like AccordionModule. An internal module might be an AppRoutingModule where you’ve defined routing paths.

## 3. The Providers Array

Put all your internal services as well as external services here. Providers are services, or classes that get/handle data.

Providers are globally scoped so they don’t need to be exported. Modules that provide services only needed to imported once across the application — in the root module. Examples of modules that provide just services include:

* HttpClientModule: “a simplified client HTTP API that rests on XMLHTTPRequest interface exposed by browsers”
* BrowserAnimationsModule or NoopAnimationsModule (if you want to mock animations, since importing one of the two is required for Angular-Material)

## 4. The Bootstrap Array

The component declared in the bootstrap array is the entry point of the application. According to the Angular documentation, “the bootstrapping process creates the component(s) listed in the bootstrap array and inserts each one into the browser DOM.”

**Components:**

* Component are key features or basic building block in angular. You build your whole application by composing it from a couple of components, which you create on your own.
* It has a selector, template, style, and other properties, and it specifies the metadata required to process the component.

**What is a Selector in Angular?**

* A selector is one of the properties of the object that we use along with the component configuration.
* A selector is used to identify each component uniquely into the component tree, and it also defines how the current component is represented in the HTML DOM.
* When we create a new component using Angular CLI, the newly created component looks like this.

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

@Component({

selector: 'my-app',

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

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

})

export class AppComponent {

name = 'This is simple component';

}

Here in **app.component.ts**, notice that we have one property called **selector** along with the unique name used to identify the app component in the HTML DOM tree once it is rendered into the browser.

Basically, the selector property of the component is just a string that is used to identify the component or an element in the DOM.

By default, the selector name may have an **app** as a prefix at the time of component creation, but we can update it. Keep in mind that two or more component selectors must not be the same.

1. **Selector as the Element Name**

This is the basic version of the selector where the selector name represents the component as the complete element.

When we create the component, we can set the **selector** property along with the appropriate name so that it can be rendered as an element into the HTML DOM tree.

Let’s look at one simple example where we create a simple component called **app**.

**app.component.ts**

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

@Component({

selector: 'my-app',

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

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

})

export class AppComponent {

name = 'This is simple component';

}

Here we have the app component along with **my-app** as the element name so that once the component is rendered, we can identify the component by its name.

**app.component.html**

<my-app></my-app>

1. **Selector as an Attribute**

We can also use a selector as an attribute of an element, just like we do along with other HTML elements.

Remember the attributes used with HTML element such as name, id, class, href, src, value and other different attributes. In Angular, we can also make use of a selector as an attribute.

This is pretty easy to handle whenever we want to play with the input coming all the way from the HTML control/element.

We can use brackets **[ ]** along with a selector, like this.

selector: '[selector\_value]'

Now let’s look at a simple example where we use selector as the attribute.

**App.component.ts**

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

@Component({

selector: '[my-app]',

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

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

})

export class AppComponent {

name = 'Angular';

}

We have just used an additional bracket along with the selector value **[my-app]**, so now we have to use the selector as an attribute and not as an element.

Open the file **index.html** and update the **<my-app>** section like this.

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>MyApp</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

**<div my-app> </div>**

</body>

</html>

1. **Selector as a Class**

we have used a selector as an element and a selector as an attribute, but we can also use the selector as a class.Just like we did the above examples, we just need to provide the selector name as the class and the element will be converted to the class into the HTML DOM.

Let’s modify the **app.component.ts** file like this.

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

@Component({

selector: '.my-app',

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

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

})

export class AppComponent {

name = 'Angular';

}

We have just modified the single statement of the selector property like this.

selector: '.my-app'

We have added a single dot operator, **(.)**, before the selector name, so we can use selector as a class name like this.

**Index.html**

<!doctype html>

<html lang="en">

<head>

<meta charset="utf-8">

<title>MyApp</title>

<base href="/">

<meta name="viewport" content="width=device-width, initial-scale=1">

<link rel="icon" type="image/x-icon" href="favicon.ico">

</head>

<body>

**<div class = "my-app"> </div>**

</body>

</html>

Here in this html file, to specify the class name we should use the property called class, but as you can see, we have used the selector name as a class value.

**Data Binding:**

* Data Binding=communication

**String Interpolation** ( **{{data}}** )

TypeScript Code

(Business logic)

Template

(HTML)

**Property Binding(** **[property]=”data”** )

Output Data

React to User (Events)

**Event Binding ( (event) =”expression” )**

Combination of Both: **Two-way-Binding(** **[(ngModel)]=”data” )**

* **Data Binding** is a way to synchronize the data between the model and view components automatically.
* Data binding can be categorized into 2 types, ie., One-way Binding & Two-way Binding.

1. **One-way Binding**:

* This type of binding is unidirectional, i.e., this binds the data flow from either component to view (DOM) or from the view (DOM) to the component.
* If the *data flow from component to view (DOM)*, then this task can be accomplished with the help of **String Interpolation**& **Property Binding.**
* **String Interpolation:** Angular interpolation is used to display a component property in the respective view template with double curly braces syntax**. Interpolation is used to transfer properties mentioned in the component class to be reflected in its template**.

**Syntax**: class="{{variable\_name}}"

**app.component.html**

<h3>Binding Types</h3>

  <p>Interpolation</p>

<br>

<h5>

      Addition of 3 and 5 with

      Interpolation is {{3+5}}

</h5>

<h5>

      Addition of 3 and 5 without

      Interpolation is 3+5

</h5>

<h2>{{val}}</h2>

**app.component.ts**

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

@Component({

      selector: 'my-app',

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

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

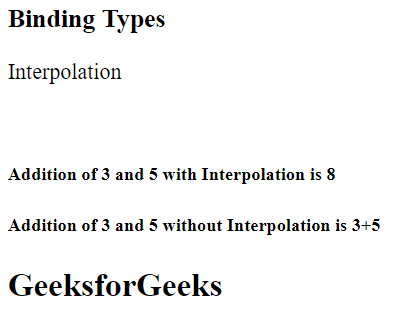
})

export class AppComponent {

      val: string = 'GeeksforGeeks';

}

**Output:**

[](https://media.geeksforgeeks.org/wp-content/uploads/20220714071253/1-300x232.png)

* **Property Binding:** The only difference between Interpolation and Property binding is that we should not store non-string values in variables while using interpolation. So **if we have to store Boolean or other data types then use Property Binding**. In simple words, we bind a property of a DOM element to a field which is a defined property in our component TypeScript code.

**Syntax**: [class]="variable\_name"

**app**.**component.html**

<h3>Binding Types</h3>

<p>Property Binding</p>

<input type="text"

    ng-bind="{{ Geeks }}"><br>

<h5>

    Learning Property binding on {{ Geeks }}

</h5>

<img [src]="image" height="50px" weight="40px">

**app.component.ts**

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

@Component({

  selector: "app-root",

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

  styleUrls: ["./app.component.css"],

})

export class AppComponent {

  title = "Geeks";

  classtype = "text-danger";

  Geeks = "GeeksforGeeks";

  image =

"https://media.geeksforgeeks.org/wp-content/uploads/geeksforgeeks-6.png";

}

**Output**:

[](https://media.geeksforgeeks.org/wp-content/uploads/20220714073643/property-300x234.png)

* If the flow of data is from view to component, then this can be achieved by using the Event Binding.
* **Event Binding:**An event is created whenever either a key is pressed or on a mouse clicked. It is used to handle the events raised by the user actions like button click, mouse movement, keystrokes, etc. When the DOM event happens at an element (e.g. click, keydown, keyup), it calls the specified method in the particular component. Using Event Binding, we can bind data from DOM to the component and hence can use that data for further purposes.

**Syntax**: <button class="btn btn-block"

(click)=showevent($event)>

Event </button>

showevent(event) {

alert("Welcome to GeeksforGeeks"); }

**app.component.html**

<h3>Binding Types</h3>

<p>Event Binding</p>

<button class="btn btn-block"

    (click)="Clickme($event)">

    Click Here

</button>

**app.component.ts**

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

@Component({

  selector: "app-root",

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

  styleUrls: ["./app.component.css"],

})

export class AppComponent {

  title = "Geeks";

  Clickme(event) {

    alert("Welcome to GeeksforGeeks");

  }

}

1. **Two-way Binding:**

* In this type of binding, the immediate changes to the view & component, will be reflected automatically, i.e. when the changes made to the component or model then the *view*will render the changes simultaneously. Similarly, when the data is altered or modified in the view then the model or component will be updated accordingly.
* To make sure the app doesn’t break, we need to import ‘FormsModule’ from ‘@angular/forms.

**app.module.ts**

import { BrowserModule } from '@angular/platform-browser';

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

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

import { AppComponent } from './app.component';

@NgModule({

  declarations: [

    AppComponent

  ],

  imports: [

    BrowserModule,

    FormsModule

  ],

  providers: [],

  bootstrap: [AppComponent]

})

export class AppModule { }

**example:**

**app.component.html**

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

    <h1 style="color: green">

        GeeksforGeeks

    </h1>

    <h3>Two-way Data Binding</h3>

    <input type="text"

        placeholder="Enter text"

        [(ngModel)]="val" />

    <br />

    {{ val }}

</div>

**app.component.ts**

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

@Component({

  selector: "my-app",

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

  styleUrls: ["./app.component.css"],

})

export class AppComponent {

  val: string;

}

**Custom Property Data Binding:**

* Using custom property binding to set the model property of a custom component is a great way for parent and child components to communicate.
* For example following statement in the parent template shows the binding for the property “childItem” in the child component.

**<app-child [childItem]=“parentItem”></app-child>**

* Here <app-child> is the selector defined in the child component.
* childItem is a field in the child component.
* parentItem is a field in parent component.
* By using custom property binding here, childItem is bound to the value of the parentItem

**Using @Input decorator in custom property binding:**

* By default any property in a component is accessible with in the component where it is defined. If you want to expose any property outside of the component then that property has to be decorated with @Input decorator.
* @Input decorator marks a class field as an input property. The input property is bound to a DOM property in the template. During change detection, Angular automatically updates the data property with the DOM property's value.

**Angular Custom Binding Example:**

* The requirement is to show user details in a child component where each user instance is passed from the parent component.

**Creating a User class:** Create a Type script class user. “model.ts” to define a User class. If you are aware of MVC (Model View Controller) pattern then this class is the Model. There are 3 fields name, age and joinDate in the User class.

**user.model.ts**

export class User {

  name : string;

  age : number;

  joinDate : Date;

  constructor(name: string, age : number, joinDate : Date) {

    this.name = name;

    this.age = age;

    this.joinDate  = joinDate;

  }

}

**app.component.ts (Parent component)**

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

import { User } from './user/user.model';

@Component({

  selector: 'app-root',

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

  styleUrls: ['./app.component.css']

})

export class AppComponent {

  users: User[];

  constructor(){

    //Adding User instances to users array

    this.users = [new User('Jack', 56, new Date('2005-03-25')),

    new User('Lisa', 32, new Date('2012-05-09')),

    new User('Jayesh', 28, new Date('2014-10-21'))] ;

}

}

AppComponent uses User model so that is imported. An array of type User is defined and user instances are added to that array in the Constructor.

**app.component.html**

**<div class="contain**er">

    <h3>User Details</h3>

    <app-user \*ngFor="let user of users"

                 [usr]="user">

    </app-user>

</div>

In the template with in the <app-user> selector users array is iterated using **[ngFor directive](https://www.netjstech.com/2020/04/angular-ngfor-directive-with-examples.html" \t "_blank)** and each user instance is bound to the usr property of the child component.

**user.component.ts (Child component)**

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

import { User } from './user.model';

@Component({

selector: 'app-user',

templateUrl: './user.component.html'

})

export class UserComponent {

@Input() usr: User;

}

In the child component usr variable is decorated with @Input decorator indicating that parent component can bind to this property.

**user.component.html**

<div class="container">

    <div class="row">

      <div class="col-xs-6">

        <label>Name: </label> {{ usr.name }}

        <label>Age: </label> {{ usr.age }}

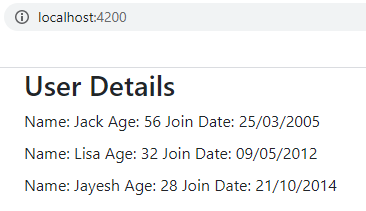
        <label>Join Date: </label> {{ usr.joinDate | date:'dd/MM/yyyy' }}

      </div>

    </div>

  </div>

**Output:**



**Custom Event Binding:**

* Using custom event binding is a great way for parent and child components to communicate where parent component is informed of any change in the child component.

**@Output decorator, $event and EventEmitter:** While doing a custom event binding in Angular three things you will come across are-

* @Output decorator
* EventEmitter class
* $event object

1. **@Output decorator:** A property in a Component that emits a custom event is decorated with @Output decorator.

For example in the statement

**@Output() onUserSelected: EventEmitter<User>;**

OnUserSelected is decorated with @Output() which means this property is going to emit a custom event. That is why this property is of type EventEmitter.

1. **EventEmitter class:** EventEmitter class is used in components with the @Output to emit custom events synchronously or asynchronously. EventEmitter maintains a list of subscribing instances and register handlers for the event.

EventEmitter class has two methods-

emit(value?: T)- Emits an event containing a given value, T signifies the value to emit.

subscribe()- Registers handlers for events emitted by this instance.

When we assign an EventEmitter to an output, subscription of instances is automatically done by Angular so you don't need to use this method explicitly in most of the scenarios.

1. **$event object:** In an event binding, Angular sets up an event handler for the target event. The binding conveys information about the event. This information is encapsulated in $event and may include data values such as an event object, string, or number.

When the event is raised, the handler executes the template statement. For example in the following statement

**(onUserSelected)="showUser ($event)"**

* **(onUserSelected)** is the target event
* **showUser ($event)** is the template statement.

When the event onUserSelected is raised, template statement, which in our example is showUser ($event) method, is executed. Argument of the showUser () method is $event which encapsulates/contains the value passed as method parameter.

**Angular custom event binding example:**

In the example there are two child components UserComponent and UserDataComponent and AppComponent is the parent component.

Initially list of user names is displayed (Done through UserComponent), on clicking any of the user User details are displayed (Done through UserDataComponent).

**user.model.ts**

export class User {

  name : string;

  age : number;

  joinDate : Date;

  constructor(name: string, age : number, joinDate : Date) {

    this.name = name;

    this.age = age;

    this.joinDate  = joinDate;

  }

}

**Directives:**

* AngularJS lets you extend HTML with new attributes called **Directives**.
* AngularJS has a set of built-in directives which offers functionality to your applications.
* AngularJS also lets you define your own directives.
* AngularJS directives are extended HTML attributes with the prefix ng-.  Directives can be used with any of controller or HTML tag which will tell the compiler what exact operation or behavior is expected.

1. **ng-app directive:**

* The ng-app directive starts an AngularJS Application. It defines the root element. It automatically initializes or bootstraps the application when the web page containing AngularJS Application is loaded.
* It is also used to load various AngularJS modules in AngularJS Application.
* The ng-app directive will **auto-bootstrap** (automatically initialize) the application when a web page is loaded.
* In the following example, we define a default AngularJS application using ng-app attribute of a <div> element.

<div ng-app = "">

...

</div>

1. **ng-init directive:**

* The ng-init directive initializes an AngularJS Application data. It is used to assign values to the variables.
* **Example:**

<div ng-app="" ng-init="sort=['quick sort', 'merge sort',

      'bubble sort']">

      Sorting techniques:

      <ul>

      <li>{{ sort[0] }} </li>

      <li>{{ sort[1] }} </li>

      <li>{{ sort[2] }} </li>

      </ul>

    </div>

* In the following example, we initialize an array of countries. We use JSON syntax to define the array of countries.

<div ng-app = "" ng-init = "countries = [{locale:'en-US',name:'United States'},

   {locale:'en-GB',name:'United Kingdom'}, {locale:'en-FR',name:'France'}]">

   ...

</div>

1. **ng-model directive:**

* ng-model is a directive which binds HTML controls( input, select and textarea) and stores the required user value in a variable and we can use that variable whenever we require that value.
* It also is used during validations in a form.

<div ng-app="" ng-init="quantity=1;price=5">

Quantity: <input type="number" ng-model="quantity">

Costs:    <input type="number" ng-model="price">

Total in dollar: {{ quantity \* price }}

1. **ng-repeat directive:**

* The ng-repeat directive repeats HTML elements for each item in a collection.
* The ng-repeat directive actually **clones HTML elements** once for each item in a collection.

<div ng-app="" ng-init="names=[

  {name:'Jani',country:'Norway'},

  {name:'Hege',country:'Sweden'},

  {name:'Kai',country:'Denmark'}]">

  <ul>

    <li ng-repeat="x in names">

      {{ x.name + ', ' + x.country }}

    </li>

  </ul>

  </div>

**Output:**

Looping with objects:

* Jani, Norway
* Hege, Sweden
* Kai, Denmark

1. **ng-bind directive:**

* The ng-bind directive in AngularJS is used to bind/replace the text content of any particular HTML element with the value that is entered in the given expression.
* The value of specified HTML content updates whenever the value of the expression changes in **ng-bind** directive.

<div ng-controller="app">

    num1: <input type="number" ng-model="num1"

        ng-change="product()" />

    <br><br>

    num2: <input type="number" ng-model="num2"

        ng-change="product()" />

    <br><br>

    <b>Product:</b> <span ng-bind="result"></span>

  </div>

  <script>

    var app = angular.module("gfg", []);

    app.controller('app', ['$scope', function ($app) {

      $app.num1 = 1;

      $app.num2 = 1;

      $app.product = function () {

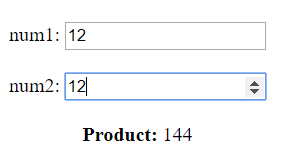
        $app.result = ($app.num1 \* $app.num2);

      }

    }]);

  </script>

**Output:**



In the above example, changing value of num1 or num2 will change product value.

**Structural Directives:** The *structural directives* alter the layout of the DOM by adding, replacing and removing its elements. The two familiar structures are ngIf and ngFor.

1. **ng-if directive:**

* The ng-if Directive in AngularJS is used to remove or recreate a portion of HTML element based on an expression
* If the expression inside it is false then the element is removed and if it is true then the element is added to the DOM.

<div>

<input type="checkbox" ng-model="showDiv" />

    <label for="showDiv">

      Check it

    </label>

    <br><br>

    <div class="geek" ng-if="showDiv">

      GeeksforGeeks is the computer science

      portal for geeks.

    </div>

</div>

 This example adds when user checks the checkbox.

1. **ng-if else directive:**

* The ng-if directive is used to show or hide parts of an angular application. It can be added to any tags, it is a normal HTML tag, template, or selectors. It is a structural directive meaning that it includes templates based on a condition constrained to Boolean.
* When the expression evaluates to true it runs/displays the template given in the “then” clause. Or when the expression evaluates to false it displays the template given in the “else” clause. If there is nothing in else clause, it will by default display blank.

**Syntax:**

<div \*ngIf="condition; else elseStatement">

  when condition is true.

</div>

<ng-template #elseStatement>

  when condition is false.

</ng-template>

<!--It can be seen that the else

  clause refers to ng-template,

  with the label #elseStatement -->

**Example:**

**app.component.ts**

<!-- \*ngIf else -->

<div class="container-fluid">

  <div class="row bg-success

      text-light h1 justify-content-center

      align-items-center" \*ngIf="check;

      else elseStatement"

      style="height:150px">Condition true

  </div>

<ng-template #elseStatement>

  <div class="row bg-danger

      text-light h1 d-flex

      justify-content-center align-items-center"

      style="height:150px">Condition false

  </div>

</ng-template>

</div>

1. **ng-for directive:** It is a repeater directive that customizes data display. It can be used to display a list of items.

**Example:**

**app.componet.html**

<ol>

    <li \*ngFor='let i of a'> {{i}} </li>

</ol>

**app.component.ts**

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

@Component({

selector: 'app-root',

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

})

export class AppComponent {

a=['gfg1', 'gfg2', 'gfg3', 'gfg4']

}

**Output:**

