ANGULAR NOTES

Q) **What is Data?**

Data is collection of raw facts. Data may be text, audio, video, streaming data ,whenever we develop any web application. Application is collection of files (code) Files namely: .html , .js , .css , .aspx , .php , .py , .jsp , etc…. we can develop web application by using Different Programming Languages, Technologies, Frameworks, Database softwares, UI Technologies. After developing the application we have to deploy the `application in webserver so that end user will access the application via Browser and internet .

Q) what is Web Browser?

Web Browser is a software which is used to access the files from webserver , Ex: - internet Explorer, Mozilla Firefox, Google chrome

Q) what is web server?

Webserver is a software which provides services to different clients like Browsers, Mobiles etc...

Q) what web server consists of ?

Web application Ex:- IIS(internet information services),.net, Jboss, java,Apache, php, Django, java.

Web application: - The application that was deployed on webserver and can be accessible via Browser and internet. Web applications are multi user applications

Web applications are of 2 Types:-

1. Static web application

2. Dynamic web application

Static web application: - The application whose output is common for all the users

Static web applications can be developed by using UI Technologies

Ex: - blogs, Movie review websites, w3schools

UI Technologies=HTML + JavaScript + CSS

HTML: - Display the content on Browser using (Headings, Paragraphs, Table, List, Link,

Frame, Form, etc,)

JavaScript :- is used to perform Operations on HTML Elements Validations, Logics

CSS--------> it is used to apply styles for HTML Elements Ex:- blogs,movie review websites ,w3schools

Dynamic web application:-The application whose output change based on user/ location/ time/ search

Dynamic web applications can be developed by using UI Technologies + Java/ .net/ Php / Python/ Angular.

Different Dynamic Web Applications:-

1. Ecommerce (amazon,flipkart,ebay)

2. Banking (icici,axis,citi)

3. insurance (health,vehicle,lic)

4. Socialnetworking(facebook,twitter,linkden)

5. Education (college,school,w3schools,jntu results,byjuice,examantaion,library)

6. Entertainment (Bookmyshow,amazon prime,netflix,hotstar)

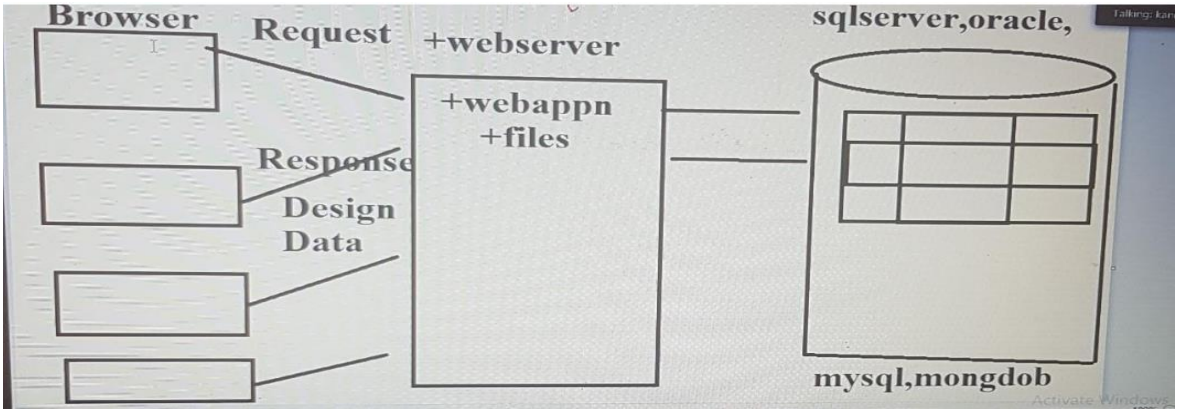
7. services

public services: eseva, meseva, irctc, manabadi, incometax websites, apsrtc, tsrtc

private: redbus, abhibus, ola,swiggy

8. Healthcare

Web Application Architecture: whenever we develop dynamic web application we require Browser, Webserver and Database server.



Technology	Developed By
Javascript :- scripting Language	Netscape
Jquery :- Javascript Library	Microsoft
AnglarJS :- Javascript Framework	Google
Typescript :- Programming Language	Microsoft
Angular2,4,5,6,7,8,9 :-Typescript Framework	Google

Prog Langs	Technologies	Frameworks	Database s/ws
java	servlets, JSP, JDBC	Spring, ORM, Spring Boot, Rest	Oracle, MySQL
C#.net	ASP.net, ADo.net	ASP.net MVC, WCF, WebAPI Entity Framework,	Sql Server
python	----------	Django, Flask	Mongo dB
	php	Cakephp	MYSQL

Web application Developers are classified into 2 Types:-

1. Full Stack Developer : - UI Technologies + Angular + Java/.net/Python/PHP+Database

2. Mean Stack Developer: - UI Technologies + Angular + nodejs + Expressjs + mongdb

After developing any webapplication we have to deploy the appn on webserver

so that enduser will access the application via Browser and internet

S/w companies

product	s/w products	service products
Microsoft	typescript,win os,.net,sqlserver, Msoffice,Azure	hotmail,skype,notepad.
Google	angularjs, angular,andoid,gcp	gmail,youtube, gpay,gdrive,gmaps

Facebook	reactjs	facebook,insta,whatsapp
amazon	AWS	amazon prime,amazon shoping cart
oracle	java,oracle	----

some service based Companies: TCS,Wipro,Tech,accenture,csc,cognizent

S/w company-------->Develop applications---→Application is collection of programs.

Programming Languages :- C,C++,Java,C#.net,Typescript,python

Scripting Languages :- Javascript,Vbscript,python

Markup Languages :- HTML,HTML5

Library :- Jquery,ReactJS

Technologies :- ASP.net,ADO.net (.net), JSP,servelets,JDBC (java),PHP

Frameworks

Client Side Framework:-AngularJS, Angular2/4/5/6/7/8/9, Vuejs, Knockjs, backboneJS,ExpressJS, node.js

Server Side Frameworks:-spring,ASP.net,MVC,SpringBoot,Django,Flask,WCF,WEBAPI,.netcore,ASpCore

DataBase Softwares :- Sqlserver,Oracle,MySQL,Mongdb

Query Languages :- SQL,HQL,LINQ,PLSQL,TSQL

Design Patterns :-MVC,MVT,MVVM,Singleton,Factpory,DAO etc..

DataTransfer :- XML,JSON

Architecture :- 3Tier,NTier

Design Principles :- SOLID,ACID

CloudTechnologies :- AWS,Azure,salesForce

SDLC Lifecycles :- Waterfall,V-Model,Spiral,Rad,Azile model

Protocols :-Http,Https,TCP,FTP,SMTP

Design Patterns:- Design Pattern is a readymade solution for already existing problems

Framework:-Framework is a software that was developed by using a Design pattern and by using a Language. These are of 2 types: MVT and MVC.

MVC(Design Pattern)			**MVT (component architecture)**	
C#.net	java	javascript	python	Typescript
ASP.net MVC	Spring	AngularJS	Django	Angular8

JRE=JVM+Library Classes and JDK=JRE+Development Tools

Development Tools are Editors which are used to develop appns

JVM:- Memory Management and execution of Java programs

.net Framework :- CLR+Base class Libraries

Development Tools:- Microsoft VisualStudio Editor

Angular is a Typescript Framework which is used to develop Dynamic Webapplications

Typescript:- Typescript is a programming Language

Concepts to be covered: Operators, Variables, Data types, Conditional Statements, Loops, OOPS, Collections, Lambda Expressions, Exception Handling.

TYPE SCRIPT

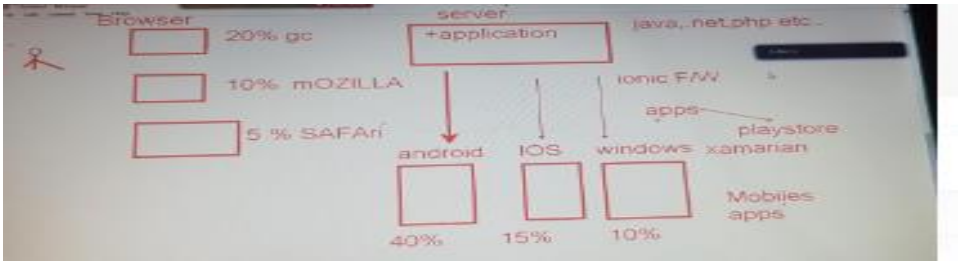
Typescript is an Open source language. It is introduced by **Anders Hejsberg.** Typescript is a product of Microsoft under the Apache 2 licence. Whenever we develop any web application we have to write types of logics.75% code will be on browser and 25% will be executed on server.

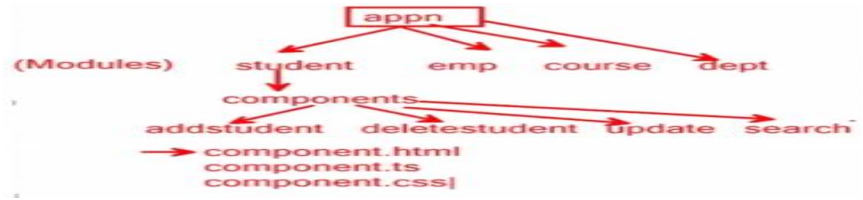
Web Application:

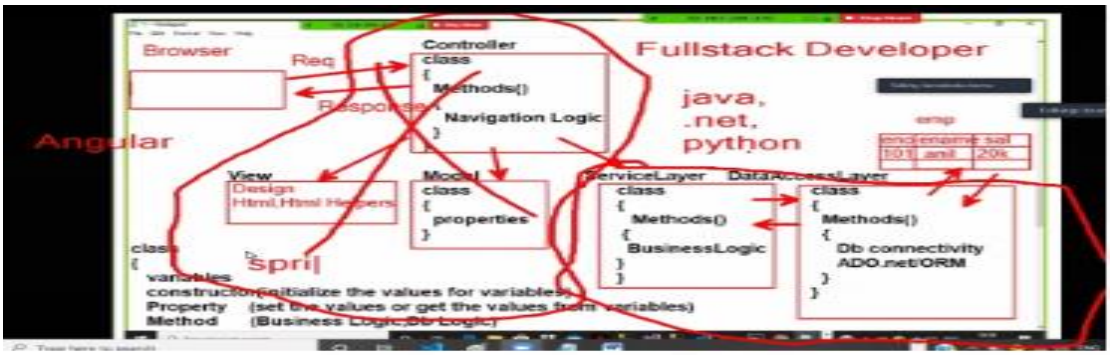
1. Presentation Logic :- Designing code(Html +CSS) + validation Code(Java Script)
2. Business Logic : - Server side Technologies(c#.net, java, Python)
3. Database Logic : - Server Side Technologies (ADO.net, Entity Framework)
4. Integration Logic (c#.net)

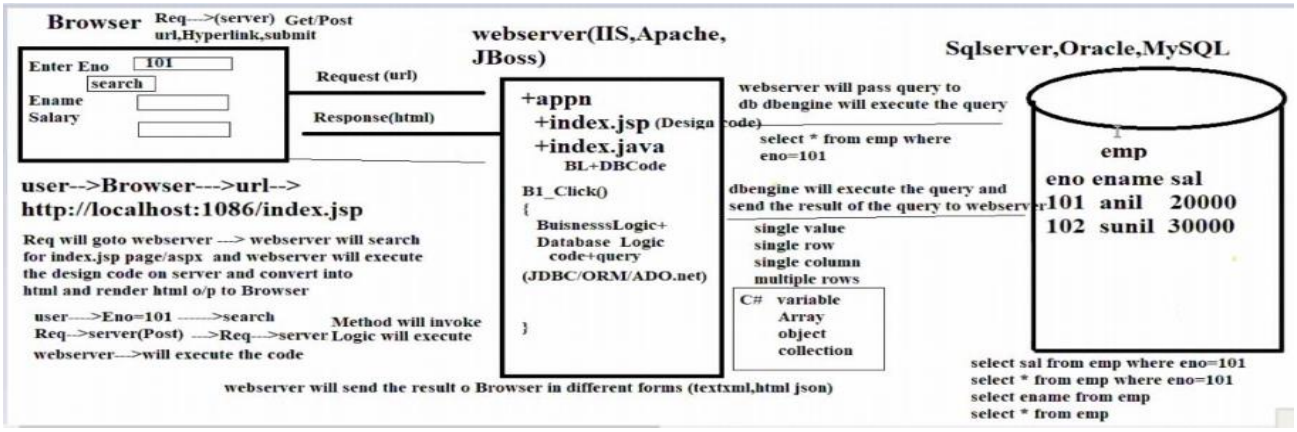
Presentation and Business logic are performed at Browser only.

Only the Database Logic is performed at Server side.

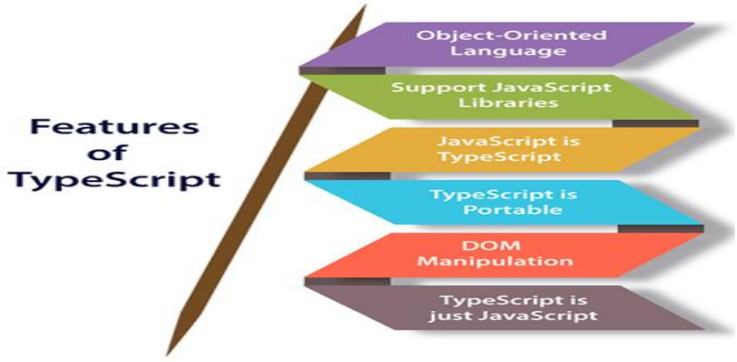








75% percent of code executed by Angular.Only 25%of code given to sever.Ex:db Connection



Advantage of TypeScript over JavaScript

- TypeScript always highlights errors at compilation time during the time of development, whereas JavaScript points out errors at the runtime.
- TypeScript runs on any browser or JavaScript engine.

Disadvantage of TypeScript over JavaScript

- TypeScript takes a long time to compile the code.
- TypeScript does not support abstract classes. If we run the TypeScript application in the browser, a compilation step is required to transform TypeScript into JavaScript.

S.No	Java Script	TypeScript
1	It doesn't support strongly typed or static typing.	It supports typed or static typing feature.
2	It was developed by netscape in 1995.	It was developed by Anders Hejsberg in 2012.
3	Java script source file in '.js' extension.	Type script file in '.ts'extension.
4	Its directly run on the browser.	It is not directly run on the browser.
5	It is just a scripting language	It supports object oriented programming concept like classes , interfaces, inheritance, generics, etc.,
6	It doesn't support optional parameters	It supports optional parameters.
7.	It is interpreted language that's why it highlighted the errors at runtime.	It compiles the code an highlighted errors during the development time.
8	Compiler is not required for java script	Typescript use tsc compiler
9	Java script does not support modules.	Type script supports modules.

10	class classname
	{
	Variables
	Methods
	}
		class classname:
		var
		methods

At compile time Typescript code will convert into javascript

Code execution process of Typescript:-

1. write the program

2. save the program

3. compile the program

4, execute the program

ts program---> .ts--->tsc ---->.js

After compilation of Typescript program then tsc compiler will convert typescript into

Q)what is Transpling?

it is a process of converting typescript code into javascript

node.js

node.js:-node.js is a Runtime Environment for javascript

it is responsible to convert typescript code into javascript

whenever we develop any appn in angualr we mostly write

angular appn consists of 3 types of files

1. .html files (Design the webpages)

2. .css files (apply styles)

3. .ts files (typescript files) (validation logic,BusinessLogic)

node.js is required only at the time of development

After developing the angular appn then Bundling and Minfication will apply and total angular appn is converted into zip file which internally consists of .html,.js,.css files

once when zip is ready node.js is not required and finally we have to deploy the zip file in webserver(IIS,Apache,JBoss).

Softwares Required to run the Angular Application:

1. install node.js

node.js is a runtime environment ,node.js will exist only at the appn development, angular appn consists of 3 types of files:.html,.css and .ts. in order to run angular project we write a command **ng serve -o**

whenever we run the above command then node.js will run .ts files and transpling will be applied i.e all ts files will convert into js files . now Bundling and Minification will happen .Bundling means grouping some set of files into single file,Minification means reduce the size of the file by removing comments and spaces. Total angular project will convert into zip file and then we have to deploy the zip file on Webservers(IIS,Jboss,Apache)

2. **npm:-** node package manager, npm is the largest Library which provides services to angular and many more javascript Frameworks

3. **Typescript**:- is a programming Language which is used to write Logics

4. **Angular**:- Angular is a typescript Framework

Angular Installation steps:

software installation steps :-

1. install node.js

goto---->Google--->node.js --->download the latest version

whenever we download and install node.js then node.js command prompt is available npm can be accessable after installing node.js

2. check whether node.js was installed or not

open node.js command prompt

node -v

3. check whether npm was installed or not

npm -v

4. install typescript

npm install -g typescript

5. check whether typescript was installed or not

tsc -v

6. install angular cli

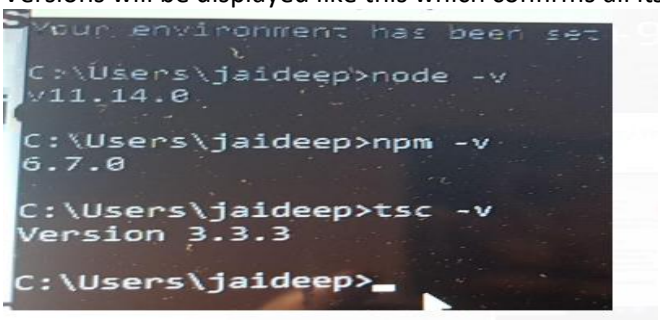
npm install -g @angular/cli

7. check whether angular cli was installed or not

ng v

8. install visual studio code ,sublime

Versions will be displayed like this which confirms all its installations.



Developing the First program in Typescript:-

1. goto F Drive and create a folder with name angular7AM
2. goto--->start--->visualstudiocode---->
3. goto-->new file
4. save the program in F:/Angular7AM with name 1.ts
5. goto---->Terminal--->new terminal---->
6. change the Drive F:
7. change the Directory cd Angular7AM
8. compile the program tsc filename.ts **Ex: tsc 1.ts**
9. Execute the program node filename **Ex:- node 1**

Syntax for declaring variable in TS : **var variablename:datatype=value;**

Example Pgm:-

```
var x:number=8;                x,y are Global variables
var y:number=4;
function Add()
{
  let sum:number=x+y;          sum is local variable
```

```
    console.log("sum is"+sum);        scope is within the function
  }
  function Sub()
  {
    let diff:number=x-y;              diff is local variable
    console.log("Diff is"+diff);
  }
```

Output :
Sum is 12
Diff is 4

PDLC

PDLC:- Program Development Life Cycle
PDLC is an approach which is used to develop programs fastly and Efficiently
Steps:-
1. Problem Definition
2. Creating solution for the problem
3. Algorithm
4. Flowchart
5. PRS(Program Requirement Specification)
6. Coding
7. Testing

1. Problem Definition:- understand the problem
Q) consider 3 subject marks of the student are 70,80,90
Display total,percentage?

2.Creating the solution for the problem :-

```
caltotal=70+80+90
calper=total/3
```

3.Algorithm:- stepwise refinement for a solution is Algorithm
1. start
2. consider m1 is 70
   m2 is 80
   m3 is 90
3. add m1,m2,m3 and store the result in total
4. divide total with 3 and store the result in p
5. print total,percentage
6. stop

4. Flowchart:- The diagramatical representation of Algorithm is Flowchart.

5.PRS:- Program Requirement Specification
1. identify the i/p variables that are required to develop the program : m1,m2,m3
2. identify the o/p variables : total,percentage
3. identify the operators : +,/,=
4. identify the operations we are performing : 2 operations
5. identify the functions : 2 functions : caltotal(), calPercentage()
6. Expected i/p Expected o/p
   m1=70 total is 240
   m2=80 percentage is 80
   m3=90

6.Coding:-

```
var m1:number=70;
var m2:number=80;
var m3:number=90;
var total:number;
function CalTotal(){
  total=m1+m2+m3;
  console.log("Total is"+total)
```

```
}
function CalPercentage()
{
  let p:number=total/3;
  console.log("Percentage is"+p);
}
CalTotal();
CalPercentage();
```

Output:

Total is 240
Percentage is 80

7.Testing:- if Expected O/p is Equal to Actual O/p then Testcase is passed otherwise Failed

Expected O/p	Actual O/p
Total is 240	Total is 240
Percentage is 80	Percentage is 80

if Expected O/p is Equal to Actual O/p the Testcase is passed otherwsie Testcase is Failed.

Variables:

A variable is the storage location, which is used to store value/information to be referenced and used by programs. It acts as a container for value in code and must be declared before the use. We can declare a variable by using the var keyword in typescript. In TypeScript, the variable follows the same naming rule as of JavaScript variable declaration. These rules are-

- The variable name must be an **alphabet** or **numeric digits(a13,s78h78,djhh876756gdf878)**.
- The variable name cannot start with digits.
- The variable name cannot contain **spaces** and **special character**, except the u**nderscore(_)** and the **dollar($)** sign.

Variable is the name given for a particular memory location. The **let** keyword is similar to **var** keyword in some respects, and **const** is an let which prevents re-assignment to a variable.

Q) What is the purpose of variable?

The purpose of variable is to store the value

Q) What is variable declaration?

Declaring the variable without assigning the value is called as variable declaration

```
var x:number;
```

Q) What is variable initialization?

Assigning the value to the variable at the time of declaring the variable is called as variable initialization. ex: var x:number=10;

Q) What is variable Assignment?

Assigning the value to the variable after declaring the variable is called as variable Assignment

```
var x:number;
  x=10;
```

Variable Declaration

We can declare a variable in one of the four ways:

1. Declare type and value in a single statement

var [variablename: [datatype] = value;

2. Declare type without value. Then the variable will be set to undefined.

var [variablename] : [datatype];

3. Declare its value without datatype. Then the variable will be set to 'any' datatype.

var [variablename] = value;

4. Declare without value and type. Then the variable will be set to 'any' datatype and initialized with 'undefined' value. var [variablename];

In functional programming variables are divided into 2 Types
1. Global Variables
2. Local Variables

1.**Global Variables:-** The variables that are declared outside the function are called as Global Variables.

The scope of Global Variable is in multiple functions in functional programming we can declare variables by using 'var' and 'let' keyword.

it is always recommended to declare Global variables with 'var'' keyword.

syntax:- var variablename:datatype=value;

Ex:- var x:number=10;

2. Local variable:- The variables that was declared inside the function are called as local variables. it is always recommended to declare local variable with **let** keyword

if we declare variable with let keyword the scope is within the function

syntax:- let x:number=10;

Ex:-

```
var x:number=8;                          x,y are Global variables
var y:number=4;
function Add()
{
  let sum:number=x+y;              sum is localvariable
  console.log("sum is"+sum);              scope is within the function
}
function Sub()
{
  let diff:number=x-y;             diff is local variable
  console.log("Diff is"+diff);
}
```

Output :
Sum is 12
Diff is 4

Let's understand all the three variable keywords one by one.

1.var keyword

Generally, **var** keyword is used to declare a variable in JavaScript. var x = 50;

We can also declare a variable inside the function:

```
function a() {
    var msg = " Welcome to JavaTpoint !! ";
    return msg;
}
a();  //function call
```

We can also access a variable of one function with the other function:

```
function a() {
   var x = 50;
   return function b() {
       var y = x+5;
       return y;
   }
}
var  c = a(); //55
b();      //returns '55'
```

Scoping rules

Variable with var keyword can be accessed through out the program because its recognised as global variable.

Example

```
function f()
{
   var X = 5; //Available globally inside f()
```

```
    if(true)
    {
        var Y = 10; //Available globally inside f()
        console.log(X); //Output 5
        console.log(Y); //Output 10
    }
    console.log(X); //Output 5
    console.log(Y); //Output 10
}
f();
console.log(X); //Returns undefined because value cannot accesses from outside function
console.log(Y); //Returns undefined because value cannot accesses from outside function
```

let declarations

The let keyword is similar to the var keyword. The let keyword has some restriction in scoping in comparison of the var keyword.

The let keyword can enhance our code readability and decreases the chance of programming error.

The let statement are written as same syntax as the var statement:

var declaration: var b = 50;

let declaration: let b = 50;

The key difference between var and let is not in the syntax, but it differs in the semantics (logic). The Variable declared with the let keyword are scoped to the nearest enclosing block which can be smaller than a function block.

Example

```
function f()
{
    var X = 5; //Available globally inside f()
    if(true)
    {
        let Y = 10; //Available locally inside if()
        console.log(X); //Output 5
        console.log(Y); //Output 10
    }
    console.log(X); //Output 5
    console.log(Y); //error
}
f();
```

const declarations

The const declaration is used to declare permanent value, which cannot be changed later. It has a fixed value. The const declaration follows the same scoping rules as let declaration, but we cannot re-assign any new value to it.

Note: According to the naming standards, the const variable must be declared in **capital letters**. Naming standards should be followed to maintain the code for the long run.

Example

```
function constTest(){
  const VAR = 10;
  console.log("Value is: " +VAR);
}
constTest();
```

Output:

```
Value is: 10
```

What will happen when we try to re-assign the const variable?

If we try to re-assign the existing const variable in a code, the code will throw an error. So, we cannot re-assign any new value to an existing const variable.

Example

```
function constTest(){
```

```
  const VAR = 10;
  console.log("Output: " +VAR);  // Output: 10
  const VAR = 10;
  console.log("Output: " +VAR);  //Uncaught TypeError: Assignment to constant variable
}
constTest();
```

Output:

SyntaxError: Identifier 'VAR' has already been declared.

Var vs. Let Keyword

SNo	Var	Let
1.	The var keyword was introduced with JavaScript.	The let keyword was added in ES6 (ES 2015) version of JavaScript.
2.	It has global scope.	It is limited to block scope.
3.	It can be declared globally and can be accessed globally.	It can be declared globally but cannot be accessed globally.

4.	Variable declared with var keyword can be re-declared and updated in the same scope.
	Example:
	function Bhargav(){
	var a = 10;
	var a = 20; //a is replaced
	console.log(a);
	}
	Bhargav();
		Variable declared with let keyword can be updated but not re-declared.
		Example:
		function varGreeter(){
		let a = 10;
		let a = 20; //SyntaxError:
		//Identifier 'a' has already been declared
		console.log(a);
		}
		varGreeter();

Decision Making Statements

The decision making always returns the Boolean result true or false.

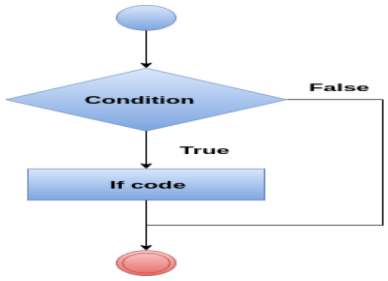
There are various types of Decision making in TypeScript:

if statement

It is a simple form of decision making. It decides whether the statements will be executed or not, i.e., it checks the condition and returns true if the given condition is satisfied.

Syntax

```
if(condition) {      // code to be executed  }
```



Example

```
let a = 10, b = 20;
if (a < b)  {    console.log('a is less than b.');  }
```

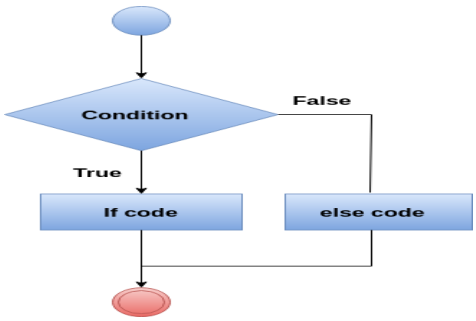
Output: a is less than b.

if-else statement:

The if statement only returns the result when the condition is true. But if we want to returns something when the condition is false, then we need to use the if-else statement. The if-else statement tests the condition. If the condition is true, it executes if block and if the condition is false, it executes the else block.

Syntax

```
if(condition) {    // code to be executed  }
 else {     // code to be executed  }
```



Example

```
let n = 10
if (n > 0) {      console.log("The input value is positive Number: " +n);
} else {
   console.log("The input value is negative Number: " +n);
}
```

Output:

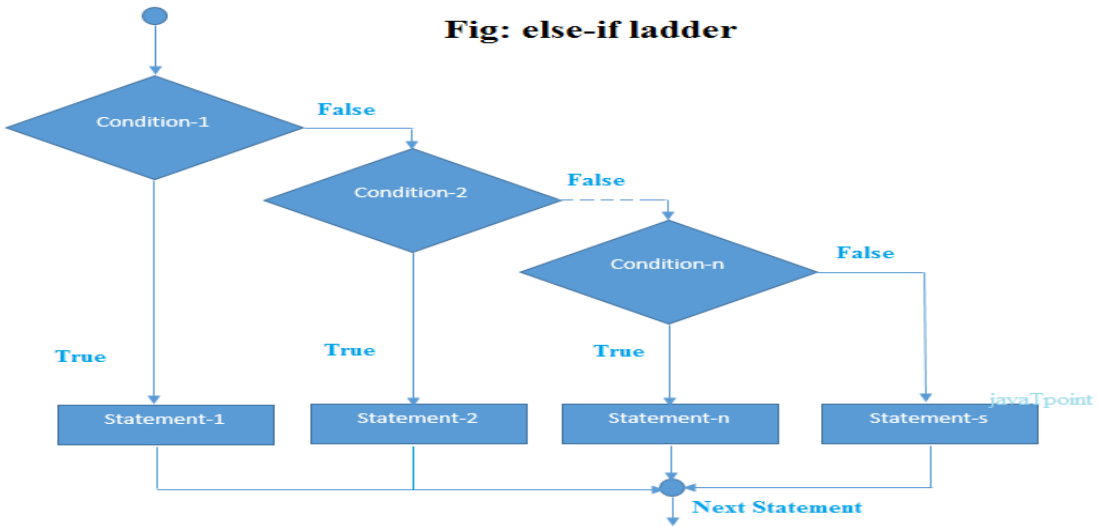
The input value is positive Number: 10

if-else-if ladder

Here a user can take decision among multiple options. It starts execution in a top-down approach. When the condition gets true, it executes the associated statement, and the rest of the condition is bypassed. If it does not find any condition true, it returns the final else statement.

Syntax

```
if(condition1){
//code to be executed if condition1 is true
}else if(condition2){
//code to be executed if condition2 is true
}
else if(condition3){
//code to be executed if condition3 is true
}
else{
//code to be executed if all the conditions are false
}
```



Example

```
let marks = 95;
if(marks<50){     console.log("fail");  }
else if(marks>=50 && marks<60){
   console.log("D grade");  }
else if(marks>=60 && marks<70){
   console.log("C grade");  }
else if(marks>=70 && marks<80){
   console.log("B grade");  }
else if(marks>=80 && marks<90){
   console.log("A grade");  }
else if(marks>=90 && marks<100){
   console.log("A+ grade");  }
else{
   console.log("Invalid!");  }
```

Output:

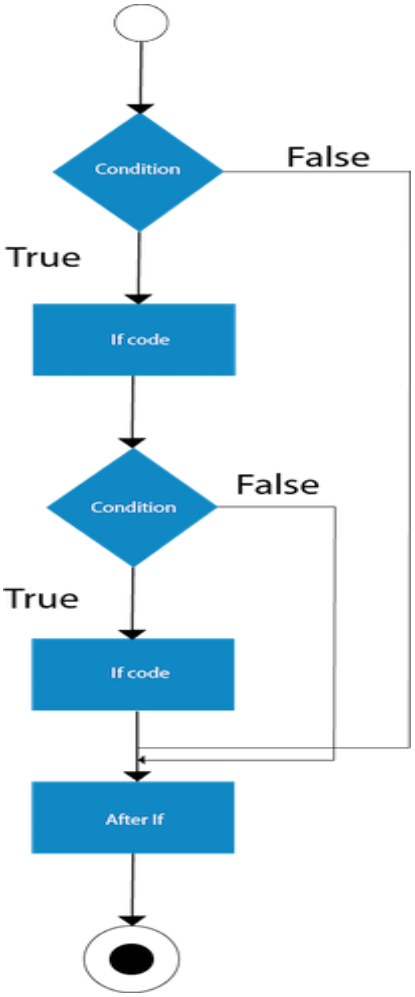
A+ grade

Nested if statement

Here, the if statement targets another if statement. The nested if statement means if statement inside the body of another if or else statement.

Syntax

```
if(condition1) {      //Nested if else inside the body of "if"
   if(condition2) {         //Code inside the body of nested "if"     }
   else {         //Code inside the body of nested "else"      }
}
else {      //Code inside the body of "else."  }
```



Example

```
let n1 = 10, n2 = 22, n3 = 25
if (n1 >= n2) {
  if (n1 >= n3) {     console.log("The largest number is: " +n1)     }
    else {    console.log("The largest number is: " +n3)    }
}
else {
    if (n2 >= n3) {   console.log("The largest number is: " +n2)    }
    else {    console.log("The largest number is: " +n3)        }
}
```

Output:

The largest number is: 25

TypeScript Switch Statement

The TypeScript switch statement executes one statement from multiple conditions. It evaluates an expression based on its value that could be Boolean, number, byte, short, int, long, enum type, string, etc. A switch statement has one block of code corresponding to each value. When the match is found, the corresponding block will be executed. A switch statement works like the if-else-if ladder statement.

The following points must be remembered in a switch statement:

- There can be N number of cases inside a switch statement.
- The case values must be unique.
- The case values must be constant.
- Each case statement has a break statement at the end of the code. The break statement is optional.

- The switch statement has a default block which is written at the end. The default statement is optional.

Syntax

```
switch(expression){
case expression1:
   //code to be executed;
   break;  //optional
case expression2:
   //code to be executed;
   break;  //optional
   ........
default:
   //when no case is matched, this block will be executed;
   break;  //optional
}
```

The switch statement contains the following things. There can be any number of cases inside a switch statement.

Case: The case should be followed by only one constant and then a semicolon. It cannot accept another variable or expression.

Break: The break should be written at the end of the block to come out from the switch statement after executing a case block. If we do not write break, the execution continues with the matching value to the subsequent case block.

Default: The default block should be written at the end of the switch statement. It executes when there are no case will be matched.

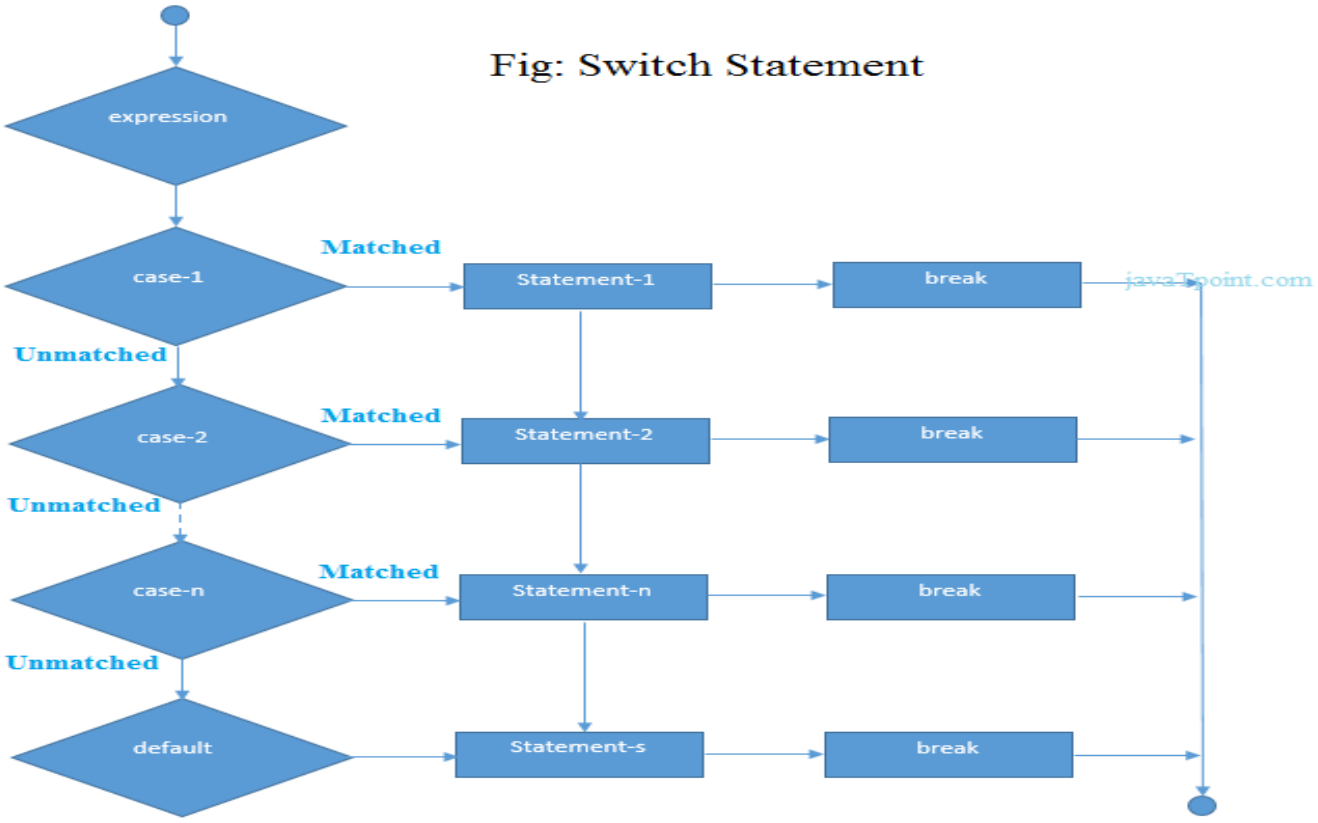


Fig: Switch Statement

Example

```
let a = 3;
let b = 2;
switch (a+b){
   case 1: {  console.log("a+b is 1.");      break;    }
```

```
5.     case 2: {  console.log("a+b is 5.");       break;     }
6.     case 3: {   console.log("a+b is 6.");      break;      }
7.     default: {  console.log("a+b is 5.");      break;     }
8.  }
```

Output:a+b is 5

Switch case with String

```
let grade: string = "A";
switch (grade)
{   case'A+':   console.log("Marks >= 90"+"\n"+"Excellent");   break;

    case'A':      console.log("Marks [ >= 80 and <90 ]"+"\n"+"Good");      break;

    case'B+':       console.log("Marks [ >= 70 and <80 ]"+"\n"+"Above Average");       break;

    case'B':      console.log("Marks [ >= 60 and <70 ]"+"\n"+"Average");       break;

    case'C':      console.log("Marks < 60"+"\n"+"Below Average");       break;

    default:        console.log("Invalid Grade.");
}
```

In this example, we have a string variable grade. The switch statement evaluates grade variable value and match with case clauses and then execute its associated statements.

Output: Marks [>=80 and <90]
Good

Switch Case with Enum

In TypeScript, we can use the switch case with Enum in the following ways.

Example

```
1.  enum Direction {
2.     East,
3.     West,
4.     North,
5.     South
6.  };
7.  var dir: DirectionDirection = Direction.North;
8.  function getDirection() {
9.     switch (dir) {
10.       case Direction.North: console.log('You are in North Direction');          break;
11.       case Direction.East: console.log('You are in East Direction');           break;
12.       case Direction.South: console.log('You are in South Direction');          break;
13.       case Direction.West: console.log('You are in West Direction');           break;
14.    }
15. }
16. getDirection();
```

Output: You are in North Direction

TypeScript Switch Statement is fall-through.

The TypeScript switch statement is fall-through. It means if a break statement is not present, then it executes all statements after the first match case.

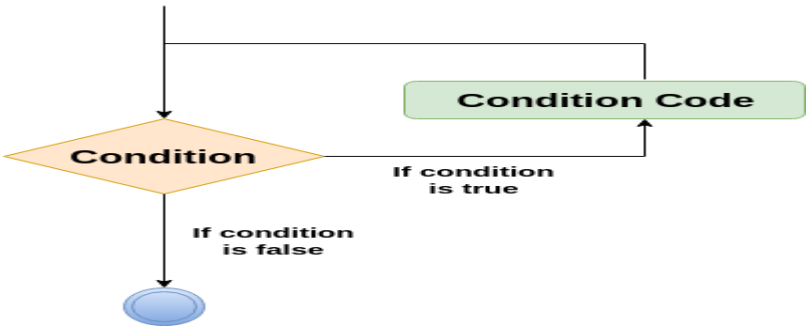
Example

```
1. let number = 20;
2. switch(number)
3. {       //switch cases without break statements
4.    case 10: console.log("10");
5.    case 20: console.log("20");
6.    case 30: console.log("30");
7.    default: console.log("Not in 10, 20 or 30");
8. }
```

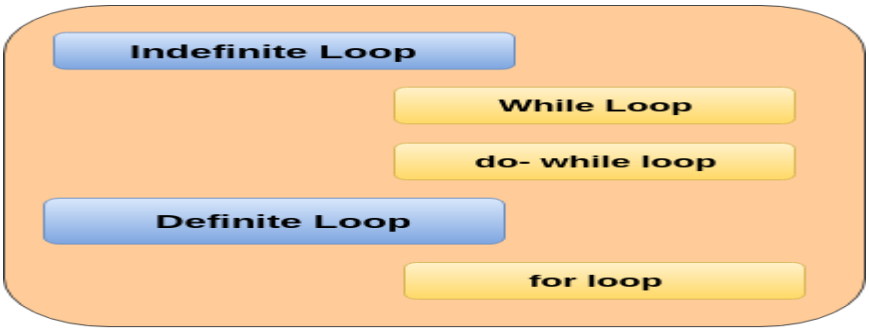
Output:20
30
Not in 10,20 or 30

TypeScript Indefinite Loops

In a programming language, loops are the sequence of instructions which continually repeated until a specific condition is not found. It makes the code compact. We can mostly use it with the array. Below is the general structure of the loop statement:



We can classify the loops into two types:



Indefinite Loop

In Indefinite loops, the number of iterations is not known before beginning the execution of the block of statements. There are two indefinite loops:

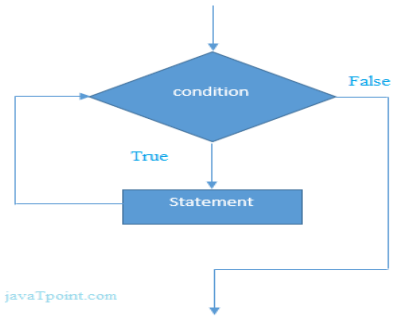
1. while loop
2. do-while loop

TypeScript while loop

The TypeScript while loop iterates the elements for the infinite number of times. It executes the instruction repeatedly until the specified condition evaluates to true. We can use it when the number of iteration is not known. The while loop syntax is given below.

Syntax

```
1. while (condition)  {     //code to be executed  }
```



The explanation of while loop syntax is:

While loop starts the execution with checking the condition. If the condition evaluates to true, the loop body statement gets executed. Otherwise, the first statement following the loop gets executed. If the condition becomes false, the loops get terminated, which ends the life-cycle of the loops.

Example

```
1. let num = 4;
2. let factorial = 1;
3. while(num >=1) {
4.    factorialfactorial = factorial * num;
5.    num--;
6. }
7. console.log("The factorial of the given number is: "+factorial);
```

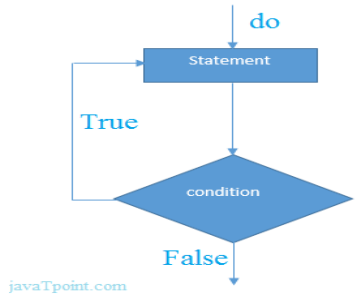
Output: The factorial of the given number is: 24

TypeScript do-while loop

The TypeScript do-while loop iterates the elements for the infinite number of times similar to the while loop. But there is one difference from while loop, i.e., it gets executed at least once whether the condition is true or false. It is recommended to use do-while when the number of iteration is not fixed, and you have to execute the loop at least once. The do-while loop syntax is given below.

Syntax

```
1. do{       //code to be executed
2. }while (condition);
```



The explanation of do-while loop syntax is:

The do-while loop starts executing the statement without checking any condition for the first time. After the execution of the statement and update of the variable value, it starts evaluating the condition. If the condition is true, the next iteration of the loop starts execution. If the condition becomes false, the loops get terminated, which ends the life-cycle of the loops.

Example

```
1. let n = 10;
2. do {    console.log(n);
3.     n++;
4.  } while(n<=15);
```

Output: 10
11
12
13
14
15

TypeScript Definite Loop

In this loop, we know about the number of iterations before the execution of the block of statements. A "**for loop**" is the best example of this loop. Here, we are going to discuss three types of the loop:

1. for loop
2. for..of loop
3. for..in loop

TypeScript for loop

A for loop is a **repetition** control structure. It is used to execute the block of code to a specific number of times. A for statement contains the initialization, condition and increment/decrement in a single line which provides a shorter, and easy to debug structure of looping. The syntax of for loop is given below.

Syntax

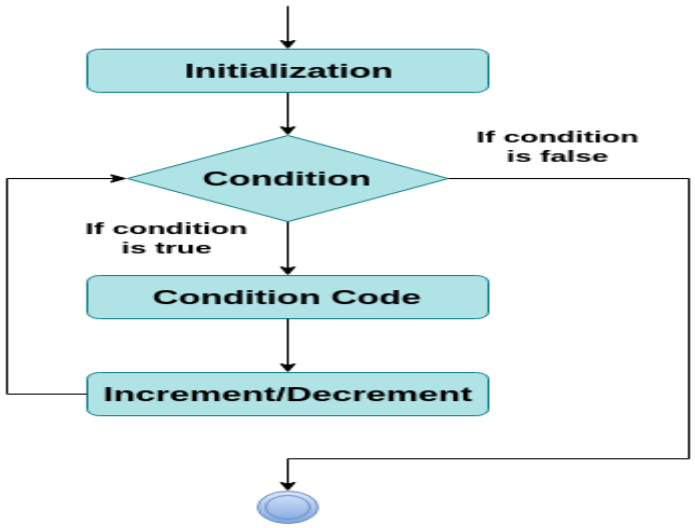
```
1. for (first expression; second expression; third expression ) {
2.     // statements to be executed repeatedly
3. }
```

Explanation of the flow of control in a "**for loop**" is:

The first expression is the **initialization** step, which is executed first, and only once. It allows us to declare and initialize the loop control variables.

The next expression evaluates the **condition**. If it is true, the body of the loop gets executed. If it is false, the loop does not execute, and the flow of control jumps to the next statement just after the "**for**" loop.

When the body of the "**for loop**" executes, the flow of control jumps to the increment/decrement statement. It allows us to update the loop control variables. It can be left blank, as long as a semicolon appears after the condition.
Now, the condition is **re-evaluated**. If it finds true, the loop executes, and the process repeats. When the condition becomes false, the "**for loop**" terminates which marks the end of the life-cycle.



Example

```
1. let num = 4,i;
2. let factorial = 1;
3. for(i=0;i<num;i++) {
4.    factorialfactorial = factorial * num;
5. }
6. console.log("The factorial of the given number is: " + factorial);
```

Output: The factorial of the given number is: 24

TypeScript for..of loop

The for..of loop is used to iterate and access the elements of an array, string, set, map, list, or tuple collection. The syntax of the for..of loop is given below.

Syntax

```
1. for (var val of list) {
2.    //statements to be executed
3. }
```

Example

```
1. let arr = [1, 2, 3, 4, 5];
2. for (var val of arr) {
3.   console.log(val);
4. }
```

Output:1
2
3
4
5

TypeScript for..in loop

The for..in loop is used with an array, list, or tuple. This loop iterates through a list or collection and returns an index on each iteration. In this, the data type of "**val**" should be a string or any. The syntax of the for..in loop is given below.

Syntax

```
1. for (var val in list) {      //statements    }
```

Example

```
1. let str:any = "JavaTpoint";
2. for (let index in str) {    console.log('Index of ${str[index]}: ${index}'); }
```

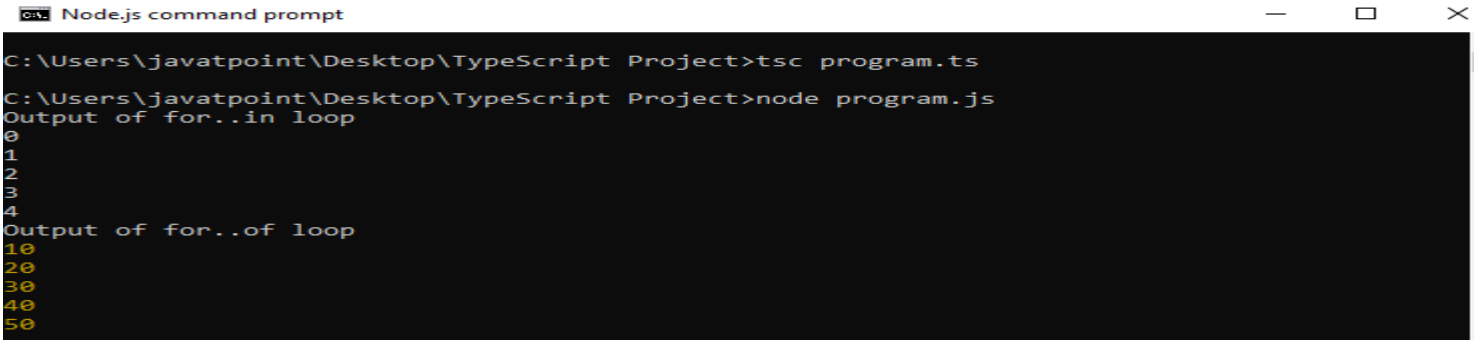
Output: Index of J: 0
Index of a: 1
Index of v: 2
Index of a: 3
Index of T: 4
Index of p: 5
Index of o: 6
Index of i: 7
Index of n: 8
Index of t: 9

for..of Vs. for..in Loop

Both the loops iterate over the lists, but their kind of iteration is different. The **for..in** loop returns a list of indexes on the object being iterated, whereas the **for..of** loop returns a list of values of the object being iterated.Below example demonstrates these differences:

```
1. let myArray = [10, 20, 30, 40, 50,];
2. console.log("Output of for..in loop ");
3. for (let index in myArray) {    console.log(index);  }
4. console.log("Output of for..of loop ");
5. for (let val of myArray) {     console.log(val);  }
```

Output:



TypeScript Operators

An Operator is a symbol which operates on a value or data. It represents a specific action on working with data. The data on which operators operates is called operand. It can be used with one or more than one values to produce a single value. All of the standard JavaScript operators are available with the TypeScript program.

Example

```
a=10;
b=10;
a + b = 20;
```

In the above example, the values '10' and '20' are known as an operand, whereas '+' and '=' are known as operators.

Operators in Typescript

In TypeScript, an operator can be classified into the following ways.

Arithmetic operators, Comparison (Relational) operators, Logical operators, Bitwise operators, Assignment operators, Ternary/conditional operator, Concatenation operator, Type Operator.

Arithmetic Operators

Arithmetic operators take numeric values as their operands, performs an action, and then returns a single numeric value. The most common arithmetic operators are addition(+), subtraction(-), multiplication(*), and division(/).

Operator	Operator_ Name	Description	Example

+	Addition	It returns an addition of the values.	let a = 20;
			let b = 30;
			let c = a + b;
			console.log(c); //
			Output:30

-	Subtraction	It returns the difference of the values.	let a = 30;
			let b = 20;
			let c = a - b;
			console.log(c); //
			Output
			10

*	Multiplication	It returns the product of the values.	let a = 30;
			let b = 20;
			let c = a * b;
			console.log(c); //
			Output
			600

/	Division	It performs the division operation, and returns the quotient.	let a = 100;
			let b = 20;
			let c = a / b;
			console.log(c); //
			Output
			5

%	Modulus	It performs the division operation and returns the remainder.	let a = 95;
			let b = 20;
			let c = a % b;
			console.log(c); //
			Output
			15

++	Increment	It is used to increments the value of the variable by one.	let a = 55;
			a++;
			console.log(a); //
			Output
			56

--	Decrement	It is used to decrements the value of the variable by one.	let a = 55;
			a--;
			console.log(a); //
			Output
			54

Comparison (Relational) Operators

The comparison operators are used to compares the two operands. These operators return a Boolean value true or false. The important comparison operators are given below.

Operator	Operator_Name	Description	Example

==	Is equal to	It checks whether the values of the two operands are equal or not.	let a = 10;
			let b = 20;
			console.log(a==b); //false
			console.log(a==10); //true
			console.log(10=='10'); //true

===	Identical(equal and of the same type)	It checks whether the type and values of the two operands are equal or not.	let a = 10;
			let b = 20;
			console.log(a===b); //false
			console.log(a===10); //true
			console.log(10==='10'); //false

!=	Not equal to	It checks whether the values of the two operands are equal or not.	let a = 10;
			let b = 20;
			console.log(a!=b); //true
			console.log(a!=10); //false
			console.log(10!='10'); //false

!==	Not identical	It checks whether the type and values of the two operands are equal or not.	let a = 10;
			let b = 20;
			console.log(a!==b); //true
			console.log(a!==10); /false
			console.log(10!=='10'); //true

>	Greater than	It checks whether the value of the left operands is greater than the value of the right operand or not.	let a = 30;
			let b = 20;
			console.log(a>b); //true
			console.log(a>30); //false
			console.log(20> 20'); //false

>=	Greater than or equal to	It checks whether the value of the left operands is greater than or equal to the value of the right operand or not.	let a = 20;
			let b = 20;
			console.log(a>=b); //true
			console.log(a>=30); //false
			console.log(20>='20'); //true

<	Less than	It checks whether the value of the left operands is less than the value of the right operand or not.	let a = 10;
			let b = 20;
			console.log(a<b); //true
			console.log(a<10); //false
			console.log(10<'10'); //false

<=	Less than or equal to	It checks whether the value of the left operands is less than or equal to the value of the right operand or not.	let a = 10;
			let b = 20;
			console.log(a<=b); //true
			console.log(a<=10); //true
			console.log(10<='10'); //true

Logical Operators

Logical operators are used for combining two or more condition into a single expression and return the Boolean result true or false. The Logical operators are given below.

Operator	Operator_Name	Description	Example

&&	Logical AND	It returns true if both the operands(expression) are true, otherwise returns false.	let a = false;
			let b = true;
			console.log(a&&b); /false
			console.log(b&&true); //true
			console.log(b&&10); //10 which is also 'true'
			console.log(a&&'10'); //false

||	Logical OR	It returns true if any of the operands(expression) are true, otherwise returns false.	let a = false;
			let b = true;
			console.log(a||b); //true
			console.log(b||true); //true
			console.log(b||10); //true
			console.log(a||'10'); //'10' which is also 'true'

!	Logical NOT	It returns the inverse result of an operand(expression).	let a = 20;
			let b = 30;
			console.log(!true); //false
			console.log(!false); //true
			console.log(!a); //false
			console.log(!b); /false
			console.log(!null); //true

Bitwise Operators

The bitwise operators perform the bitwise operations on operands. The bitwise operators are as follows.

Operator	Operator_Name	Description	Example

&	Bitwise AND	It returns the result of a Boolean AND operation on each bit of its integer arguments.	let a = 2;
			let b = 3;
			let c = a & b;
			console.log(c);//**Output 2**

|	Bitwise OR	It returns the result of a Boolean OR operation on each bit of its integer arguments.	let a = 2;
			let b = 3;
			let c = a | b;
			console.log(c);//**Output** 3

^	Bitwise XOR	It returns the result of a Boolean Exclusive OR operation on each bit of its integer arguments.	let a = 2;
			let b = 3;
			let c = a ^ b;
			console.log(c); //**Output** 1

~	Bitwise NOT	It inverts each bit in the operands.	let a = 2;
			let c = ~ a;
			console.log(c); //**Output** -3

>>	Bitwise Right Shift	The left operand's value is moved to the right by the number of bits specified in the right operand.	let a = 2;
			let b = 3;
			let c = a >> b;
			console.log(c); //**Output** 0

<<	Bitwise Left Shift	The left operand's value is moved to the left by the number of bits specified in the right operand. New bits are filled with zeroes on the right side.	let a = 2;
			let b = 3;
			let c = a << b;
			console.log(c); //**Output** 16

>>>	Bitwise Right Shift with Zero	The left operand's value is moved to the right by the number of bits specified in the right operand and zeroes are added on the left side.	let a = 3;
			let b = 4;
			let c = a >>> b;
			console.log(c); //**Output** 0

Assignment Operators

Assignment operators are used to assign a value to the variable. The left side of the assignment operator is called a variable, and the right side of the assignment operator is called a value. The data-type of the variable and value must be the same otherwise the compiler will throw an error. The assignment operators are as follows.

Operator	Operator_Name	Description	Example

=	Assign	It assigns values from right side to left side operand.	let a = 10;
			let b = 5;
			console.log("a=b:" +a); //**Output** 10

+=	Add and assign	It adds the left operand with the right operand and assigns the result to the left side operand.	let a = 10;
			let b = 5;
			let c = a += b;
			console.log(c); //**Output** 15

-=	Subtract and assign	It subtracts the right operand from the left operand and assigns the result to the left side operand.	let a = 10;
			let b = 5;
			let c = a -= b;
			console.log(c); //**Output** 5

*=	Multiply and assign	It multiplies the left operand with the right operand and assigns the result to the left side operand.	let a = 10;
			let b = 5;
			let c = a *= b;
			console.log(c); //**Output** 50

/=	Divide and assign	It divides the left operand with the right operand and assigns the result to the left side operand.	let a = 10;
			let b = 5;
			let c = a /= b;
			console.log(c); //**Output** 2

%=	Modulus and assign	It divides the left operand with the right operand and assigns the result to the left side operand.	let a = 16;
			let b = 5;
			let c = a %= b;
			console.log(c); //**Output** 1

Ternary/Conditional Operator

The conditional operator takes three operands and returns a Boolean value based on the condition, whether it is true or false. Its working is similar to an if-else statement. The conditional operator has right-to-left associativity. The syntax of a conditional operator is given below.

1. expression ? expression-1 : expression-2;
    - **expression:** It refers to the conditional expression.
    - **expression-1:** If the condition is true, expression-1 will be returned.
    - **expression-2:** If the condition is false, expression-2 will be returned.

Example

```
let num = 16;
let result = (num > 0) ? "True":"False"
console.log(result);
```

Output: True

Concatenation Operator

The concatenation (+) operator is an operator which is used to append the two string. In concatenation operation, we cannot add a space between the strings. We can concatenate multiple strings in a single statement. The following example helps us to understand the concatenation operator in TypeScript.

Example

```
1. let message = "Welcome to " + "JavaTpoint";
2. console.log("Result of String Operator: " +message);
```

Output: Result of String Operator: Welcome to JavaTpoint

Type Operators

There are a collection of operators available which can assist you when working with objects in TypeScript. Operators such as typeof, instanceof, in, and delete are the examples of Type operator. The detail explanation of these operators is given below.

Operator_ Name	Description	Example
In	It is used to check for the existence of a property on an object.	let Bike = {make: 'Honda', model: 'CLIQ', year: 2018}; console.log('make' in Bike); // **Output:**true

delete	It is used to delete the properties from the objects.	let Bike = { Company1: 'Honda',
		Company2: 'Hero',
		Company3: 'Royal Enfield'
		};
		delete Bike.Company1;
		console.log(Bike);// **Output:**{ Company2: 'Hero', Company3:
		'Royal Enfield' }

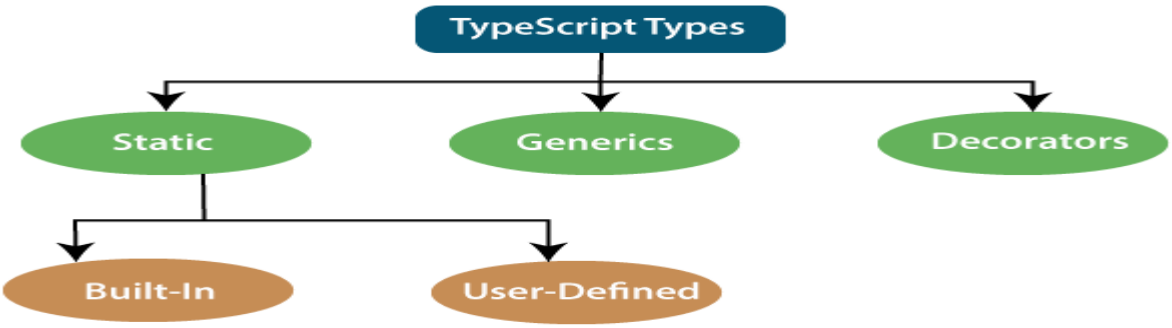
typeof	It returns the data type of the operand.	let message = "Welcome to " + "JavaTpoint"; console.log(typeof message); // **Output:**String

instanceof	It is used to check if the object is of a specified type or not.	let arr = [1, 2, 3];
		console.log(arr instanceof Array);// true
		console.log(arr instanceof String);//false

TypeScript DataTypes

The TypeScript language supports different types of values. It provides data types for the JavaScript to transform it into a strongly typed programing language. JavaScript doesn't support data types, but with the help of TypeScript, we can use the data types feature in JavaScript. TypeScript plays an important role when

the object-oriented programmer wants to use the type feature in any scripting language or object-oriented programming language. TypeScript provides data types as an optional Type System. We can classify the TypeScript data type as following.

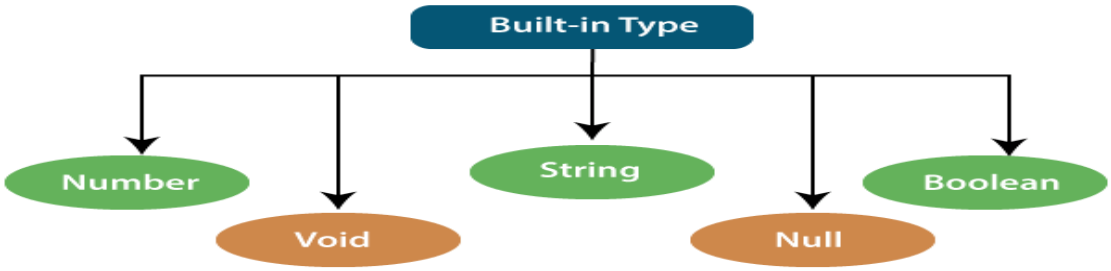


1. Static Types

In the context of type systems, static types mean "at compile time" or "without running a program." In a statically typed language, variables, parameters, and objects have types that the compiler knows at compile time. The compiler used this information to perform the type checking. Static types can be further divided into two sub-categories:

Built-in or Primitive Type

The TypeScript has five built-in data types, which are given below.



Number

Like JavaScript, all the numbers in TypeScript are stored as floating-point values. These numeric values are treated like a number data type. The numeric data type can be used to represents both integers and fractions. TypeScript also supports Binary(Base 2), Octal(Base 8), Decimal(Base 10), and Hexadecimal(Base 16) literals.

Syntax: let variablename: number = value;
Examples:-

```
let first: number = 12.0;            // number
let second: number = 0x37CF;         // hexadecimal
let third: number = 0o377 ;          // octal
let fourth: number = 0b111001;       // binary
console.log(first);          // 12.0
console.log(second);         // 14287
console.log(third);          // 255
console.log(fourth);         // 57
```

Number Properties

The Number objects have the following set of properties:

SN	Property_Name	Description
1.	MAX_VALUE	It returns the largest possible value of a number in JavaScript and can have 1.7976931348623157E+308.
2.	MIN_VALUE	It returns the smallest possible value of a number in JavaScript and can have 5E-324.
3.	NEGATIVE_INFINITY	It returns a value that is less than MIN_VALUE.
4.	POSITIVE_INFINITY	It returns a value that is greater than MAX_VALUE.
5.	NaN	When some number calculation is not representable by a valid number, then TypeScript returns a value NaN. It is equal to a value that is not a number.
6.	Prototype	It is a static property of the Number object. It is used to assign new properties and methods to the Number object in the current document.

Example

```
1. console.log("Number Properties: ");
2. console.log("A number variable can hold maximum value: " + Number.MAX_VALUE);
3. console.log("A number variable can hold minimum value: " + Number.MIN_VALUE);
4. console.log("Value of Negative Infinity: " + Number.NEGATIVE_INFINITY);
5. console.log("Value of Positive Infinity:" + Number.POSITIVE_INFINITY);
6. console.log("Example of NaN: " +Math.sqrt(-5)); // NaN
```

Output: Number Properties:

```
A number variable can hold maximum value: 1.7976931348623157e+308
A number variable can hold minimum value: 5e-324
Value of Negative Infinity: -Infinity
Value of Positive Infinity: Infinity
Example of NaN: NaN
```

Number Methods

The list of Number methods with their description is given below.

SN	Method_Name	Description
1.	toExponential()	It is used to return the exponential notation in string format.
2.	toFixed()	It is used to return the fixed-point notation in string format.

3.	toLocaleString()	It is used to convert the number into a local specific representation of the number.
4.	toPrecision()	It is used to return the string representation in exponential or fixed-point to the specified precision.
5.	toString()	It is used to return the string representation of the number in the specified base.
6.	valueOf()	It is used to return the primitive value of the number.

Example

```
let myNumber: number = 12345;
let myNumber_1: number = 12.8789;
let myNumber_2: number = 12667.976;
let myNumber_3: number = 12.5779;
let myNumber_4: number = 1234;
let myNumber_5 = new Number(123);
console.log("Number Method: toExponential()");
console.log(myNumber.toExponential());
console.log(myNumber.toExponential(2));
console.log("Number Method: toString()");
console.log(myNumber.toString());
console.log(myNumber.toString(4));
console.log("Number Method: toFixed()");
console.log(myNumber_1.toFixed());
console.log(myNumber_1.toFixed(3));
console.log("Number Method: toLocaleString()");
console.log(myNumber_2.toLocaleString()); // returns in US English
console.log("Number Method: toPrecision()");
console.log(myNumber_3.toPrecision(1));
console.log(myNumber_3.toPrecision(3));
console.log("Number Method: tovalueOf()");
console.log(myNumber_5)
console.log(myNumber_5.valueOf())
console.log(typeof myNumber)
```

Output:

```
Number Method: toExponential()
1.2345e+4
1.23e+4
Number Method: toString()
12345
3000321
Number Method: toFixed()
13
12.879
Number Method: toLocaleString()
```

```
12,667.976
Number Method: toPrecision()
1e+1
12.6
Number Method: tovalueOf()
[Number: 123]
123
number
```

String

We will use the string data type to represents the text in TypeScript. String type work with textual data. We include string literals in our scripts by enclosing them in single or double quotation marks. It also represents a sequence of Unicode characters. It embedded the expressions in the form of **$ {expr/variablename}**.

Syntax let identifier: string = " "; Or let identifier: string = ' ';

Examples

```
1. let empName: string = "Rohan";
2. let empDept: string = "IT";
3. // Before-ES6
4. let output1: string = employeeName + " works in the " + employeeDept + " department.";
5. // After-ES6
6. let output2: string = `${empName} works in the ${empDept} department.`;
7. console.log(output1);//Rohan works in the IT department.
8. console.log(output2);//Rohan works in the IT department.
```

In TypeScript, the string is an object which represents the sequence of character values. It is a primitive data type which is used to store text data. The string values are surrounded by single quotation mark or double quotation mark. An array of characters works the same as a string.

Syntax let var_name = new String(string);

Example

```
1. let uname = new String("Hello JavaTpoint");
2. console.log("Message: " +uname);
3. console.log("Length: "+uname.length);
```

Output:

```
Message: Hello JavaTpoint
Length: 16
```

There are three ways in which we can create a string.

1. Single quoted strings

It enclosed the string in a single quotation mark, which is given below.

Example var studentName: String = 'Peter';

2. Double quoted strings

It enclosed the string in double quotation marks, which is given below.

Example var studentName: String = "Peter";

3. Back-ticks strings

It is used to write an expression. We can use it to embed the expressions inside the string. It is also known as Template string. TypeScript supports Template string from ES6 version.

Example

```
let empName:string = "Rohit Sharma";
let compName:string = "JavaTpoint";
// Pre-ES6
let empDetail1: string = empName + " works in the " + compName + " company.";
// Post-ES6
let empDetail2: string = `${empName} works in the ${compName} company.`;
console.log("Before ES6: " +empDetail1);
console.log("After ES6: " +empDetail2);
```

Output:

```
Before ES6: Rohit Sharma works in the JavaTpoint company.
After ES6: Rohit Sharma works in the JavaTpoint company.
```

Multi-Line String

ES6 provides us to write the multi-line string. We can understand it from the below example.

Example

```
let multi = 'hello ' +
    'world ' +
    'my ' +
    'name ' +
    'is ' +
    'Rohit';
```

If we want that each line in the string contains "new line" characters, then we have to add "**\n**" at the end of each string.

Example

```
let multi = ' hello\n ' +
    'JavaTpoint\n ' +
    'my\n ' +
    'name\n ' +
    'is\n ' +
    'Rohit Sharma';
console.log(multi);
```

Output:

```
hello
JavaTpoint
my
```

```
name
is
Rohit Sharma
```

String Literal Type

A string literal is a sequence of characters enclosed in double quotation marks (" "). It is used to represent a sequence of character which forms a null-terminated string. It allows us to specify the exact string value specified in the "string literal type." It uses **"pipe" or " | "** symbol between different string value.

Syntax Type variableName = "value1" | "value2" | "value3"; // upto N number of values

String literal can be used in two ways-

1. Variable Assignment We can assign only allowed values to a literal type variable. Otherwise, it will give the compile-time error.

Example

```
type Pet = 'cat' | 'dog' | 'Rabbit';
let pet: Pet;
if(pet = 'cat'){
    console.log("Correct");
};
if(pet = 'Deer')
{
    console.log("compilation error");
};
```

Output:

```
Correct
compilation error
```

2. Function Parameter We can pass only defined values to literal type argument. Otherwise, it will give the compile-time error.

Example

```
type FruitsName = "Apple" | "Mango" | "Orange";
function showFruitName(fruitsName: FruitsName): void {
    console.log(fruitsName);}
showFruitName('Mango');   //OK - Print 'Mango'
//Compile Time Error
showFruitName('Banana');
```

Output:

```
Mango
Banana
```

String Methods

The list of string methods with their description is given below.

SN	Method	Description
1.	charAt()	It returns the character of the given index.
2.	concat()	It returns the combined result of two or more string.
3.	endsWith()	It is used to check whether a string ends with another string.
4.	includes()	It checks whether the string contains another string or not.
5.	indexOf()	It returns the index of the first occurrence of the specified substring from a string, otherwise returns -1.
6.	lastIndexOf()	It returns the index of the last occurrence of a value in the string.
7.	match()	It is used to match a regular expression against the given string.
8.	replace()	It replaces the matched substring with the new substring.
9.	search()	It searches for a match between a regular expression and string.
10.	slice()	It returns a section of a string.
11.	split()	It splits the string into substrings and returns an array.
12.	substring()	It returns a string between the two given indexes.
13.	toLowerCase()	It converts the all characters of a string into lower case.
14.	toUpperCase()	It converts the all characters of a string into upper case.
15.	trim()	It is used to trims the white space from the beginning and end of the string.
16.	trimLeft()	It is used to trims the white space from the left side of the string.
17.	trimRight()	It is used to trims the white space from the right side of the string.

18.	valueOf()	It returns a primitive value of the specified object.

Example

```
1. //String Initialization
2. let str1: string = 'Hello';
3. let str2: string = 'JavaTpoint';
4. //String Concatenation
5. console.log("Combined Result: " +str1.concat(str2));
6. //String charAt
7. console.log("Character At 4: " +str2.charAt(4));
8. //String indexOf
9. console.log("Index of T: " +str2.indexOf('T'));
10. //String replace
11. console.log("After Replacement: " +str1.replace('Hello', 'Welcome to'));
12. //String uppercase
13. console.log("UpperCase: " +str2.toUpperCase());
```

Output:

```
Combined Result: HelloJavaTpoint
Character At 4: T
Index of T: 4
After Replacement: Welcome to
UpperCase: JAVATPOINT
```

Boolean

The string and numeric data types can have an unlimited number of different values, whereas the Boolean data type can have only two values. They are "true" and "false." A Boolean value is a truth value which specifies whether the condition is true or not.

Syntax let variablename: boolean = Booleanvalue;

Examples: let isDone: boolean = false;

Void

A void is a return type of the functions which do not return any type of value. It is used where no data type is available. A variable of type void is not useful because we can only assign undefined or null to them. An undefined data type denotes uninitialized variable, whereas null represents a variable whose value is undefined.

Syntax let unusable: void = undefined;

Examples

```
1. function helloUser(): void {
      alert("This is a welcome message");
      }
2. let tempNum: void = undefined;
  tempNum = null;
  tempNum = 123;      //Error
```

Null

Null represents a variable whose value is undefined. Much like the void, it is not extremely useful on its own. The Null accepts the only one value, which is null. The Null keyword is used to define the Null type in TypeScript, but it is not useful because we can only assign a null value to it.

Examples

```
let num: number = null;
let bool: boolean = null;
let str: string = null;
```

Undefined

The Undefined primitive type denotes all uninitialized variables in TypeScript and JavaScript. It has only one value, which is undefined. The undefined keyword defines the undefined type in TypeScript, but it is not useful because we can only assign an undefined value to it.

Example

```
let num: number = undefined;
let bool: boolean = undefined;
let str: string = undefined;
```

Any Type

It is the "super type" of all data type in TypeScript. It is used to represents any JavaScript value. It allows us to opt-in and opt-out of type-checking during compilation. If a variable cannot be represented in any of the basic data types, then it can be declared using "**Any**" data type. Any type is useful when we do not know about the type of value (which might come from an API or 3rd party library), and we want to skip the type-checking on compile time.

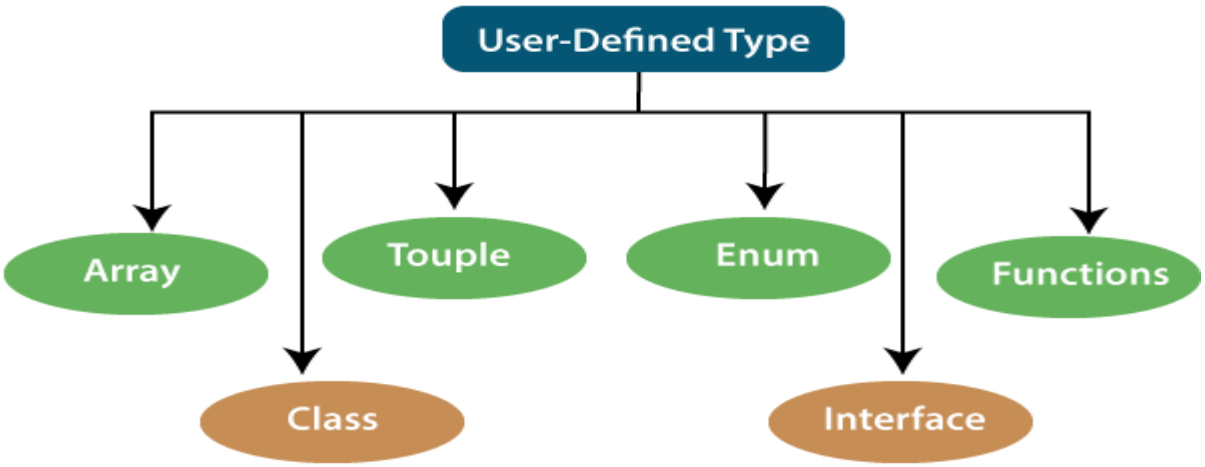
Syntax let identifier: any = value;

Examples

```
1. let val: any = 'Hi';
   val = 555;   // OK
   val = true;   // OK
2. function ProcessData(x: any, y: any) {
                return x + y;
        }
        let result: any;
        result = ProcessData("Hello ", "Any!"); //Hello Any!
        result = ProcessData(2, 3); //5
```

User-Defined DataType

TypeScript supports the following user-defined data types:



Array

An array is a collection of elements of the same data type. Like JavaScript, TypeScript also allows us to work with arrays of values. An array can be written in two ways:

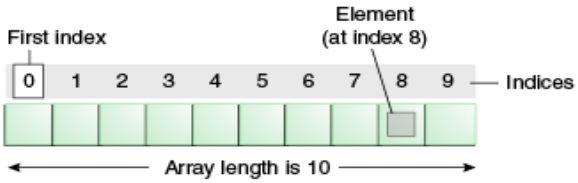
1. Use the type of the elements followed by [] to denote an array of that element type:

```
var list : number[] = [1, 3, 5];
```

2. The second way uses a generic array type:

```
var list : Array<number> = [1, 3, 5];
```

An array is a homogenous collection of similar type of elements which have a contiguous memory location.
An array is a user-defined data type.
An array is a type of data structure where we store the elements of a similar data type. In an array, we can store only a fixed set of elements. We can also use it as an object.
The array is index-based storage, where the first element stored at index 0. The below structure helps to understand the structure of an array.



Characteristics of an Array

1. An array stores elements which have the same data type.
2. Array elements stored in contiguous memory locations.
3. The storage of 2-D array elements is rowed by row in a contiguous memory location.
4. Array name represents the address of the starting element.
5. The size of an array should be initialized at the declaration time.
6. Array size should be a constant expression and not a variable.
7. We can retrieve array elements by specifying the element's corresponding index value.

Advantage

Code Optimization: An array helps to make the code optimized, which increases the speed and performance of the program. It allows us to retrieve or sort the array data more efficiently.

Random access: It provides the ability to access any data of an array in constant time (independent of its position and size). Thus, we can get any data of an array located at any index position directly.

Disadvantage

Size Limit: An array allows us to store only the fixed number of elements. Once the array is declared, we cannot alter its size. Hence, if we want to insert more element than declared, it is not possible.

Array declaration

Just like JavaScript, TypeScript also supports arrays. There are two ways to declare an array:

1. Using square brackets. let array_name[:datatype] = [val1,val2,valn..]

Example: let fruits: string[] = ['Apple', 'Orange', 'Banana'];

2. Using a generic array type. let array_name: Array<**elementType**> = [val1,val2,valn..]

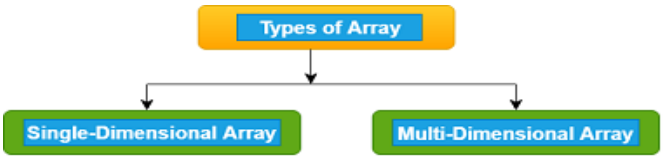
Example: let fruits: Array<**string**> = ['Apple', 'Orange', 'Banana'];

Types of the array in TypeScript

There are two types of an array:

1. Single-Dimensional Array

2. Multi-Dimensional Array



Single-Dimensional Array

A single-dimensional array is a type of linear array, which contains only one row for storing data. It has a single set of the square bracket ("[]"). We can access its elements either using row or column index.

Syntax let array_name[:datatype];

Initialization array_name = [val1,val2,valn..]

Example

```
1. let arr:number[];
2. arr = [1, 2, 3, 4]
3. console.log("Array[0]: " +arr[0]);
4. console.log("Array[1]: " +arr[1]);
```

Output:

```
Array[0]: 1
Array[1]: 2
```

Multi-Dimensional Array

A multi-dimensional array is an array which contains one or more arrays. In the multi-dimensional array, data is stored in a row and column-based index (also known as matrix form). A two-dimensional array (2-D array) is the simplest form of a multi-dimensional array.

	Column 0	Column 1	Column 2
ROW 0	a[0][0]	a[0][1]	a[0][2]
ROW 1	a[1][0]	a[1][1]	a[1][2]
ROW 2	a[2][0]	a[2][1]	a[2][2]

Syntax let arr_name:datatype[][] = [[a1,a2,a3], [b1,b2,b3]];

Initialization
let arr_name:datatype[initial_array_index][referenced_array_index] = [[val1,val2,val 3], [v1,v2,v3]];

Example

```
1. var mArray:number[][] = [[1,2,3],[5,6,7]] ;
2. console.log(mArray[0][0]);
3. console.log(mArray[0][1]);
4. console.log(mArray[0][2]);
5. console.log();
```

```
6. console.log(mArray[1][0]);
7. console.log(mArray[1][1]);
8. console.log(mArray[1][2]);
```

Output:

```
1
2
3

5
6
7
```

Array Object

Array objects allow us to store multiple values in a single variable. We can create an array by using the Array object. The Array constructor is used to pass the following arguments for array creation.

- A numeric value which represents the size of an array or
- A list of comma-separated values.

Syntax let arr_name:datatype[] = new Array(values);

Example

```
1. //array by using the Array object.
2. let arr:string[] = new Array("JavaTpoint","2200","Java","Abhishek");
3. for(var i = 0;i<arr.length;i++) {
4.    console.log(arr[i]);
5. }
```

Output:

```
JavaTpoint
2200
Java
Abhishek
```

Array Traversal by using a for...in loop

Example

```
1. let i:any;
2. let arr:string[] = ["JavaTpoint", "2300", "Java", "Abhishek"];
3. for(i in arr) {
4.    console.log(arr[i])
5. }
```

Output:

```
JavaTpoint
2300
Java
```

```
Abhishek
```

Passing Arrays to Functions

We can pass arrays to functions by specifying the array name without an index.

Example

```
let arr:string[] = new Array("JavaTpoint", "2300", "Java", "Abhishek");
//Passing arrays in function
function display(arr_values:string[]) {
   for(let i = 0;i<arr_values.length;i++) {
      console.log(arr[i]);
   }
}
//Calling arrays in function
display(arr);
```

Output:

```
JavaTpoint
2300
Java
Abhishek
```

TypeScript Spread operator

The spread operator is used to initialize arrays and objects from another array or object. We can also use it for object de-structuring. It is a part of the ES 6 version.

Example

```
let arr1 = [ 1, 2, 3];
let arr2 = [ 4, 5, 6];
//Create new array from existing array
let copyArray = [...arr1];
console.log("CopiedArray: " +copyArray);
//Create new array from existing array with more elements
let newArray = [...arr1, 7, 8];
console.log("NewArray: " +newArray);
//Create array by merging two arrays
let mergedArray = [...arr1, ...arr2];
console.log("MergedArray: " +mergedArray);
```

Output:

```
CopiedArray: 1,2,3
NewArray: 1,2,3,7,8
MergedArray: 1,2,3,4,5,6
```

Array Methods

The list of array methods with their description is given below.

SN	Method	Description
1.	concat()	It is used to joins two arrays and returns the combined result.
2.	copyWithin()	It copies a sequence of an element within the array.
3.	every()	It returns true if every element in the array satisfies the provided testing function.
4.	fill()	It fills an array with a static value from the specified start to end index.
5.	indexOf()	It returns the index of the matching element in the array, otherwise -1.
6.	includes()	It is used to check whether the array contains a certain element or not.
7.	Join()	It is used to joins all elements of an array into a string.
8.	lastIndexOf()	It returns the last index of an element in the array.
9.	Pop()	It is used to removes the last elements of the array.
10.	Push()	It is used to add new elements to the array.
11.	reverse()	It is used to reverse the order of an element in the array.
12.	Shift()	It is used to removes and returns the first element of an array.
13.	slice()	It returns the section fo an array in the new array.
14.	sort()	It is used to sort the elements of an array.
15.	splice()	It is used to add or remove the elements from an array.
16.	toString()	It returns the string representation of an array.
17.	unshift()	It is used to add one or more elements to the beginning of an array.

Tuple

The Tuple is a data type which includes two sets of values of different data types. It allows us to express an array where the type of a fixed number of elements is known, but they are not the same. For example, if we want to represent a value as a pair of a number and a string, then it can be written as:

```
// Declare a tuple
let a: [string, number];
// Initialize it
a = ["hi", 8, "how", 5]; // OK
```

TypeScript Tuples

We know that an array holds multiple values of the same data type. But sometimes, we may need to store a collection of values of different data types in a single variable. Arrays will not provide this feature, but TypeScript has a data type called Tuple to achieve this purpose. A Tuple is an array which store multiple fields belong to different data types. It is similar to the structures in the C programming language.

A tuple is a data type which can be used like any other variables. It represents the heterogeneous collection of values and can also be passed as parameters in a function call.

In abstract mathematics, the term tuple is used to denote a multi-dimensional coordinate system. JavaScript does not have tuple as data type, but tuples are available in TypeScript. The order of elements in a tuple is important.

Syntax

```
let tuple_name = [val1,val2,val3, ...val n];
```

Example

```
let arrTuple = [101, "JavaTpoint", 105, "Abhishek"];
console.log(arrTuple);
```

Output:

```
[101, 'JavaTpoint', 105, 'Abhishek']
```

We can also declare and initialize a tuple separately by initially declaring the tuple as an empty tuple in Typescript.

Example

```
let arrTuple = [];
arrTuple[0] = 101
arrTuple[1] = 105
```

Accessing tuple Elements

We can read or access the fields of a tuple by using the index, which is the same as an array. In Tuple, the index starts from zero.

Example

```
let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
console.log("Name of the Employee is : "+empTuple [0]);
console.log("Age of the Employee is : "+empTuple [1]);
console.log(empTuple [0]+" is working in "+empTuple [2]);
```

Output:

```
Name of the Employee is: Rohit Sharma
Age of the Employee is: 25
Rohit Sharma is working in JavaTpoint
```

Operations on Tuple

A tuple has two operations: Push() and Pop()

Push(): The push operation is used to add an element to the tuple.

Example

```
let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
console.log("Items: "+empTuple);
console.log("Length of Tuple Items before push: "+empTuple.length);   // returns the tuple size
empTuple.push(10001);   // append value to the tuple
console.log("Length of Tuple Items after push: "+empTuple.length);
console.log("Items: "+empTuple);
```

Output:

```
Items: Rohit Sharma, 25, JavaTpoint
Length of Tuple Items before push: 3
Length of Tuple Items after push: 4
Items: Rohit Sharma, 25, JavaTpoint, 10001
```

Pop(): The pop operation is used to remove an element from the tuple.

Example

```
let empTuple = ["Rohit Sharma", 25, "JavaTpoint", 10001];
console.log("Items: "+empTuple);
console.log("Length of Tuple Items before pop: "+empTuple.length);   // returns the tuple size
empTuple.pop();   // removed value to the tuple
console.log("Length of Tuple Items after pop: "+empTuple.length);
console.log("Items: "+empTuple);
```

Output:

```
Items: Rohit Sharma,25, JavaTpoint, 10001
Length of Tuple Items before pop: 4
Length of Tuple Items after pop: 3
Items: Rohit Sharma, 25, JavaTpoint
```

Update or Modify the Tuple Elements

Tuples are mutable, which means we can update or change the values of tuple elements. To modify the fields of a Tuple, we need to use the index of the fields and assignment operator. We can understand it with the following example.

Example

```
let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
empTuple[1] = 30;
console.log("Name of the Employee is: "+empTuple [0]);
console.log("Age of the Employee is: "+empTuple [1]);
```

```
5. console.log(empTuple [0]+" is working in "+empTuple [2]);
```

Output:

```
Name of the Employee is: Rohit Sharma
Age of the Employee is: 30
Rohit Sharma is working in JavaTpoint
```

Clear the fields of a Tuple

We cannot delete the tuple variable, but its fields could be cleared. To clear the fields of a tuple, assign it with an empty set of tuple field, which is shown in the following example.

Example

```
1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. empTuple = [];
3. console.log(empTuple);
```

Output:[]

Destructuring the Tuple

Destructuring allows us to break up the structure of an entity. TypeScript used destructuring in the context of a tuple.

Example

```
1. let empTuple = ["Rohit Sharma", 25, "JavaTpoint"];
2. let [emp, student] = empTuple;
3. console.log(emp);
4. console.log(student);
```

Output:

```
Rohit Sharma
25
```

Passing Tuple to Functions

We can pass a tuple to functions, which can be shown in the below example.

Example

```
1. //Tuple Declaration
2. let empTuple = ["JavaTpoint", 101, "Abhishek"];
3. //Passing tuples in function
4. function display(tuple_values:any[]) {
5.    for(let i = 0;i<empTuple.length;i++) {          console.log(empTuple[i]);       } }
6. //Calling tuple in function
7. display(empTuple);
```

Output:

```
JavaTpoint
101
Abhishek
```

Interface

An Interface is a structure which acts as a contract in our application. It defines the syntax for classes to follow, means a class which implements an interface is bound to implement all its members. It cannot be instantiated but can be referenced by the class which implements it. The TypeScript compiler uses interface for type-checking that is also known as "duck typing" or "structural subtyping."

Example

```
1. interface Calc {      subtract (first: number, second: number): any;  }
2. let Calculator: Calc = {      subtract(first: number, second: number) {  return first - second;      }  }
```

Class

Classes are used to create reusable components and acts as a template for creating objects. It is a logical entity which store variables and functions to perform operations. TypeScript gets support for classes from ES6. It is different from the interface which has an implementation inside it, whereas an interface does not have any implementation inside it.

Example

```
class Student
1. {   RollNo: number;
2.     Name: string;
3.     constructor(_RollNo: number, Name: string)       {   this.RollNo = _rollNo;
4.         this.Name = _name;
5.     }
6.     showDetails()     {       console.log(this.rollNo + " : " + this.name);     }
7. }
```

Enums

Enums define a set of named constant. TypeScript provides both string-based and numeric-based enums. By default, enums begin numbering their elements starting from 0, but we can also change this by manually setting the value to one of its elements. TypeScript gets support for enums from ES6.

Example

```
1. enum Color {         Red, Green, Blue  };
2. let c: Color;
3. ColorColor = Color.Green;
```

TypeScript Enums

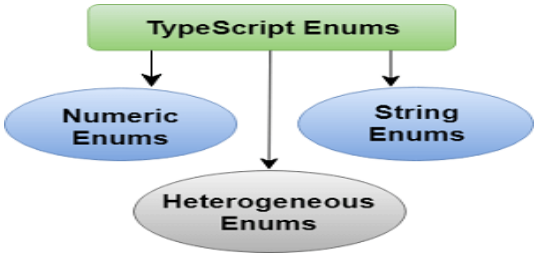
Enums stands for **Enumerations**. Enums are a new data type supported in TypeScript. It is used to define the set of **named constants**, i.e., a collection of related values. TypeScript supports both **numeric** and **string-based** enums. We can define the enums by using the **enum** keyword.

Why Enums?Enums are useful in TypeScript because of the following:

- It makes it easy to change values in the future.
- It reduces errors which are caused by transporting or mistyping a number.
- It exists only during compilation time, so it does not allocate memory.
- It saves runtime and compile-time with inline code in JavaScript.
- It allows us to create constants that we can easily relate to the program.

- It will enable developers to develop memory-efficient custom constants in JavaScript, which does not support enums, but TypeScript helps us to access them.

There are **three** types of Enums in TypeScript. These are:



Numeric Enums

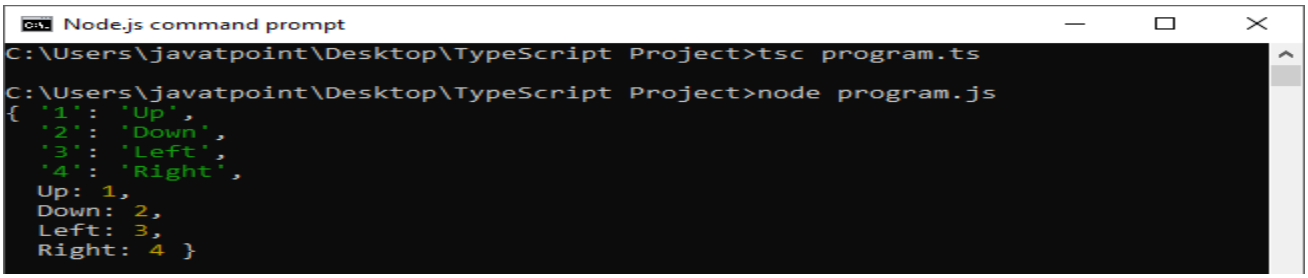
Numeric enums are **number-based** enums, which store values as numbers. It means we can assign the number to an instance of the enum.

Example

```
1. enum Direction {
2.     Up = 1,
3.     Down,
4.     Left,
5.     Right,  }
6. console.log(Direction);
```

In the above example, we have a numeric enum named **Direction**. Here, we initialize **Up** with 1, and all of the following members are **auto-incremented** from that point. It means Direction.Up has the value 1, **Down** has 2, **Left** has 3, and **Right** has 4.

Output:



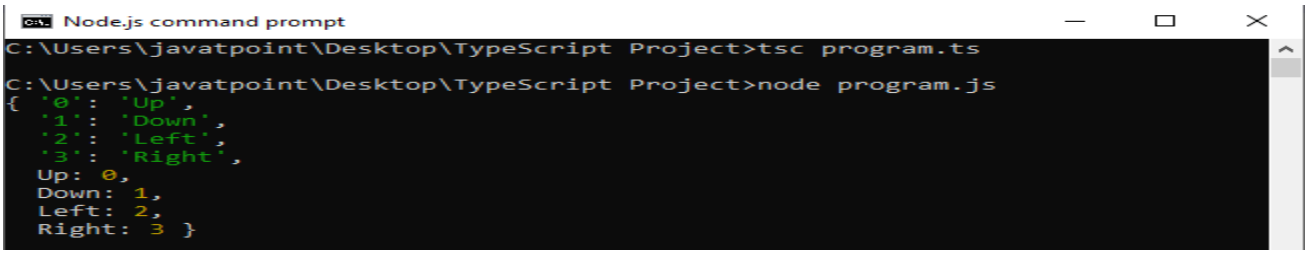
According to our need, it also allows us to leave off the initialization of enumeration. We can declare the enum without initialization as below.

```
1. enum Direction {
2.     Up,
3.     Down,
4.     Left,
5.     Right,}
6. console.log(Direction);
```

Here, Up have the value 0, and all of the following members are auto-incremented from that point. It means Direction.Up has the value 0, Down has 1, Left has 2, and Right has 3. The *auto-incrementing* behavior is

useful when there is no need to care about the member values themselves. But each value must be **distinct** from other values in the same enum.

Output:

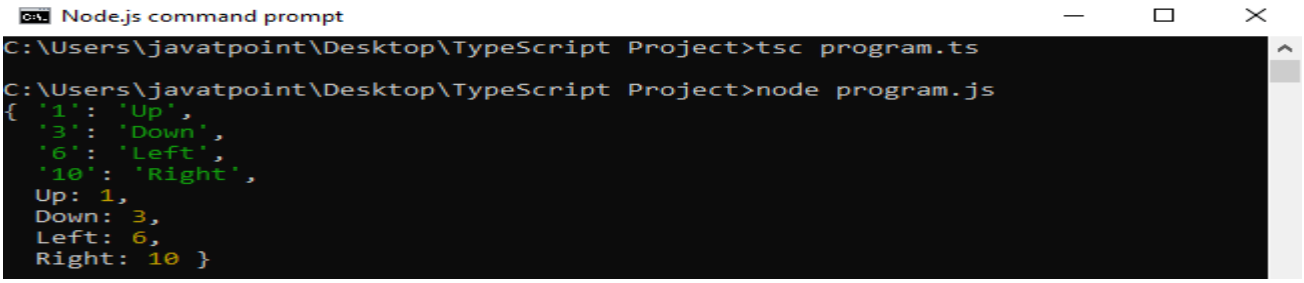


In TypeScript enums, it is not necessary to assign **sequential** values to enum members always. We can provide any values to the enum members, which looks like the below example.

Example

```
enum Direction {
    Up=1,
    Down=3,
    Left=6,
    Right=10,
}
console.log(Direction);
```

Output:



Enum as a function argument

We can also use an enum as a **function type** or **return type**, which we can see in the below example.

```
enum AppStatus {
    ACTIVE,
    INACTIVE,
    ONHOLD
}
function checkStatus(status: AppStatus): void {
    console.log(status);
}
checkStatus(AppStatus.ONHOLD);
```

In the above example, we have declared an enum **AppStatus**. Next, we create a function **checkStatus()** that takes an input parameter status which returns an enum AppStatus. In the function, we **check** for the type of status. If status name matches, we get the matched enum member.

Output: 2

Here, we can see that the value printed '2' in the last statement is not much useful in most of the scenarios. That's why it is **preferred** and **recommended** to use **string-based** enums.

String Enums

String enums are a similar concept to numeric enums, except that the enum has some subtle runtime differences. In a string enum, each enum values are **constant-initialized** with a string literal, or with another string enum member rather than numeric values. String enums do not have **auto-incrementing** behavior. The benefits of using this enum is that string enums provides better **readability**. If we were debugging a program, string enums allow us to give a meaningful and readable value when our code runs, independent of the name of the enum member itself. Consider the following example of a numeric enum, but it is represented as a string enum:

Example

```
enum AppStatus {
    ACTIVE = 'ACT',
    INACTIVE = 'INACT',
    ONHOLD = 'HLD',
    ONSTOP = 'STOP'  }
function checkStatus(status: AppStatus): void {
    console.log(status);  }
checkStatus(AppStatus.ONSTOP);
```

Output: STOP

In the above example, we have declared a string enum **AppStatus** with the same values as the numeric enum above. But string enum is different from numeric enum where string enum values are initialized with **string literals**. The difference between these enums is that the numeric enum values are auto-incremented, whereas string enum values need to be initialized individually.

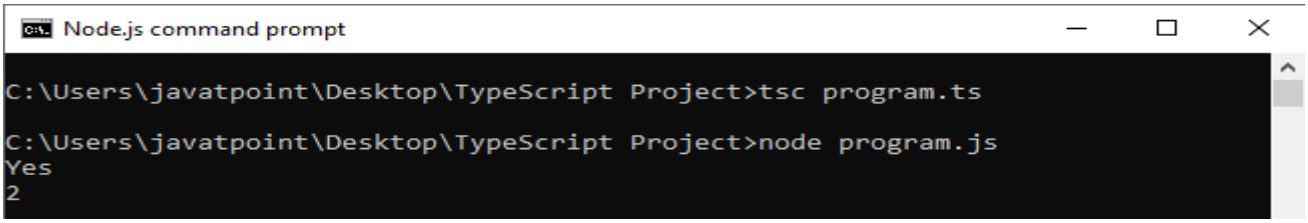
Heterogeneous Enums

The heterogeneous enums are enums, which contains both **string** and **numeric** values. But it is advised that you don't do this unless there is a need to take advantage of JavaScript runtime behavior.

Example

```
enum AppStatus {
    ACTIVE = 'Yes',
    INACTIVE = 1,
    ONHOLD = 2,
    ONSTOP = 'STOP'
}
console.log(AppStatus.ACTIVE);
console.log(AppStatus.ONHOLD);
```

Output:



Computed and constant members

We know that each enum members has a value associated with it. These values can be either constant or computed. We can consider enum member as **constant** if:

1. It is the first member of the enum and has no initializer value. In this case, it is assigned the value 0.

Example

```
1. // Name.Abhishek is constant:
2. enum Name {
3.    Abhishek   }
4. console.log(Name);
```

2. It has no initializer value, and the preceding enum member is a numeric constant. In this case, the value of the current enum member will be the value of the preceding enum member plus one.

```
1. // All enum members in 'Name' and 'Profile' are constant.
2. enum Name {
3.    Abhishek,
4.    Ravi,
5.    Ajay
6. }
7. enum Profile {
8.    Engineer=1,
9.    Leader,
10.   Businessman
11. }
```

In TypeScript, we can say that an expression is a constant enum expression if it is:

- A literal enum expression.
- A reference to the previously defined constant enum member.
- A parenthesized constant enum expression.
- It is one of the +, -, ~ unary operators which is applied to constant enum expression.
- +, -, *, /, %, <<, >>, >>>, &, |, ^ binary operators with constant enum expressions as operands.

In all other cases, the enum member is considered **computed**. The following enum example includes enum members with computed values.

```
1. enum Weekend {
2.   Friday = 1,
3.   Saturday = getDate('Dominoz'),
4.   Sunday = Saturday * 40  }
```

```
5.
6.  function getDate(day : string): number {
7.      if (day === 'Dominoz') {
8.          return 3;     }
9.  }
10. console.log(Weekend.Saturday);
11. console.log(Weekend.Sunday);
```

Output:3
120

Reverse mapping: TypeScript enums also support reverse mapping. It means we can access the value of an enum member, and also can access a member name from its value. We can understand the reverse mapping from the below example.

> 1. *Note:* ***The string enum does not support reverse mapping.***

Example

```
1.  enum Weekend {
2.    Friday = 1,
3.    Saturday,
4.    Sunday  }
5.  console.log(Weekend.Saturday);
6.  console.log(Weekend["Saturday"]);
7.  console.log(Weekend[3]);
```

Output:2
2
Sunday

Enums at runtime

Enums are the real objects which exist at runtime. We can understand it from the below example.

```
1.  enum E {
2.      A, B, C}
```

It can actually be passed around to functions, which we can see in the below example.

```
1.  function f(obj: { A: number }) {
2.      return obj.A;
3.  }  // Works, since 'E' has a property named 'A' which is a number.
4.  f(E);
```

Ambient enums

We can use ambient enums for describing the **shape** of already existing enum types.

```
1.  Declare enum Enum{
2.      X=1,
3.      Y,
4.      Z=2
5.  }
```

There is mainly one ***difference between ambient and non-ambient enums***. In regular enums, members that do not have an initializer is considered as **constant** if its preceding enum member is considered constant. But, an ambient (and non-const) enum member that does not have initializer is always considered **computed** enums.

TypeScript Union

In TypeScript, we can define a variable which can have multiple types of values. In other words, TypeScript can combine one or two different types of data (i.e., number, string, etc.) in a single type, which is called a union type. Union types are a powerful way to express a variable with multiple types. Two or more data types can be combined by using the pipe ('|') symbol between the types.

Syntax

```
1. (type1 | type2 | type3 | ..... | type-n)
```

Example

```
1. let value: number|string;
2. value = 120;
3. console.log("The Numeric value of a value is: "+value);
4. value = "Welcome to JavaTpoint";
5. console.log("The String value of a value is: "+value);
```

Output:

```
The Numeric value of the value is: 120
The String value of the value is: Welcome to JavaTpoint
```

Passing Union Type in Function Parameter

In function, we can pass a union type as a parameter. We can understand it from the below example.

Example

```
1. function display(value: (number | string))
2. {
3.     if(typeof(value) === "number")
4.         console.log('The given value is of type number.');
5.     else if(typeof(value) === "string")
6.         console.log('The given value is of type string.');
7. }
8. display(123);
9. display("ABC");
```

Output:

```
The given value is of type number.
The given value is of type of string.
```

Passing Union Type to Arrays

TypeScript allows passing a union type to an array. We can understand it from the below example.

Example

```
1. let arrType:number[]|string[];
2. let i:number;
3. arrType = [1,2,3,4];
4. console.log("Numeric type array:")
```

```
for(i = 0;i<arrType.length;i++){
   console.log(arrType[i]);  }
arrType = ["India","America","England"];
console.log("String type array:")
for(i = 0;i<arrType.length;i++){
   console.log(arrType[i]);  }
```

Output:

```
Numeric type array:
1
2
3
4
String type array:
India
America
England
```

The union can replace enums.

Enums are used to create types that contain a list of constants. By default, enums have index values (0, 1, 2, 3, etc.). We can see the enums in the following example, which contains the list of colors.

Example

```
export enum Color {RED, BLUE, WHITE}
```

Instead of enums, we can use union types and can get similar benefits in a much shorter way.

Example

```
export type Color = 'red' | 'white' | 'blue';
const myColor: Color = 'red';
console.log(myColor.toUpperCase());
```

Output:RED

TypeScript Map

TypeScript map is a new data structure added in **ES6** version of JavaScript. It allows us to store data in a **key-value pair** and remembers the original **insertion order** of the keys similar to other programming languages. In TypeScript map, we can use any value either as a **key** or as a **value**.

Create Map

We can create a map as below.

```
var map = new Map();
```

Map methods

The TypeScript map methods are listed below.

SN	Methods	Descriptions
1.	map.set(key, value)	It is used to add entries in the map.
2.	map.get(key)	It is used to retrieve entries from the map. It returns undefined if the key does not exist in the map.

3.	map.has(key)	It returns true if the key is present in the map. Otherwise, it returns false.
4.	map.delete(key)	It is used to remove the entries by the key.
5.	map.size()	It is used to returns the size of the map.
6.	map.clear()	It removes everything from the map.

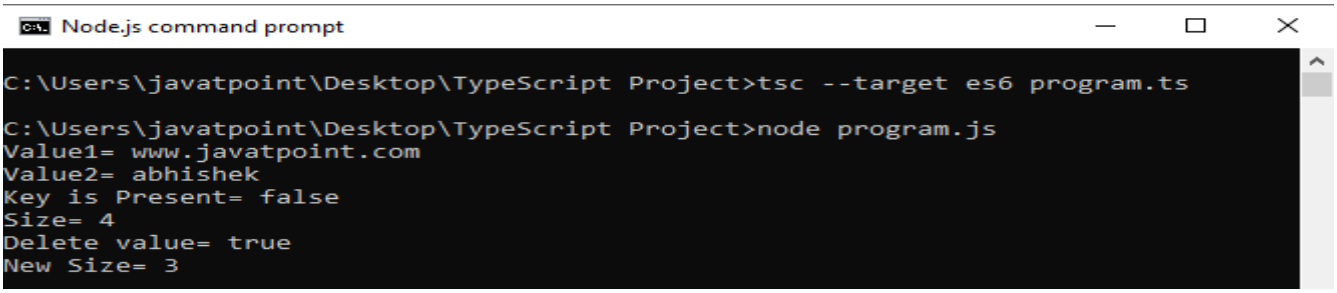
Example

We can understand the map methods from the following example.

```
1. let map = new Map();
2. map.set('1', 'abhishek');
3. map.set(1, 'www.javatpoint.com');
4. map.set(true, 'bool1');
5. map.set('2', 'ajay');
6. console.log( "Value1= " +map.get(1)   );
7. console.log("Value2= " + map.get('1') );
8. console.log( "Key is Present= " +map.has(3) );
9. console.log( "Size= " +map.size );
10. console.log( "Delete value= " +map.delete(1) );
11. console.log( "New Size= " +map.size );
```

Output:

When we execute the above code snippet, it returns the following output.



Iterating Map Data

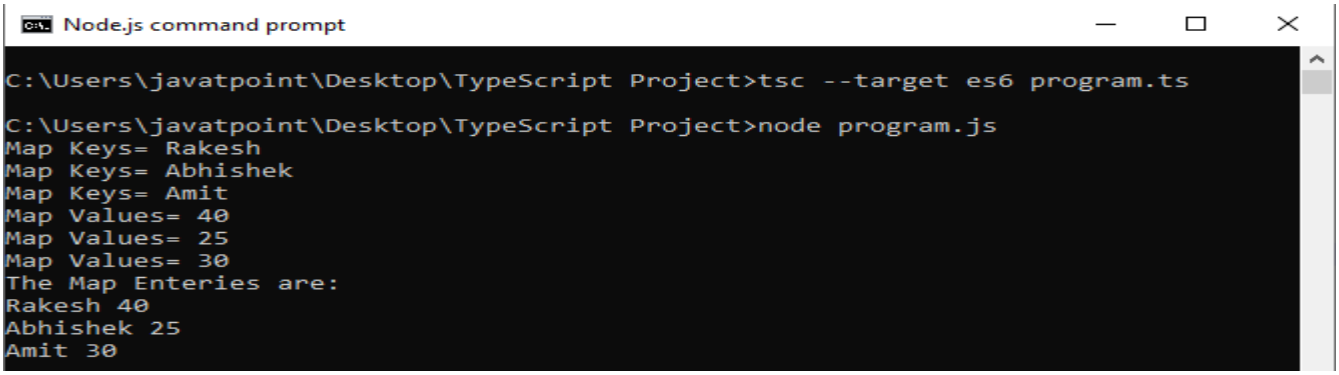
We can iterate over map keys or values or entries by using '**for...of**' loop. The following example helps to understand it more clearly.

Example

```
1. let ageMapping = new Map();
2. ageMapping.set("Rakesh", 40);
3. ageMapping.set("Abhishek", 25);
4. ageMapping.set("Amit", 30);
5. //Iterate over map keys
```

```
for (let key of ageMapping.keys()) {
    console.log("Map Keys= " +key);         }
//Iterate over map values
for (let value of ageMapping.values()) {
    console.log("Map Values= " +value);    }
console.log("The Map Enteries are: ");
//Iterate over map entries
for (let entry of ageMapping.entries()) {
    console.log(entry[0], entry[1]);   }
```

Output:



TypeScript Set

TypeScript set is a new data structure added in **ES6** version of JavaScript. It allows us to store **distinct data** (each value occur only once) into the **List** similar to other programming languages. Sets are a bit similar to **maps**, but it stores only **keys**, not the **key-value** pairs.

Create Set: We can create a set as below.

```
let mySet = new Set();
```

Set methods: The TypeScript set methods are listed below.

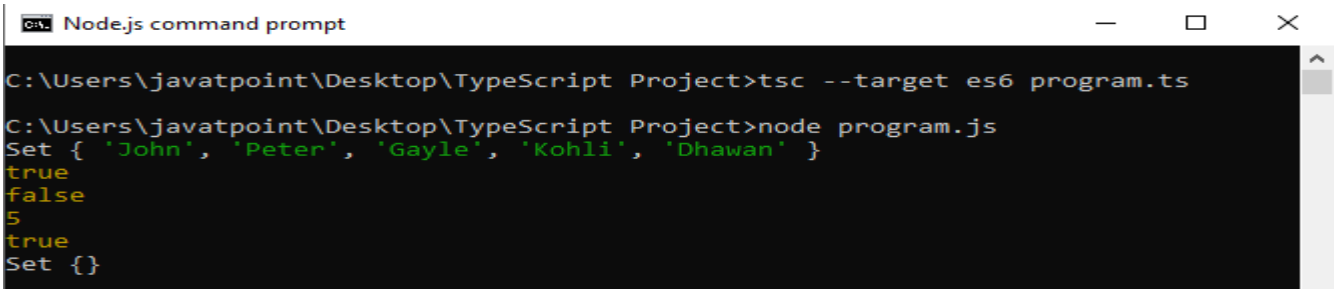
SN	Methods	Descriptions
1.	set.add(value)	It is used to add values in the set.
2.	set.has(value)	It returns true if the value is present in the set. Otherwise, it returns false.
3.	set.delete()	It is used to remove the entries from the set.
4.	set.size()	It is used to returns the size of the set.
5.	set.clear()	It removes everything from the set.

Example: We can understand the set methods from the following example.

```
let studentEntries = new Set();
```

```
studentEntries.add("John");  //Add Values
studentEntries.add("Peter");
studentEntries.add("Gayle");
studentEntries.add("Kohli");
studentEntries.add("Dhawan");
console.log(studentEntries);   //Returns Set data
console.log(studentEntries.has("Kohli"));       //Check value is present or not
console.log(studentEntries.has(10));
console.log(studentEntries.size);    //It returns size of Set
console.log(studentEntries.delete("Dhawan"));      //Delete a value from set
studentEntries.clear();   //Clear whole Set
console.log(studentEntries);  //Returns Set data after clear method
```

Output: When we execute the above code snippet, it returns the following output.



Chaining of Set Method

TypeScript set method also allows the chaining of **add()** method. We can understand it from the below example.

Example

```
let studentEntries = new Set();
//Chaining of add() method is allowed in TypeScript
studentEntries.add("John").add("Peter").add("Gayle").add("Kohli");
//Returns Set data
console.log("The List of Set values:");
console.log(studentEntries);
```

Output: The List of Set values:

Set{‘John’, ’Peter’, ’Gayle’, ‘Kohli’}

Iterating Set Data

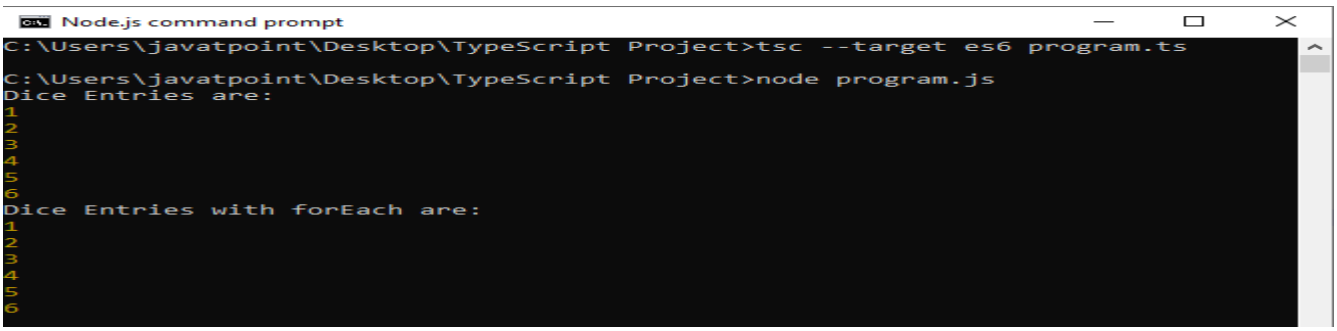
We can iterate over set values or entries by using '**for...of**' loop. The following example helps to understand it more clearly.

Example

```
let diceEntries = new Set();
diceEntries.add(1).add(2).add(3).add(4).add(5).add(6);
console.log("Dice Entries are:");   //Iterate over set entries
```

4. for (let diceNumber of diceEntries) { console.log(diceNumber); }
5. console.log("Dice Entries with forEach are:"); // Iterate set entries with forEach
6. diceEntries.forEach(function(value) { console.log(value); });

Output:



TypeScript Functions:

Typescript is used to develop functional programming and object oriented programming.

Q)what is a function?

function is a subprogram which is used to perform a specific operation. Function will gets executed when we call it single function is used to perform single operation .Functions ensure that our program is readable, maintainable, and reusable. A function declaration has a function's name, return type, and parameters.

Advantage of function: These are the main advantages of functions.

- **Code reusability:** We can call a function several times without writing the same block of code again. The code reusability saves time and reduces the program size.
- **Less coding:** Functions makes our program compact. So, we don't need to write many lines of code each time to perform a common task.
- **Easy to debug:** It makes the programmer easy to locate and isolate faulty information.

Function Aspects

There are three aspects of a function.

Function declaration: A function declaration tells the compiler about the function name, function parameters, and return type. The syntax of the function declaration is:

```
function functionName( [arg1, arg2, ...argN] );
```

Function definition: It contains the actual statements which are going to executes. It specifies what and how a specific task would be done. The syntax of the function definition is:

```
function functionName( [arg1, arg2, ...argN] ){        //code to be executed    }
```

Function call: We can call a function from anywhere in the program. The parameter/argument cannot differ in function calling and a function declaration. We must pass the same number of functions as it is declared in the function declaration. The syntax of the function call is:

```
FunctionName();
```

Different Types of functions:-

1. Named functions
2. Anonymous functions
3. Arrow Functions

named functions:- declaring the function with some name

syntax:-

function functionname([arguments]) { logic }

syntax to call the function:- functionname();

function will take i/p from use and return some value .At the time of declaring the function we have to declare parameters and at the time of calling the function we have to pass values.The no of values,order of values ,type of values that we pass must match with no of parameters,order of parameters,type of parameters.

Example//Function Definition

```
1. function display() {      console.log("Hello JavaTpoint!");  }
2. //Function Call  display();
```

Output: Hello JavaTpoint!

dynamic program **static program**

```
function Add(x:number,y:number)
   {
    console.log(x+y);
   }
    Add(8,3);
```

```
function Add()
{
 let x:number=10;
 let y:number=5;
  console.log(x+y);  }
```

Q)what are function Parameters?

The variables that are declared within the function paranthesis are called as function parameters

Q)what is the difference between document.write() and console.log() ?

document.write() :- it is used to print the output in Browser
console.log() :- it is used to print the output in console

function with some value :-

function funname(){ return somevalue;}
Ex:- function Add(x:number,y:number) { return x+y; }
note:- if the function will return value then the at the time of calling the function the result of the function must store in variable..
var sum:number=Add(8,3);
console.log(sum);
in what scenarios we use return: if the o/p of one function is input for another function then we have to use return keyword.
Parameters are of 2 Types:-
1. Formal Parameter
2. Actual Parameter
Formal Parameter :- The variables that are declared within the paranthesis at the time declaring the function are called as Formal Parameters
Actual Parameters:- The values or variables that we pass at the time of calling the function are called as Actual Parameters
Typescript supports 3 types of function parameters
1. Optional Parameter
2. Default Parameter
3. Rest Parameter
Optional Parameter:-At the time of invoking the function if the no of values that we pass will not match with no of parameters then compile time error will occur

```
function Add(x:number,y:number,z:number)
{
        console.log(x+y+z);
}
Add(6,5);  Error
```

Typescript has given a flexibility to make the parameter as optional declaring the parameter with '?' symbol.
i.e At the time declaring the function if we declare the parameter as optional (with '?')
At the time calling the function passing the value is also optional
EX:

```
function showDetails(id:number,name:string,e_mail_id?:string) {
  console.log("ID:", id, " Name:",name);
  if(e_mail_id!=undefined)
```

```
  console.log("Email-Id:",e_mail_id);
}
showDetails(101,"Virat Kohli");
showDetails(105,"Sachin","sachin@sathya.com");
```

Output:

ID: 101 Name: Virat Kohli
ID: 105 Name: Sachin
Email-Id: sachin@satya.com

NOTE: Always declare the optional parameters at the end.

EX:

```
function showDetails(id:number,name?:string,e_mail_id?:string) {
  console.log("ID:", id, "Name:",name);
  if(e_mail_id!=undefined)
  console.log("Email-Id:",e_mail_id);
}
ShowDetails(106);
showDetails(101,"Virat Kohli");
showDetails(105,"Sachin","sachin@sathya.com");
```

Output:

ID: 106 undefined
ID: 101 Name: Virat Kohli
ID: 105 Name: Sachin
Email-Id: sachin@satya.com

Default Parameter:-it is used to set default value for parameter

EX:

```
function displayName(name: string,greeting: string = "Hello") : string{ return greeting + ' ' + name + '!';}
console.log(displayName('sathya'));   //Returns "Hello sathya!"
console.log(displayName('sathya', 'Hi'));   //Returns "Hi sathya!".
console.log(displayName('sathya'));    //Returns "Hello sathya!"
```

Rest Parameter:-It is used to pass zero or more values to a function. We can declare it by prefixing the three "dot" characters ('...') before the parameter. It allows the functions to have a different number of arguments without using the arguments object. The TypeScript compiler will create an array of arguments with the rest parameter so that all array methods can work with the rest parameter. The rest parameter is useful, where we have an undetermined number of parameters.

Rules to follow in rest parameter:

We can use only one rest parameter in a function.
It must be an array type.
It must be the last parameter in a parameter list.

```
function sum(a: number,...b: number[]): number {
  let result = a;
  for (var i = 0; i < b.length; i++) {  result =result+ b[i]; }
  return result;
}
let result1 = sum(3, 5);
let result2 = sum(3, 5, 7, 9);
console.log(result1 +"\n" + result2);
```

Output: 8
24

2. Anonymous functions:-Declaring the function without any name is called as Anonymous function
Anonymous function is also called as nameless function
we will assign function to variable

named function:-	**Anonymous function:-**
function f1()	var x=function()
{	{
console.log("hi hru");	console.log("hi hru");
}	}

f1(); x();

Parameterized Anonymous function:

we need to assign Anonymous Function to variable

```
var sum:number=function(x:number,y:number)
{
 let sum:number=x+y;
console.log(x+y);  }  sum(6,3);
```

3. Arrow functions or Lambda Expression :-

Arrow function:-Arrow function is the shorthand syntax of Anonymous function

syntax:-

()=>logic;

lambda operator (or) goes to

i/p=> logic

Left hand side we have to declare parameters

Right hand side we have to write Logic

```
var x=()=>console.log("i am without parameter");
x();
====================
var sum=(x,y)=>console.log(x+y);
  sum(6,3); //9
====================
var sum=(x,y)=> x+y; //indicates returning value
 var s=sum(6,3);
  console.log(s); //9
======================
```

2. Generic

Generic is used to create a component which can work with a variety of data type rather than a single one. It allows a way to create reusable components. It ensures that the program is flexible as well as scalable in the long term. TypeScript uses generics with the type variable <T> that denotes types. The type of generic functions is just like non-generic functions, with the type parameters listed first, similarly to function declarations.

Example

```
1. function identity<T>(arg: T): T {
2.     return arg;
3. }
4. let output1 = identity<string>("myString");
5. let output2 = identity<number>( 100 );
```

3. Decorators

A decorator is a special of data type which can be attached to a class declaration, method, property, accessor, and parameter. It provides a way to add both annotations and a meta-programing syntax for classes and functions. It is used with "@" symbol.

A decorator is an experimental feature which may change in future releases. To enable support for the decorator, we must enable the **experimentalDecorators** compiler option either on the **command line** or in our tsconfig.json.

Example

```
1. function f() {
2.     console.log("f(): evaluated");
```

```
3.      return function (target, propertyKey: string, descriptor: PropertyDescriptor) {
4.          console.log("f(): called");
5.      } }
6.  class C {     @f()      method() {}  }
```

TypeScript forEach

The forEach() method is an array method which is used to execute a function on ***each item in an array***. We can use it with the JavaScript data types like Arrays, Maps, Sets, etc. It is a useful method for displaying elements in an array.

Syntax

We can declare the forEach() method as below.

array.forEach(callback[, thisObject]);

The forEach() method executes the provided **callback** once for each element present in the array in **ascending order**.

Parameter Details

1. callback: It is a function used to test for each element. The callback function accepts **three arguments**, which are given below.

Element value: It is the current value of the item.

Element index: It is the index of the current element processed in the array.

Array: It is an array which is being iterated in the forEach() method.

Note: ***These three arguments are optional.***

2. thisObject: It is an object to use as this when executing the callback.

Return Value: It will return the created array.

Example with string

```
let apps = ['WhatsApp', 'Instagram', 'Facebook'];
let playStore = [];
apps.forEach(function(item){
  playStore.push(item)
});
console.log(playStore);
```

The corresponding JavaScript code is:

```
var apps = ['WhatsApp', 'Instagram', 'Facebook'];
var playStore = [];
apps.forEach(function (item) {
    playStore.push(item);
});
console.log(playStore);
```

Output:[‘Whatsapp’ , ‘Instagram’ , ‘Facebook’]

Example with number

```
1.  var num = [5, 10, 15];
2.  num.forEach(function (value) {     console.log(value);  });
```

Output: 5
10
15

Disadvantage of forEach()

The following are the disadvantages of the use of the forEach() method:

It does not provide a way to stop or **break** the forEach() loop.

It only **works** with arrays.

OOPS:

OOPS:- Object Oriented Programming System, OOPS is a concept which is used to write programs by using classes and objects

class: class is a blueprint. The class is a group of similar entities (data). It is only a logical component and not the physical entity. For example, if you had a class called "Expensive Cars" it could have objects like Mercedes, BMW, Toyota, etc. Its properties (data) can be price or speed of these cars. While the methods may be performed with these cars are driving, reverse, braking etc.

Object: An object can be defined as an instance of a class, and there can be multiple instances of a class in a program. An Object contains both the data and the function, which operates on the data. For example - chair, bike, marker, pen, table, car, etc.

Principles of OOPS: Abstraction, Encapsulation, Inheritance, and Polymorphism

Q)what is Object Oriented Programming Language?

Any Language that supports the above 4 Principles then that language is called as Object Oriented Programming Language. Ex:- C#.net,Java,Python,Typescript

Q)what is Object Based Programming Language?

Any Language that doesn't support atleast one among the above 4 Principles then that language is called as Object Based Programming Language. Ex:- Javascript, vbscript

Q)what is Data? Data is collection of raw facts.

Data

numeric		character			boolean
integer	floatingpoint	SC	GC	ANC	true
200	2.3	a	abc	abc1234	false
5000	5.6				

Q)what is variable?

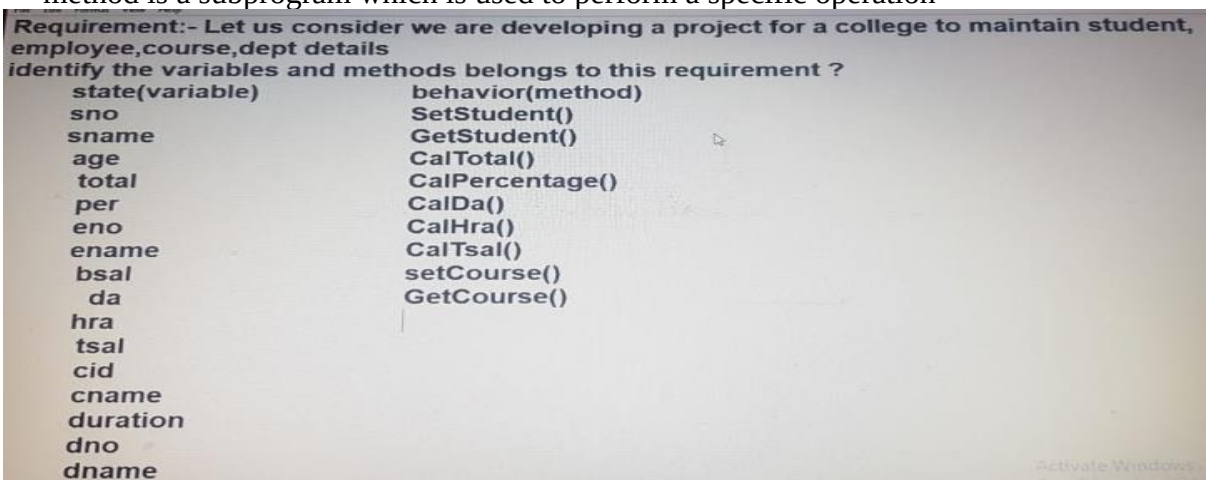
variable is the name given for a particular memory location

Q)what is the purpose of variable?

The purpose of variable is to store the value

Q)what is method?

method is a subprogram which is used to perform a specific operation



Naming Conventions in Typescript OOPS

It consists of compound words or phrases such that each word or abbreviation begins with a capital letter or first word with a lowercase letter, rest all with capital.

1. **Classes and Interfaces** :
   - Class names should be **nouns**, in mixed case with the **first** letter of each internal word capitalised. Interfaces name should also be capitalised just like class names.
   - Use whole words and must avoid acronyms and abbreviations.

   Examples:

```
class HumanBeing{ }

class KudalaBhargavReddy{ }

interface  Bicycle

class MountainBike implements Bicycle

interface Sport

class Football implements Sport
```

2. **Methods :**
   - Methods should be **verbs**, in mixed case with the **first letter lowercase** and with the first letter of each internal word capitalised.

   Examples:

```
changeGear(newValue:number){ }

speedUp(increment:number){ }
```

3. **Variables :** Variable names should be short yet meaningful.
   - Should **not** start with underscore('_') or dollar sign '$' characters.
   - **One-character variable names should be avoided** except for temporary variables.
   - Common names for temporary variables are i, j, k, m, and n for integers; c, d, and e for characters.

   Examples:

```
// variables for MountainBike class

int speed = 0;

int gear = 1;
```

4. **Constant variables:**
   - Should be **all uppercase** with words separated by underscores ("_").
   - There are various constants used in predefined classes like Float, Long, String etc.

   Examples:

```
MIN_WIDTH:number = 4;
```

Q)what is Abstraction?

it is a process of getting the required data and hiding unnecessary data

Q)what is Encapsulation?

it is a process of binding or grouping or wrapping of state and behaviour in a single container

or

it is a process of binding variable and methods in a single container in object oriented programming languages like C#.net and Java ,Typescript we can achieve Encapsulation by using class

Q) what is class?

class is a user defined datatype which consists of variables and methods

syn:-

```
class classname
 {     variables
   methods     }
```

Variables:- variable is the name given for a particular memory location

The purpose of variable is to store the value

Different types of variables are:-

1. static variable
2. instance variable
3. Method parameters
4. Local variables

1.static variable:- static variable must declare with static keyword

static variable can be accessible by using classname. They need not be accessed by using object.

Syntax to declare the static variable in class: static variablename:datatype=value;

Syntax to access the static variable in class: classname.variablename

```
class A
{  static x:number=10;
  static y:number=20; }
console.log(A.x + A.y);
```

2.instance variable :- instance variable is the instance member of a class. instance variable must declared inside the class and outside the method without any special keyword. These are also called as class variables.

```
Class Student{
   sno:number;
   sname:string;
  age:number;
}
```

3. local variable:- The variable that was declared inside the method or within the block is called as local variable. The scope of local variable is within the method or with in the block.

4.method parameters:-Method parameters are used to pass the values to instance variables at runtime.

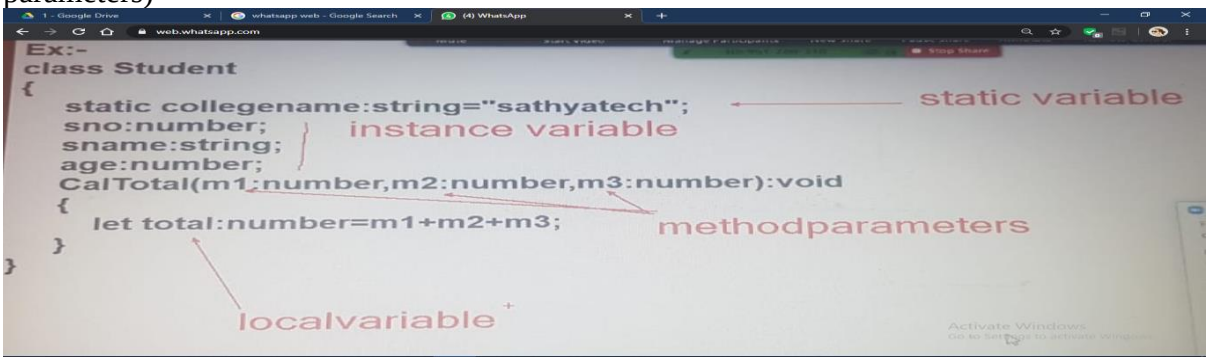
```
class Student
{   static collegename:string="sathyatech";
   sno:number;
   sname:string;
   CalTotal(m1:number,m2:number,m3:number)   {       let total:number=m1+m2+m3;   }
}
```

static variable:- collegename(if value is common for everyone then make it as static)

instance variable:- sno,sname(If value keeps changing for every individual Student then declare it as instance variable)

Method parameters :- m1,m2,m3(If a variable is just inside the method then declare it as a local variables)

Localvariable:- total(The values passed as parameters or arguments to method , they are called method parameters)



object:-object is an instance of class. instance means allocating memory for instance variables

i.e whenever we create object for a class then memory will allocate for instance variables

syntax to create object:-

```
var objectname:classname=new classname();
```

when compiler sees the **new** keyword compiler will get an information that it has to allocate memory for a class object. **classname();** will give information to which class the object is to created and how much memory is to allocated, and initializing the instance variables assigned in constructor also done.

var objectname:classname is similar to var variablename:datatype → here this store the memory location where the memory allocated for the object. Every object can be identified with some name

Q)what is Reference variable ?

Reference variable is the name given for the object.

Q)what is the purpose of Reference variable ?

The purpose of Reference variable is to access instance variables and instance methods

Every object will have 2 References

1. Default reference variable **this(automatically given by compiler to access the instance variables).**
2. user defined reference variable

this:- this is a default reference variable given for the object

Q) what is the purpose of this?

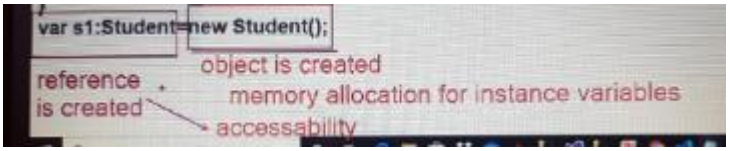
The purpose of this is to access instance variables. In Typescript if we want to access instance variables inside the method or constructor then we have to use this keyword.

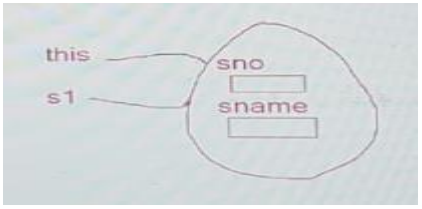
Q)what is user defined reference variable? User defined reference variable is the name given for object.

Q)what is the purpose user defined reference variable?

The purpose of user defined reference variable is to access instance methods.

```
class Student
{  sno:number;
  sname:string;
  setStudent()
  {      this.sno=101;
     this.sname="anil";
  }
  displayStudent()
 {    console.log(this.sno);
    console.log(this.sname);
 }
}
var s1:Student=new Student();
       s1.setStudent();
       s1.displayStudent();
```





Q)what is the purpose of object Creation?

Memory allocation for instance variables

Q)what is the purpose of Reference ?

Accessing the instance variables and instance methods. **observation:-** in the above program if we create multiple objects then all the objects will assign with same values

Q)How to dynamic values for different objects?

if we want to pass dynamic values for different objects then we have to use method parameters.

```
class Student{
   sno:number;
   sname:string;
   SetStudent(no:number,name:string)    {
     this.sno=no;
     this.sname=name;     }
  GetStudent()   {
  console.log(this.sno);
  console.log(this.sname);   }
}
var s1:Student=new Student();
    s1.SetStudent(101,"anil");
    s1.GetStudent();
```

```
var s2:Student=new Student();
    s2.SetStudent(102,"sunil");
    s2.GetStudent();
```
For method if return type is not mentioned, then by default its return type is void.

Methods:

Method:- Method is a subprogram which is used to perform operation
Method will gets executed when we call it:-
Syntax to declare method:-
Methodname():returntype{ logic }
Methods are divided into 2 Types:-
1. static method
2. instance method

static method:- static method must declare with static keyword
static method can be invoked(called) by using classname

```
class A {
 static Show() {   console.log("I am show");  }
}
A.Show();
```

instance method :- it is used to perform operations on instance variables
we can instance variables in instance method by using this keyword

```
class A
{
  x:number;
  setX():void  {      this.x=10;  }
  getX():void  {  console.log(this.x);  }
}
var a1:A=new A();
    a1.setX();
    a1.getX();
```

if the method is having i/p parameters and the time of calling the method we have to pass values

---

Constructor

constructor:-constructor is used to initialize the values for variables in Typescript constructor. It must be declared with **constructor** keyword. constructor will gets executed when we create object for a class. Separate call is not required for constructor. It is called special type of setter as it is initializing the instances variables as soon as the object is created.(Not separate for constructor req).

syntax to declare constructor:-

```
class classname{   constructor()  {  } }
```

Ex:-

```
class A
{
  x:number;
  y:number;
  z:number;
  constructor()  {
      this.x=10;
      this.y=5;
  }
 getSum()  {
   this.z=this.x+this.y;
   Console.log(this.z);
```

```
  }
}
var a1:A=new A();
  a1.getSum();//instance  method
```
==========

Parameterised constructor:- At the time of declaring the constructor we have to declare parameters and at the time creating object we have to pass values through constructor(classname([parameters]))

```
class A
{  x:number;//x, y and z instance variables
  y:number;
  z:number;
  constructor(a:number,b:number) {
      this.x=a;
      this.y=b;                  }
  GetSum():void  {     this.z=this.x+this.y;
     console.log(this.z);  }
}
var a1:A=new A(6,3);
  a1.GetSum();
var a2:A=new A(8,4);
     a2.GetSum();
```

Example when no values initialised to instance variables

```
class Student
{
    x:number;
    y:number;
    z:number;
   getSum()  {
     console.log("Z:",this.z);
     this.z=this.x+this.y;
     console.log("X:",this.x);
     console.log("y:",this.y);
     console.log("Z after addition:",this.z)
   }
}
var a1:Student=new Student();
  a1.getSum();
```

```
C:\Windows\System32\cmd.exe

D:\angular\OOPs>tsc Student.ts

D:\angular\OOPs>node Student.js
Z: undefined
X: undefined
y: undefined
Z after addition: NaN

D:\angular\OOPs>
```

---

class: class consists of variables and methods
variable: store the value
constructor: inject service objects within the components(DI)
method: perform operations
object: Memory allocation for variables
this: access instance variables
user defined: instance variables

Inheritance

inheritance:-it is a mechanism of creating a new class by using already existing class
It is a mechanism of obtaining variables and methods from one class to another class.(parent class to child class)
The class which is giving variables and methods is called as super class or parent class or base class and the
The class which is taking variables and methods is called as sub class or child class or derived class.

IMP NOTE:The object must always be declared for the most bottom derived/child class.

Rules to be followed while applying inheritance:

1. Minimum 2 classes must exists
2. Relationship must exist
3. Relationship can establish by using extends keyword.

```
class A{
2 variables                    No of variables in A: 2
```

3 methods No of methods in A: 3
1 constructor No of constructors in A: 1
}
Class B extends A{
1 variables No of variables in A: 3
2 methods No of methods in A: 5
1 constructor No of constructors in A: 2
}

Advantages:-
1. Reusability
2. Extensibility
3. Reimplementation

```
class person      {// this is an example for is-a relationship
name:string;
phno:number;
emailid:string;
gender:string;
dob:number;
 }
```

```
class Student extends person
{
  total
  per
  grade
}
```

```
class Employee extends person
{
  bsal
  da
  hra
  tsal
}
```

Different Types of inheritance:-
1. Single Level inheritance
2. MultiLevel inheritance
3. Hierarchical inheritance
4. Multiple inheritance
5. Hybrid inheritance

Single Level inheritance:- creating a derived class (child class) class by using properties and methods from base class(parent class). It contains only one parent class and one child class.

```
class A
{      Show()      {        console.log("i am show");       }
}
class B extends A
{
      Disp()      {        console.log("i am Disp");       }
}
var bhargav:B=new B();
bhargav.Show();
bhargav.Disp();
```

MultiLevel inheritance:- creating a derived class by using another derived class.

```
class A
{  Show()  {   console.log("i am show"); } }
class B extends A
{  Disp()  {   console.log("i am display");} }
class C extends B
{  Print()  {   console.log("i am print");  } }
var c1:C=new C();
    c1.Show();
    c1.Display();
```

```
c1.Print();
```

Hierarchical inheritance:- creating Multiple derived classes by using single base class

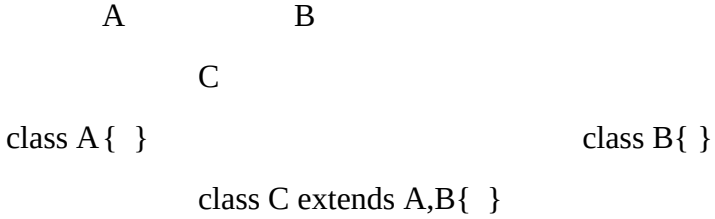
```
class A
{
  Show(){ }
}
```

```
class B extends A
{
  Disp(){}
}
var b1:B=new B();
  b1.Show();
  b1.Disp();
```

```
class C extends A
{
        Print(){ }
}
var c1:C=new C();
c1.Show();
c1.Print();
```

```
class D extends A
{
        family(){ }
}
var d1:D=new D();
d1.show();
d1.family();
```

<u>Multiple Inheritance:</u> - Creating a derived class by using multiple base classes.



Typescript doesnt support multiple inheritance by using classes. We can achieve by Multiple Inheritance using Interfaces.

Q) Why typescript will not support multiple inheritance?

If we declare same method name in multiple base classes because of inheritance, these 2 methods are available in derived class and with derived class reference if we access the method then an ambiguity problem will occur. To overcome above ambiguity problem typescript does not support multiple inheritance by using classes.

<u>Hybrid inheritance:-</u> Which can supports combination of any two inheritances except multiple inheritance.

Points to be Remembered:

1. In inheritance always create object for bottom/least derived class.
2. Whenever we create derived class object then object is not created for super class.
3. Whenever we create derived class object then memory will allocate for base/parent class and derived/child class instance variables.

   ```
   class A{
           x:number;
           y:number;   }

   class B extends A {     z:number;}

   new B();
   ```

4. Subclass object can refer in 2 ways
   a. By using super class name

      ```
      var a1:A=new B();
      ```

   b. Subclass name

      ```
      var b1:B=new B();
      ```

5. What methods we can access with super class reference?

```
class A {
  Show()  {     I am show   }
}
Class B extends A{
        Display ()  {    Iam Display  }
}
```

With super class reference we can access only super class methods.

var a1:A=new B();

a1.Show(); valid

a1.Display(); Error

Q) What is upcasting? Subclass object assigned to superclass reference.

Q) In what scenario we hav to apply upcasting?

In the scenario where single parent and multiple child exists.

The object will create at runtime by Framework or server or Runtime Environment.

		Object	Reference	Ex
Case: 1	Single parent single child	Derived	Derived	var b1:B=new B();
Case:2	Single Parent multiple child	Derived	Base class	var a1:A=new ..();

A

B C D

.. may be B,C,D at runtime object is created by server ,Framework.

Q) What is Downcasting?

Downcasting means super class reference assigned to subclass reference.

Rules to be followed while applying Down casting

1. Inheritance is mandatory
2. Upcasting is mandatory

Q) What is upcasting?

Upcasting means subclass object assigned to super class reference. After Upcasting done if we want to access subclass method without creating a new object then we have to apply Downcasting.

Q)Does constructors will participate in inheritance?

constructors will not participate in inheritance

Q)How constructors will work with inheritance?

Always constructors will call from bottom to top (child class to parent class) and execute from top to bottom (parent class to child class).

super() :- super() is used to invoke super/parent/base class constructor.

super(); should be first statement in the constructor.

```
class A{
   constructor()   {       console.log("i am A class DC");   }
}
class B extends A{
     constructor()     {         super();
        console.log("i am B class DC")     }
```

```
}
var b1:B=new B();
```
Output:
```
i am A  class DC
i am B class DC
==========================
class A{
   constructor(x:number)    {        console.log("i am A class SPC");    }
}
class B extends A{
constructor(){    super(10);
   console.log("i am B class DC") }
}
var b1:B=new B();
```
Output:
```
i am A  class SPC
i am B  class DC
==================
class Person
{
   no:number;
   name:string;
   age:number;
   constructor(x:number,y:string,z:number)
   {
      this.no=x;
      this.name=y;
      this.age=z;
   }
}
class Employee extends Person{
   bsal:number;
   constructor(eno:number,ename:string,eage:number,bsal:number)    {
      super(eno,ename,eage);
      this.bsal=bsal;
   }
   DispEmp()    {
      console.log(this.no);
      console.log(this.name);
      console.log(this.age);
      console.log(this.bsal);
   }
}
var e1:Employee=new Employee(101,"anil",24,20000);
e1.DispEmp();
```
Output:

101

Anil

24

20000

POLYMORPHISM

polymorphism:- polymorphism came from 2 greek words poly and morphos . poly means many and morphos means forms .simply polymorphism means anything that exist in multiple forms, in Object oriented programming Languages like C#.net, Java, Typescript we can achieve .Polymorphism by using Overloading and Overriding

Overloading:-it is a process of defining multiple methods with same name but with different parameters .in overloading which methods will gets executed was decided based on the values that we pass

Method:- Method is a subprogram which is used to perform specific operation
Method will gets executed when we call it
Method=Method Heading+Method Body
MethodHeading---->Accessmodifier+returntype+Methodname+parameters
MethodBody:- The logic that was written within the block
{ }
is called as Method body
MethodHeading :- The purpose of Method Heading is to identify the type of operation we are Performing

```
class A {
      public Sum(x:number,y:number):void      {      }
}
```

OverLoading:- Defyning Multiple Methods with same name and with different parameters
in overloading the Implementation of the method must written in seperate method.

```
class A   {
  public Sum(x:number,y:number):number;
  public Sum(x:number,y:number,z:number):number;
  public Sum(x:number,y:number,z?:number):number   {
      if(typeof z=='undefined')      {          return x+y;      }
     else      {       return x+y+z;      }
  }
}
var a1:A=new A();
var sum:number=a1.Sum(6,5);
console.log(sum);
var ts:number=a1.Sum(6,5,2);
console.log(ts);
```

Ex:-

```
class A{
   public Add(x:number,y:number):number;
   public Add(x:string,y:string):string;
   public Add(x:any,y:any):any  {
        if(typeof x =='number')          {
           return x+y;
        }
        else if(typeof x=='string')          {
            return x+y;
        }
   }
}
var a1:A=new A();
var sum=a1.Add(6,3);
console.log(sum); //output:9
var fname=a1.Add("sathya","tech");
console.log(fname); //output: sathya tech
```

NOTE:

```
var x:number=”abc”;   invalid
var x:any=”abc”;      valid
var x:any=10;         valid
class  {
```

variables store the value

constructors initialize the values to the instance variables

methods operation

}

object ---------->memory allocation for instance variables

this ---------->accessing the instance variables

userdefined Reference ---->accessing the methods

Overriding

Overridding is a process of defining multiple methods with same method heading and with different method body in base class and derived class

it is a process of Reimplementing the base class method in Derived class

if the base class method is not satisfying the requirement of Derived class then we have to override the base class in Derived class

Rules to be followed while working with overriding:-

1. inheritance is manadatory

2. same method heading and Different Method body in base class and Derived class

3. Always in Overriding Derived class method will gets executed

```
class A {
   public Show():void   {      console.log("i am A Show");   }    }
class B extends A  {
  public Show():void  {    console.log("i am B Show");  }
}
var b1:B=new B();
b1.Show();              o/p:- i am B show
```

Ex:-

```
  class RBI{      public ROI():void    {       console.log(3%);          }      }
class ICICI extends RBI{   public ROI():void{ console.log(5%);}   }
class AXIS extends RBI{   public ROI():void{ console.log(7%);}   }
class SBI extends RBI{   public ROI():void{ console.log(9%);}      }
var r1:RBI=new ----------();    ---------- is object created at runtime by server
var r1:RBI=new ICICI();
```

r1.ROI(); output:5%

==========

var i1:ICICI=new ICICI();

==========

class A { }

class B extends A{ }

object :- B

Reference :- B

var b1:B=new B();

==========

class A{ }

class B extends A{ } class C extends A{ }

Single Parent-multiple child Exist

object:- Derived class

Reference:- Super class

var a1:A=new ----------();

==========

class A{}

class B extends A{}

class C extends B{}

object :- C class

Reference :- C class

var c1=new C();

==========

OverLoading	**OverRiding**
1. it is a process of defyning multiple methods with same method name and with different parameters is called as Method OverLoading	1. it is a process of defyning multiple methods with same method heading and with different Method body in base class and Derived class
2. we can overload operators and methods	2. we can override methods
3.Overloading will exist in single class	3. Overriding is not possible in single class
4. inheritance is optional in overloading	4. inheritance is manadtory in Overriding
5. in Overloading which method will gets executed was decided	5. in Overriding which method will gets executed was decided based on the object creation

based on the values that we pass

==========

Interface

Q) Why Interface?

using classes multiple inheritance was not achieved to overcome that interfaces were introduced.

What is multiple Inheritance?

Multiple inheritance means creating a derived class by using multiple base classes

class A{} class B{}

class C extends A,B{} **Error**

Typescript, c#.net, java doesnot support multiple inheritance by using classes in typescript we can achieve multiple inheritance by using interfaces

Interface is a typ which consists of abstract methods and variables declarations. By default interface members are public and abstract.

In TypeScript, you can define an interface by using the keyword interface as below. By default, all the members in an interface are public, abstract.

```
interface Person {
 fullName: string;
 toString(); //method declaration only
}
```

Once the interface is defined, you can implement it in a class by following this convention: class [ClassName] implements [InterfaceName]. Let's create two classes named Employee and Customer implementing the Person interface:

```
class Employee implements Person {
 empID: string;
 fullName: string;
 constructor (eID: string, name: string) {
  this.empID = eID;
  this.fullName = name;
 }
 toString() {// method definition of above mentioned class(mandatory)
  console.log(`EMP ID of ${fullName}: ${empID}`);
 }
}
class Customer implements Person
{
 custID: string;
 fullName: string;
 constructor (cID: string, name: string)
{
  this.custID = cID;
  this.fullName = name;
 }
 toString() {
  console.log(`Customer ID of ${fullName}: ${custID}`);
 }
}
```

Let's create the instance of the classes. As both the Employee and the Customer object is of type Person, let us create it this way:

```
var employee: Person = new Employee("E001", "Kunal");
var customer: Person = new Customer("C001", "");
```

Let's call the toString() method of both the instances and observe how it prints the person detail in the screen:

```
employee.toString(); // prints employee details
customer.toString(); // prints customer details
```

Interface Extending Another Interface

In TypeScript, you can also extend an interface from another interface. This allows you to copy the members of one interface into another. So, it gives a higher degree of flexibility by separating your interfaces into reusable components. For example, the TwoWheeler interface extends the Vehicle interface as below:

```
interface Vehicle {
}
interface TwoWheeler extends Vehicle {
}
```

In TypeScript, an interface can also extend multiple interfaces. For example, let's look at the following code where the TwoWheeler interface extends the Vehicle and Engine interfaces:

```
interface Vehicle {
}
interface Engine {
}
interface TwoWheeler extends Vehicle, Engine {
}
```

Interface Extending Classes

TypeScript allows you to extend an interface from a class type. In this case, the declaration of the members of the class gets inherited to the interface but not their implementations. This is as good as a class inheriting from an interface.

```
class Vehicle {
   constructor (public brand: string) { }
   getBrandName() {
      return brand;
   }
}
class Engine {
   constructor (public manufacturer: string) { }
   getManufacturerName() {
      return manufacturer;
   }
}
interface TwoWheeler extends Vehicle, Engine {
   getBrandName();
   getManufacturerName()
}
```

In other words, you can create an interface that extends a class and then it can be implemented in another class or interface.

interface is a type which consists of public abstract members .By default interface members are public and abstract. interface will provide some set of specifications and these specifications must implement in the Derived class. interface members must implement in derived class, interface doesn't have any implementations (No Method body)

interface implementations must define in derived class

class {	interface{	abstract class{
variables	abstract methods;	concrete methods
concrete methods	variable declarations	Abstract methods
}	}	}

Interface can be implemented in another class

concrete method:- Method having body is called as concrete method

concrete method is also called as implemented method

```
class A{
  Show():void  {
   ----------
   ----------
  }
}
```

abstract method:-Method without body is called as abstract method.abstract method is also called as unimplemented method.we can declare abstract methods in interface or abstract classes. interface members must implement in Derived class.we cannot create object for interface .but we can create Reference for interface.

syn:-interface interfacename{ }

class classname implements interfacename{ }

class A{ }	abstract class A{ }	interface A{ }	interface A{ }
class B extends A{ }	class B extends A{ }	class B implements A{ }	interface B extends A{ }

A class can extend another class, A class must implement interface

```
interface A                          interface B
{  Show():void;}                     {Disp():void;}
class B implements A,B{
  Show():void  {    console.log("i am show");  }
  Disp():void  {    console.log("i am Display");  }
}
var b1:B=new B();
b1.Show();
```

Q)can we create object for interface ? no we cannot create object for interface but we create reference

Q)why we cannot create object for interface?

interface consists of unimplemented methods(method without body)

Q)can we create reference for interface? yes we can create reference

Q)what methods we can access with interface reference?

with interface reference we can access the implemented methods in the derived class

note:- Always interface and abstract class was declared as super class

Q)in what scenario we have to declare super class reference?

in the scenario where single parent and multiple child exists then we have to declare super class Reference

```
interface Shape {Draw():void; }

class Square implements Shape{ Draw():void { console.log("Draw a square");} }

class Rectangle implements Shape { Draw():void{console.log("Draw a Rectangle");} }

var s1:Shape=new ------();
```

```
interface IDbConnection {
    Open():void;
    Close():void;
}

class Oracle implements IDbConnection{
  Open():void {code to connect to oracle DB}
  Close():void { } }

class MySQL implements IDbConnection{
  Open():void{code to connect to MYSQL DB}
  Close():void { } }
```

interface is used to achieve multiple Inheritance .multiple Inheritance means creating a derived class by using multiple base classes.Typescript doesnot support multiple inheritance by using classes.we can achive multiple inheritance by using interfaces .

Q)why Typescript doesnot support Doesnot support multiple inheritance by using classes?

```
class A{
  Show():void{ Iam A show}
}

class B {
    Show():void{ Iam B show}
}

class C extends A,B { }   //This program will not working typescript
```

observation:- in the above program same method exist in both the base classes ,because of inheritance these 2 methods are available in Derived class with derived class object and Reference if we access Show() then an ambiguity problem will occur that which method must gets executed. inorder to overcome the above ambiguity problem Typescript doesnot support multiple inheritance by using classes

```
interface A{ Show():void;}
interface B{ Show():void;}

class C implements A,B{ }
```

note:- a class can extend only one class, but a class can implement multiple interfaces

Tightly coupling:- creating object for one class inside another class by using new keyword then we can say that both the classes are Tightly coupled with each other .classes and objects are dependent on each other

```
class A{ }
class B {  A a1=new A();
  Get():void {  A a2=new A();  }   }
class Square{
   Draw():void{    console.log("Draw a square");   }   }
class Rectangle {
          Draw():void{    console.log("Draw a Rectangle");   }   }
class Triangle  {
            Draw():void{    console.log("Draw a Triangle");   }     }
var t1:Triangle=new Traingle();
var s1:Shape=new Shape();
s1.GetShape();
```

```
class Shape{
        GetShape():void{
        var s1:Square=new Square();
        s1.Draw();
       var r1:Rectangle=new Rectangle();
        r1.Draw();
         t1.Draw(); }
}
```

Loosly coupling:-Loosly coupling means decrease the degree of dependencies between the classes. we can achive Loosly coup[ling by using interfaces. in Loosly coupling Mechanism we must not directly create object for one class inside another class by using new keyword.

```
interface Ishape{   Draw():void;   }
class Shape { GetShape(Ishape is):void{ is.Draw();}}
     var s1:Shape=new Shape();
    s1.GetShape(new Square());
class Square implemets IShape{   Draw():void          {    console.log("Draw a square");   }   }
class Rectangle  implements IShape{  Draw():void  {    console.log("Draw a Rectangle");   }   }
class Triangle implements IShape{  Draw():void  {    console.log("Draw a Triangle");   }  }
```

Method:- Method is a subprogram which is used to perform some operation

Method will have parameters and return type

Datatypes are of 2 Types:-Primitive (number and String), Reference type(class , interface, arrays)

Method will have Parameters:-

The no of, order of, type of values that we pass must match with no of, order of type of parameters if the parameter of the method is primitive type then at the time of calling the method we have to pass values

```
class A  {
   Show(x:number):void     {          console.log(x);      }
}
var a1:A=new A();
 a1.Show(6);
```

if the i/p parameter of the method is class then at the time of calling the method we have to pass object or Reference

```
class A  {      Show():void      {        console.log("i am show");         }    }
class B{    Get(A a1):void    {      a1.Show();    }    }
 var b1:B=new B();
  b1.Get(new A());
```

if the i/p parameter of the method is super class then then at the time of calling the method we have to pass derived class object

```
class A{   Show(){  }}
class B extends A{ }   class C extends A{ }              class D extends A{ }
class X{   Get(A a1):void   {   } }
var x1:X=new X();
   x1.Get(new B());
```

==========

if the i/p parameter of the method is interface then then at the time of calling the method we have to pass derived class object

==========

	object	Reference	Example
single parent- single child	child	child	child c=new child();
single parent -mutiple child	child create at runtime	parent	parent p=new -------();
interface--multiple childs	child create at runtime	parent	parent p=new -------();

Abstract Classes

```
interface IStudent {
  stdname:string;
  sno:number;
}
function GetStudent(s:IStudent){ console.log("sname is"+s.stdname+" "+"sno is "+s.sno);}
GetStudent({stdname:"Anil",sno:101});
```

interfaces:-interface is a type which consists of public abstract methods and variables

interface methods must implement in Derived class

By using interfaces we can achive Multiple Inheritance

By using interfaces we can achive Loosly coupling

interface is a contract or an aggrement between itself and its implemented class i.e interface will provide some set of specifications and these specifications must impelment in derive class

abstract class:- abstract class must declare with abstract keyword

abstract class consists of concrete methods and abstract methods

concrete method :- method with body is called as concrete method

concrete method is implemented method

abstract method:- method without body is called as abstract method

abstract method is unimplemented metho

abstract method must implement in derived class

class	interface	abstract class
{	{	{
variables	declare variables	variables
concrete methods	abstract methods	concrete methods
constructors	}	abstract methods
static methods		constructors
}		static methods
		}

we cannot create object for abstract class or interface

but we can create reference

```
Ex:-abstract class A
{   public abstract Show():void;
   public Display():void   {      console.log("i am Display")   }
}
class B extends A{ public Show():void {  console.log("i am show"); }     }
var b1:B=new B();
b1.Show();
b1.Display();
Ex:-   interface Icar   {
        price():number;
        CompanyName():string;
        Model():string;
   }
 abstract class Maruti implements Icar  {
      abstract price():number;
      abstract   Model():string;
      CompanyName():string        {
         return "Martuti";
      }
 }
```

```
class Dzire extends Maruti
{
    price():number{  return 10,00,000; }
    Model():string{ return "LXI";
}
```

```
class Wagonar extends Maruti
{
        price():number{  return 6,00,000; }
    Model():string{ return "VXI";   }
}
```

	class	static class	abstract class	interface
1. static variables	yes	yes	yes	no
2. instance variables	yes	no	yes	no
3. constructor	yes	no	yes	no
4. static method	yes	yes	yes	no
5. instance method	yes	no	yes	no
6. abstract method	no	no	yes	yes
7. static property	yes	yes	yes	no
8. instance property	yes	no	yes	no
9. abstract property	no	no	yes	yes
10. object	yes	no	no	no
11. Reference	yes	no	yes	yes
12. inhetritance	yes	no	yes	yes
13. multiple inheritance	no	no	no	yes

ANGULAR

Q) What is Angular?

Angular is open source and front end based web development platform, which perform best when used for building dynamic ,SPA (Single Page Application) , developed and maintained by GOOGLE.

Angular was developed by using Typescript language. We can create web application that run on Multiple Devices. Angular is not a developed version of AngularJS, Angular was completely rewritten by using Typescript language.

Angular Versions(Misko Hevery)

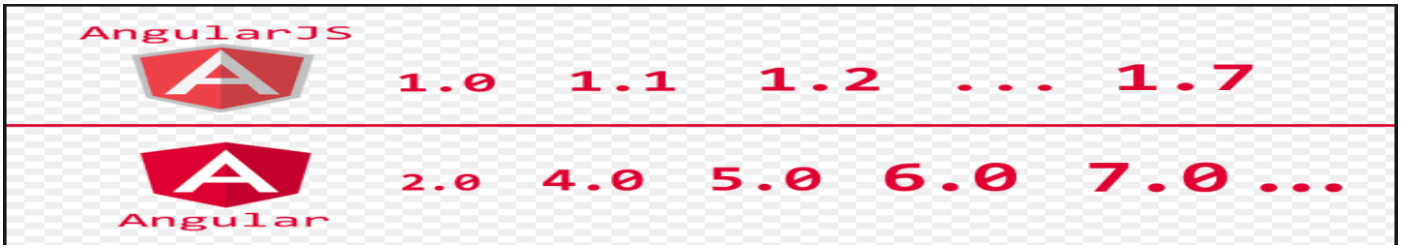
JavaScript → Browser

JQuery→ Library

Typescript → Programming Language (Microsoft)

AngularJS → Client side Java script Framework (Google) by using MVC Design pattern.

Angular 2/4/5/6/7/8/9 → TypeScript Framework (Google) by using Component Architecture(MVT)

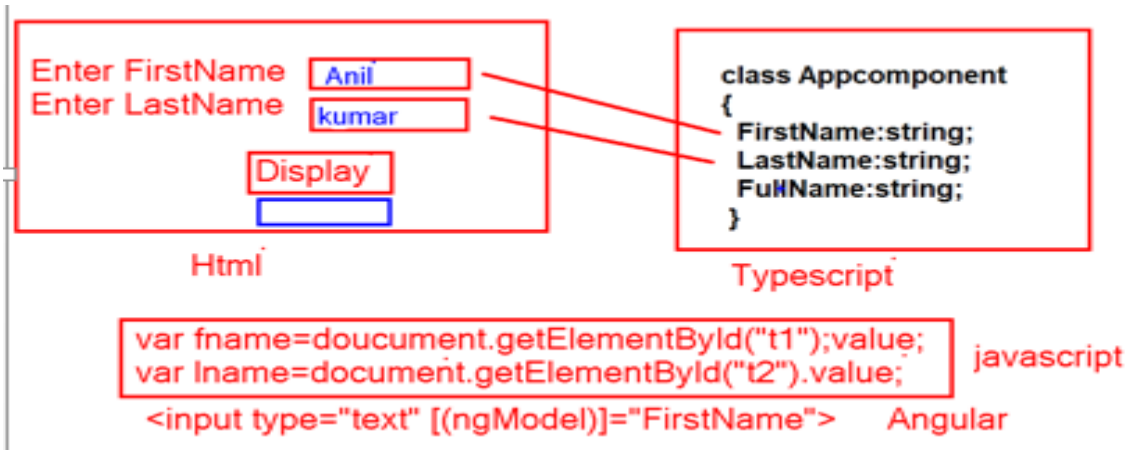


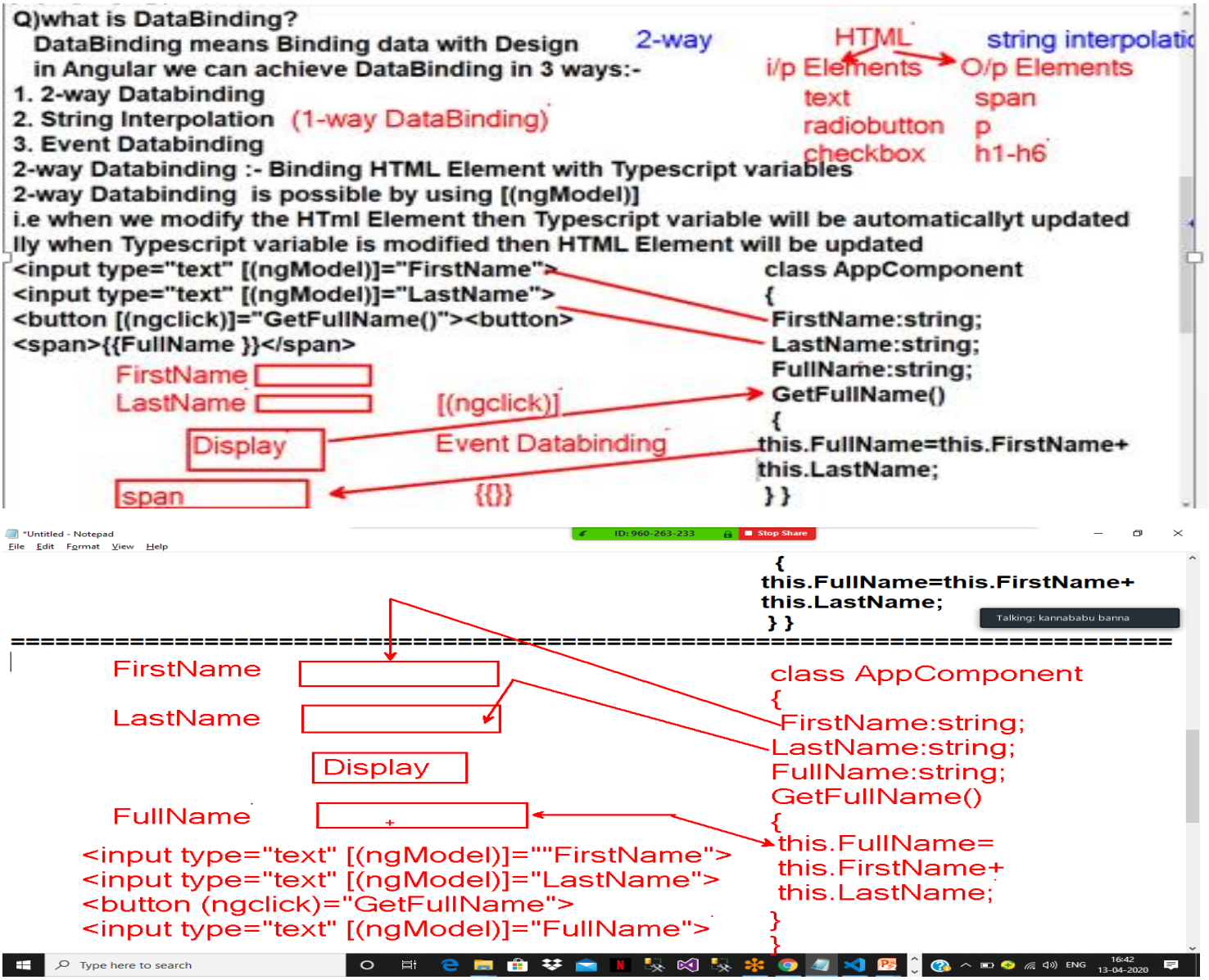
Difference between AngularJS and Angular

	AngularJS	**Angular**
Author	Google	Google
Architecture	MVC Framework	Component Based
Language	JavaScript	Typescript
Rendering	Client Side	Server Side
Data Binding	BiDirectional	BiDirectional
Latest Version	Angular 1.7	Angular 9
	It uses Controller and scope	It uses Component and Directives
Setup	Setup is not difficult	Setup is difficult
Casing	It doesn't use camel case for build in Directives like ng-model	It uses camel case for build in Directives like ngModel
Dependency Injection	Dependency Injection achieved via Controller	Dependency Injection achieved via Constructor
Mobile support	No	Yes

Q) What is Data Binding?

Data Binding means Binding HTML Element to Typescript variable.

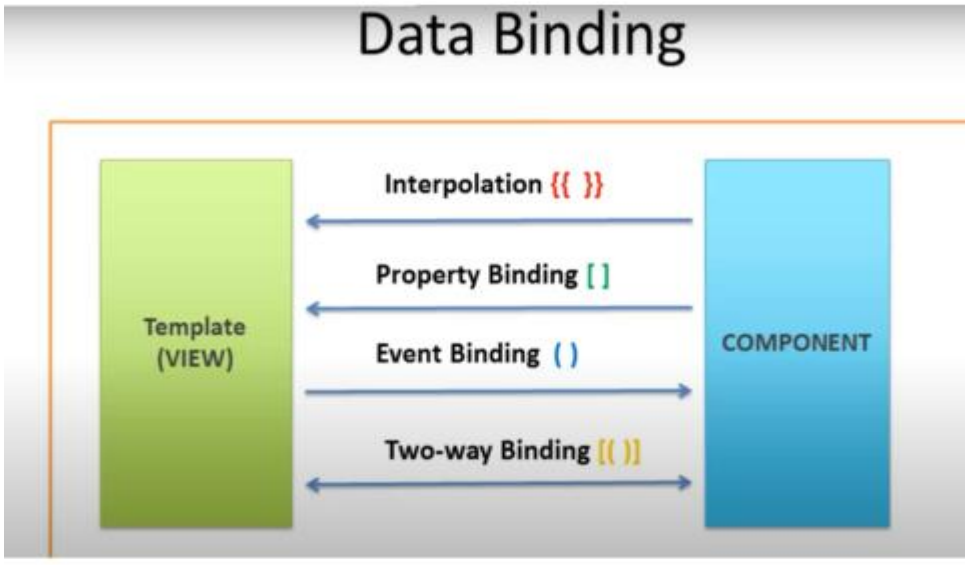




Data Binding is achieved by ngModel. ngclick is an event data binding.

ngModel and ngclick are predefined directives. (ngModel and ngclick are case sensitive)

{{ }} →String Interpolation.(1 way data binding). To display typescript data(output) into html page..



Differences between Angular and React?

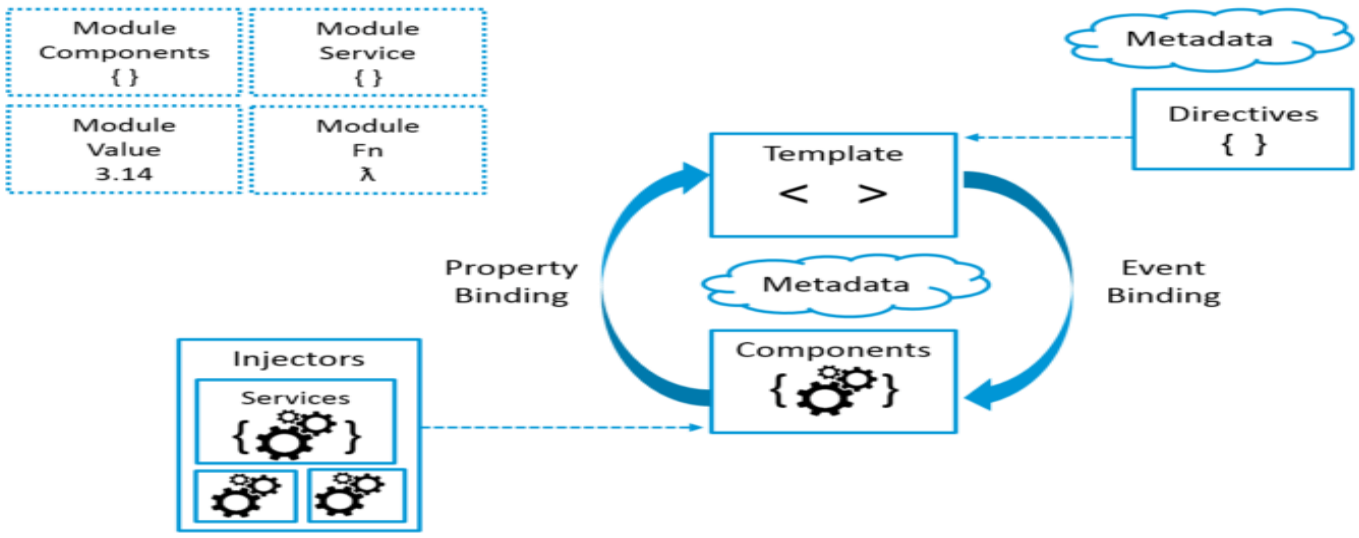
Technology	Angular	React
Created by	Google	Facebook
Technology Type	Component based Framework using Typescript.	User Interface Library with a component based architecture using java script.
Data Binding	2 way Data Binding	1 way data binding
Size	Quite large and since it needs to be shipped to the client side, it increases the initial load time	Quit small in size especially when compared with angular
Learning curve	Quite sleep ,given the number of features and options you hav in angular	Its easy to pick up and learn
Simplicity	Quite Complex	Fairly simple but takes some time to set up a project and configure everything.

Building Blocks of Angular

The main building blocks of Angular are:

- Modules: Group of components, Modules, services.
- Components: These are typescript classes which consists of various properties and methods with some decorators added to it.
- Templates: It's a html file.
- Decorators
- Data binding
- Directives
- Services
- Dependency injection
- Pipes

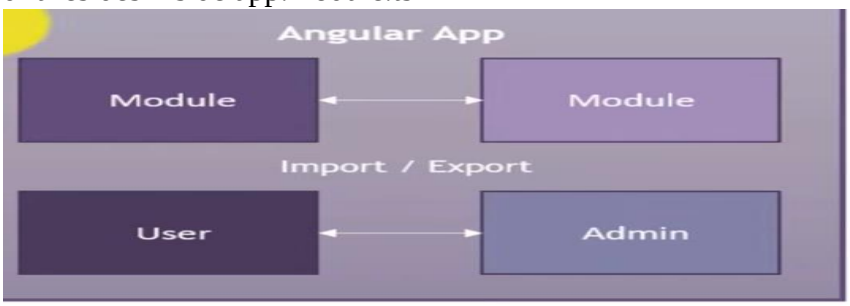
Architecture of Angular web application



What is Module?

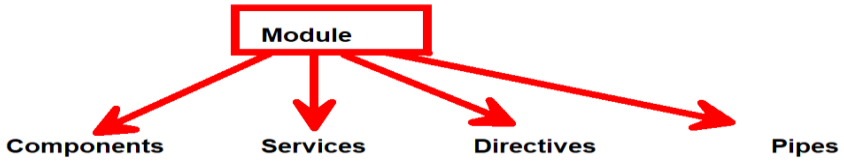
- Module in angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.

- Every Angular application has atleast one module which is root module and by convention it is AppModule and it resides inside app.module.ts



Modules in Angular

- Angular Modules are used to group components,directives,pipes,services that are related to a particular group of angular appn
- Angular apps are modular and to maintain modularity, we have *Angular modules* or you can say *NgModules*.
- Every Angular app contains at least one Angular module, i.e. the root module. Generally, it is named as *AppModule*.
- The *root module* can be the only module in a small application.
- in some applications we have multiple modules. Any angular module is a class with @NgModule decorator.



Root Module----->AppModule(launch angular application)
Featured Module--->Appn becomes large we will create multiple Featured Modules
and we will import all featured modules in RootModule
Core Module---------->include providers of HttpServices
Services in angular are singleton objects
Core module must imported in RootModule only
Shared Module---------->Reusable components,Pipes and Directives that we want to use across our entire application
common modules,formsmodule

Routing Module:- app.routing.module.ts
All application routes are available under Routing module

Advantgaes of Angular Modules:-
1. Better Code Organization
2. Code Reusability
3. Code Maintainability
4. Improve Performance

Decorators

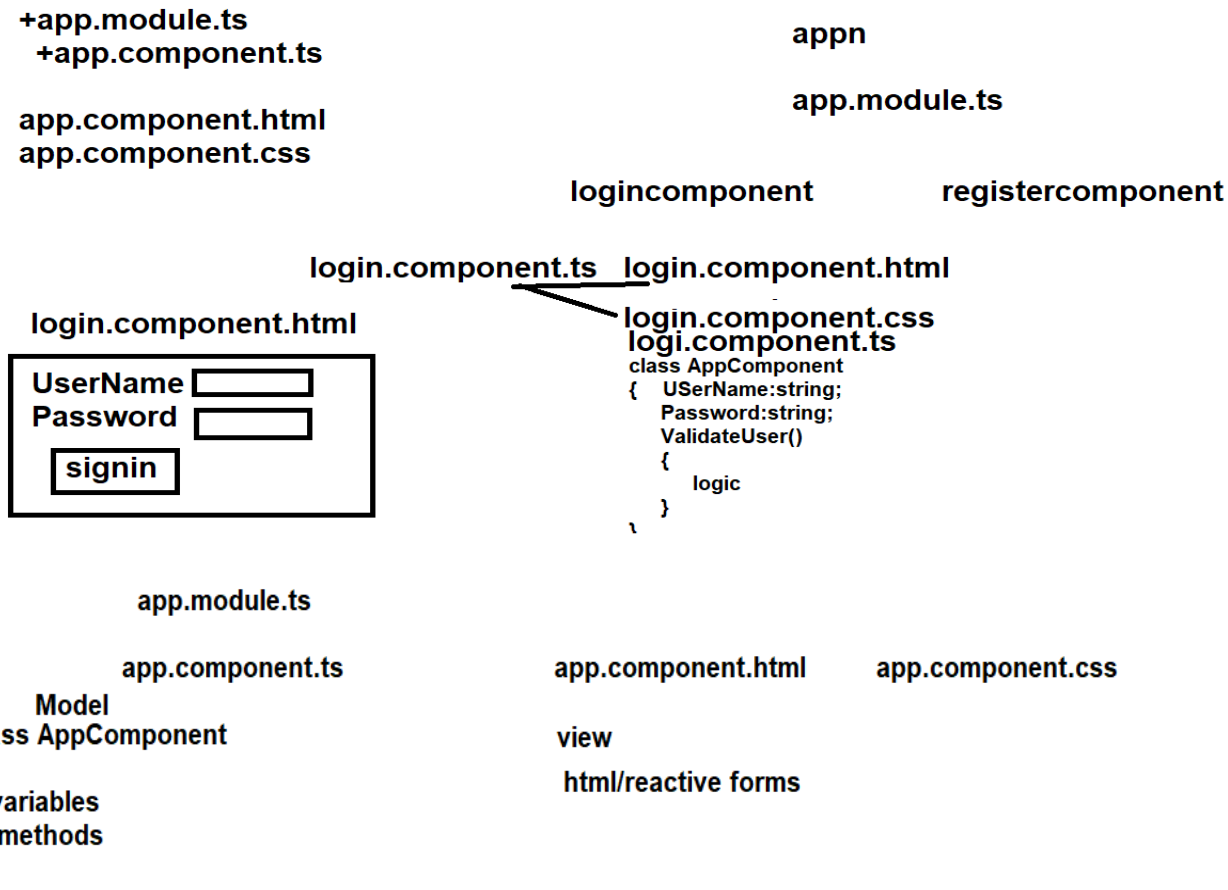
Decorators are basically used for attaching metadata to classes so that, it knows the configuration of those classes and how they should work. *NgModule* is a decorator function that takes metadata objects whose properties describe the module.

- ***declarations:*** The classes that are related to views and they belong to this module. There are three classes of Angular that can contain view: components, directives, and pipes. We will talk about them in a while.

- ***exports:*** The classes that should be accessible to the components of other modules.
- ***imports:*** Modules whose classes are needed by the component of this module.
- ***providers:*** Services present in one of the modules which are to be used in the other modules or components. Once a service is included in the providers it becomes accessible in all parts of that application
- ***bootstrap:*** The *root component* which is the main view of the application. This root module only has this property and it indicates the component that is to be bootstrapped

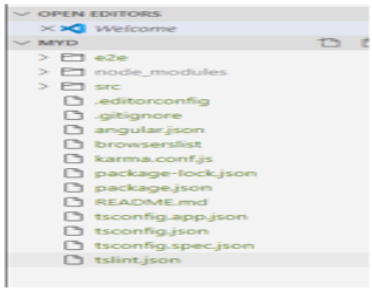
import vs export keywords in Angular

- An ***import*** is what you put in the imports property of the @NgModule decorator. It enables an Angular module to use functionality that was defined in another Angular module.
- An ***export*** what you put is the exports property of the @NgModule decorator. It enables an Angular module to expose some of its components/directives/pipes to the other modules in the applications. Without it, the components/directives/pipes defined in a module could only be used in that module.

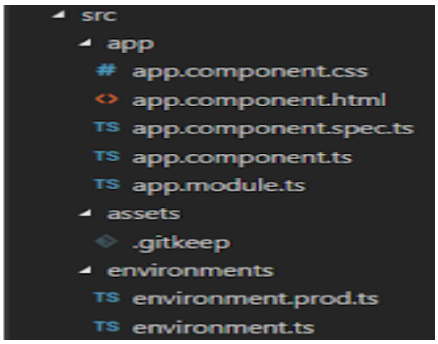


Folder structure

```
+projectname
   +e2e
   +nodemodules
   +src
       +app
          +app-routing.module.ts
          +app.component.css
          +app.component.html
          +app.component.ts
          +app.module.ts
        +assets
        +environments
        +index.html
        +karma.config.js
        +main.ts
```



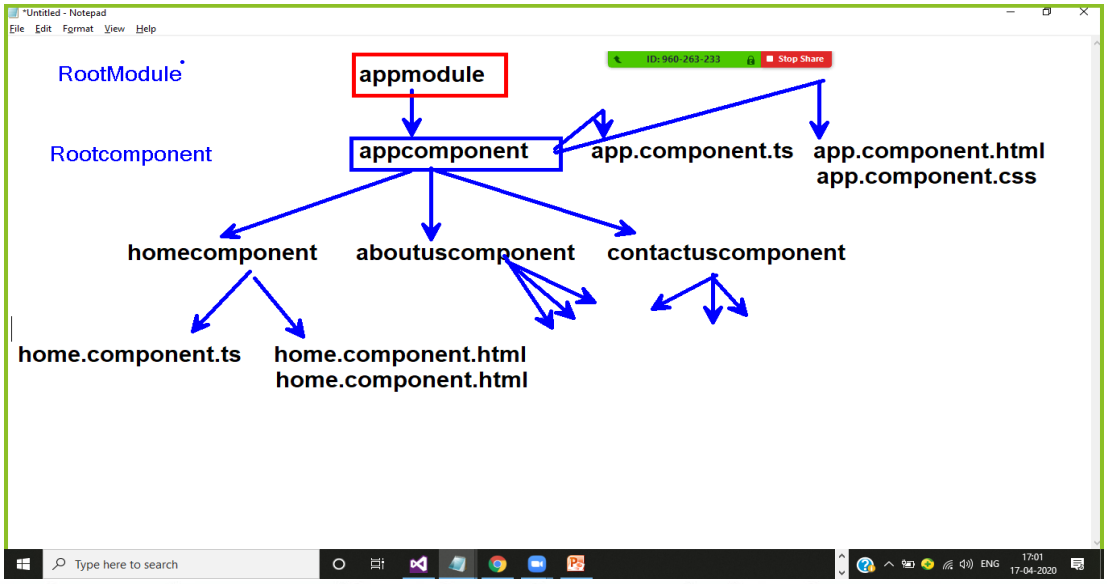
- **e2e:-** e2e stands for "end to end", this is the place where we can write the end to end test(testing related info available here). The end to end test is basically an automated test that simulates a real user.
- So, we can write code to launch our browser. Navigate to the home page of our application, then click a few links etc. are the example of end to end testing.
- **node_module:** In this folder, you can find all the third party libraries on which the application may depend. This folder is purely for development.(predefined modules are available in node modules).
- **src**
  This folder contains the actual source code for developers. This folder contains –
- ***app folder***
  Which contains all the "modules" and "components" of your application. Every application has at least one "module" and one "component".

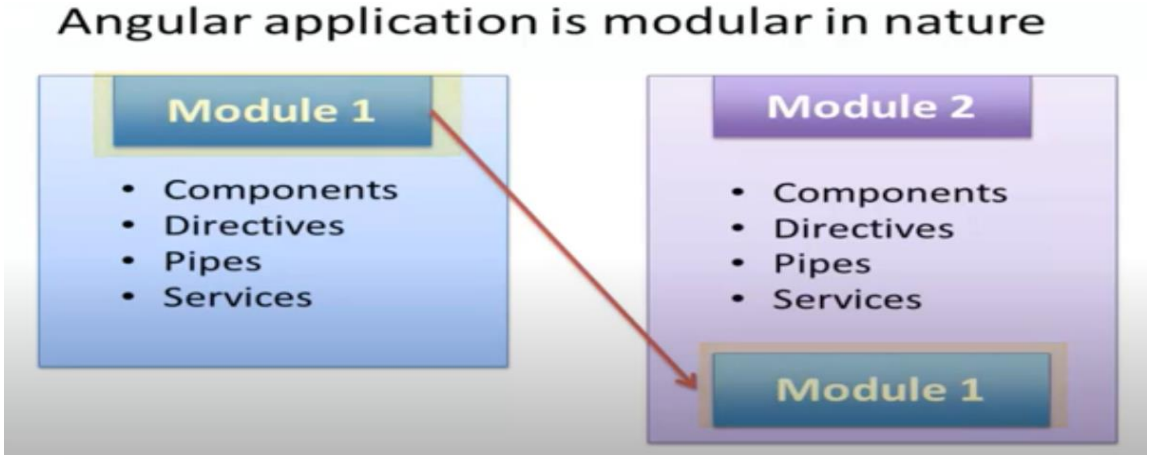


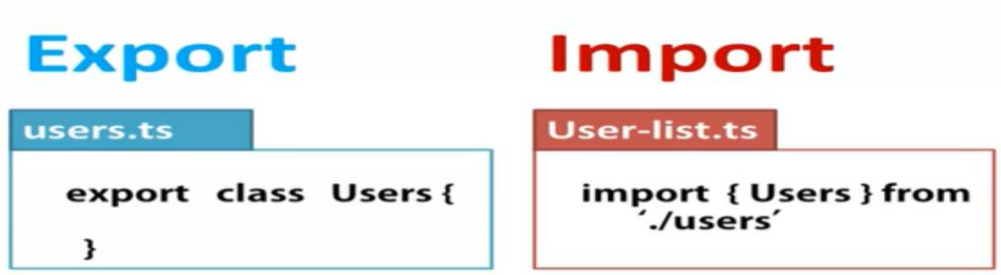
- ***assets folder***
  Where we can store static assets of our application for example images, icons etc.
- ***environment folder***
  Where we can store configuration settings for different environments. Basically, this folder contains two files,
  - *environment.prod.ts* file for the production environment.
  - *Environment.ts* file for the development environment.
- **index.html** file contains our Angular application. Here, you can not find any reference to CSS or other JS files. All other pages are dynamically inserted into the page.
- **main.ts** file is a TypeScript file. It is the starting point of our application. Here, we can bootstrap (the process of initializing or loading) our main module using bootstrapModule method like,
- *platformBrowserDynamic().bootstrapModule(AppModule);*
- **style.css** is where we can add global styles for our applications.
- **angular-cli.json** is standard configuration file of your application
- **polyfiles.ts:** to make our application run on old browser related information is kept here.
- **Karma.conf.js** file is used to store the setting of Karma i.e. test cases.
- **package.json** file is a standard file. Every node and Angular project contain this file. Basically, this file contains all information like name of the project, versions information, dependencies and dev-dependencies settings.
- **tsconfig.json** file has a bunch of settings for your TypeScript compiler, so your typescript compiler looks at the setting and based on these settings, compile your typescript code into javascript, so that browser can understand.

- **Note:**Module, Component, Services are all classes

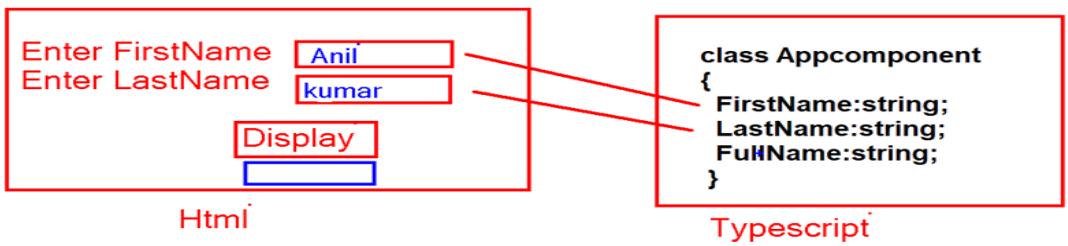
• **Module**	**Component**	**Services**
• class{ }	class{}	class{ }

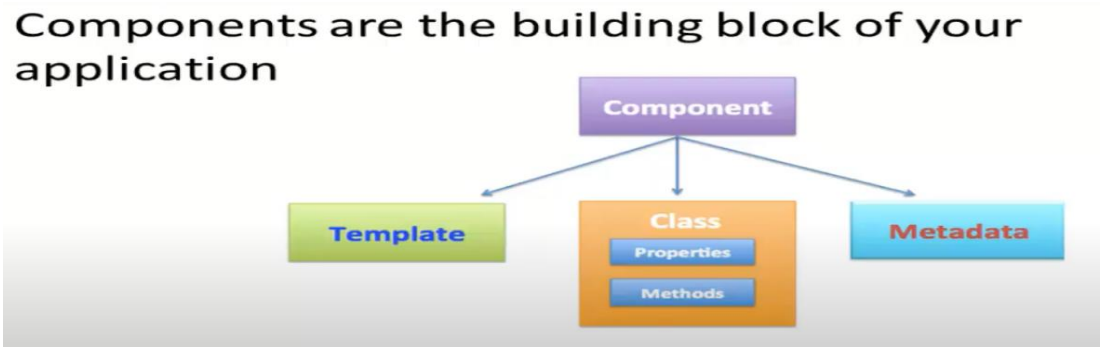
- 

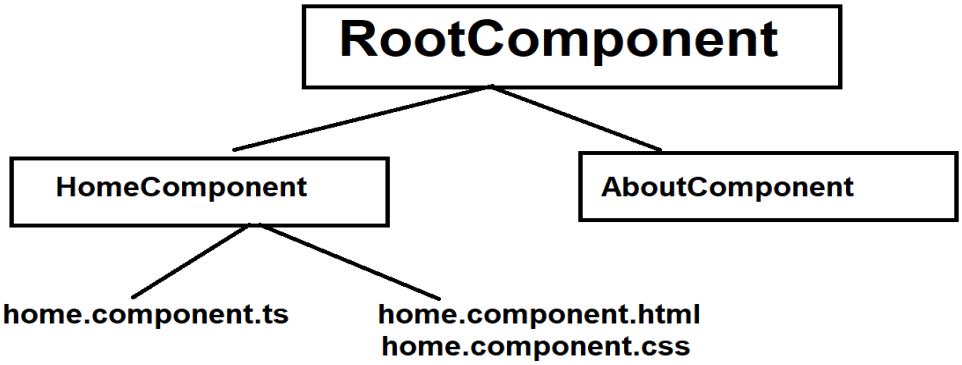
- 

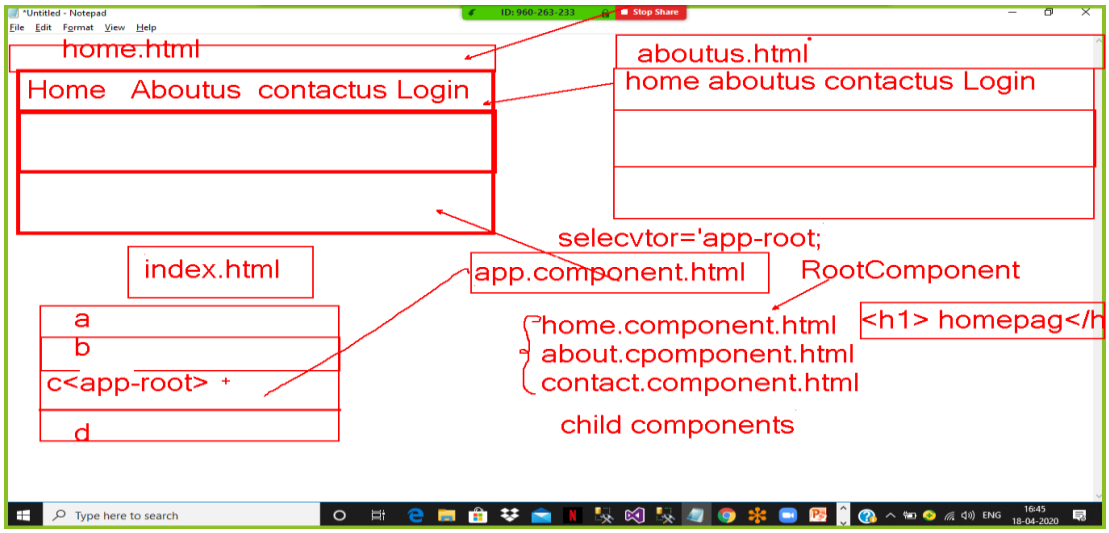
- 

- A Component needs to have a view. To define a view, you can define a template.
- A template is a form of HTML that tells angular how to render the component.

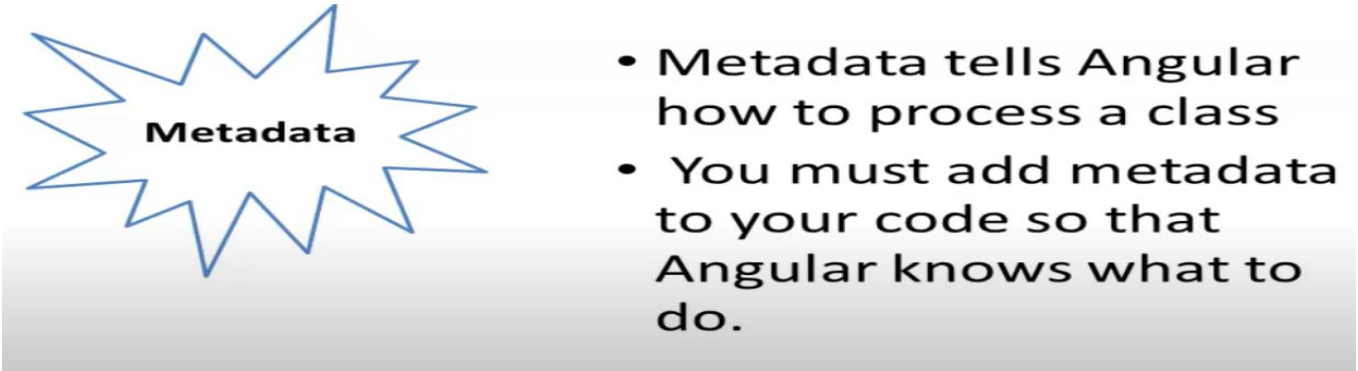
- 

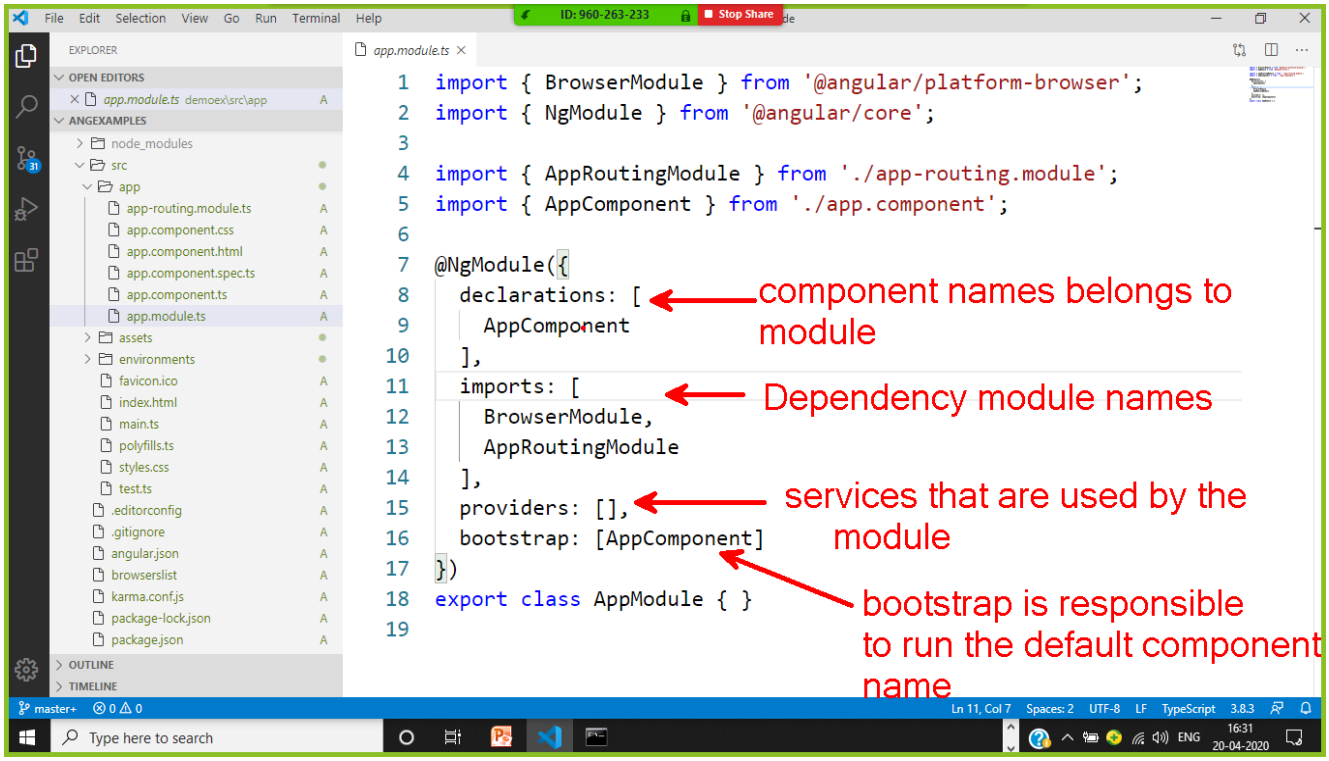
- 

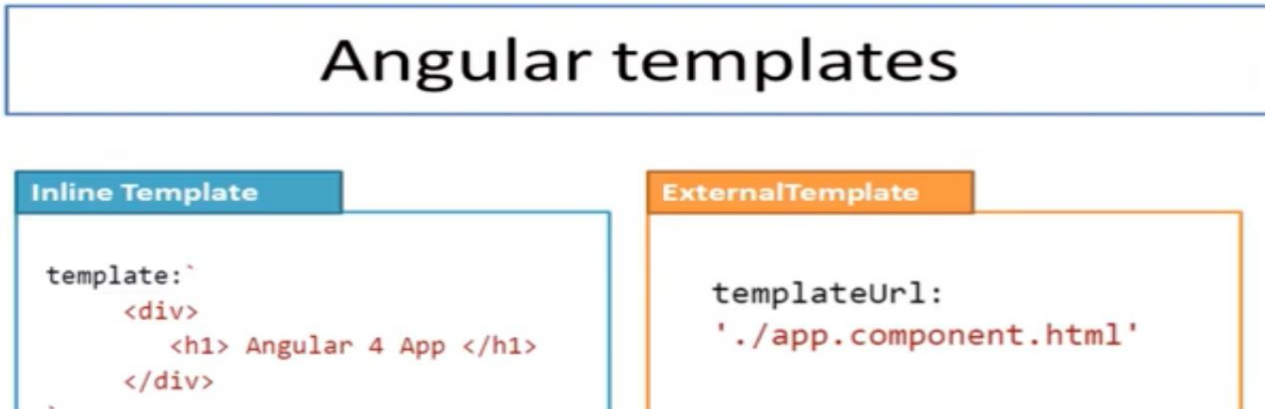





```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root,
  templateUrl: '',
  styleUrls: ['']
})
export class AboutComponent  {
  constructor() { }
  ngOnInit() {
  }
}
```



- 

- 

- @Component({
- selector: 'app-root',
- template: '<h1> Welcome to Angular </h1>',
- styleUrls: ['./app.component.css']
- })

- import { Component } from '@angular/core';
  @Component({
- selector: 'app-root',
- template: "<h1> Welcome to Angular </h1>",
- styleUrls: ['./app.component.css']
- })
- export class AppComponent {
- title = 'cp';
- }

- import { Component } from '@angular/core';
  @Component({
- selector: 'app-root',
- template: `<h1>
- Welcome to Angular
- </h1>`,
- styleUrls: ['./app.component.css']

```
})
export class AppComponent {
  title = 'cp';
}
```

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'cp';
}
```

- **Execution Flow of Angular project**

- ng serve -o
- main.ts will gets executed
- main.ts will search for RootModule(AppModule)
- app.module.ts will gets executed
- bootstrap---->will load AppComponent
- app.component.ts will execute
- @Component({
- selector: 'app-root',
- templateUrl: './app.component.html',
- styleUrls: ['./app.component.css']
- })
- app.component.html will execute and the o/p will render with app-root of index.html and display index.html o/p in Browser

---

1. install nodejs
2. check whether nodejs was installed or not
   node -v
3. check whether npm was installed or not
   npm -v
4. install typescript
   npm install -g TypeScript
5. check whether typescript was installed or not
   tsc -v
6. install angularcli
   npm install -g @angular/cli
7. check whether angularcli was install or not
   ng v
8. install visualstudio code

in order to generate modules,components,services in angular we have some commands:-

cli :- command line interface

g :- generate

m:- module

c :- component

s :- service

ng :- angular

o :- open

v :- version

- 1. command to check nodejs installation
- node -v
- 2. command to check npm installation
- npm -v
- 3. command to check typescript installation
- tsc -v
- 4. command to check angular installation
- ng v
- 5. command to create angular project
- ng new projectname
- Ex:- ng new myproject
- 6. command to create angular module
- ng g m modulename
- Ex:- ng g m student
- 7. command to create angular component
- ng g c componentname
- Ex:- ng g c home
- 8. command to create angular service
- ng g s servicename
- Ex:- ng g s login
- 9. command to run angular project
- ng serve -o
- whenever we run ng serve -o command then node.js run the angular project

Creating Angular Project

1. goto----> F Drive and create a folder with name Ag7AM

2. open node.js command prompt

3. change the Drive D:

4. change the Directory: cd ag7am

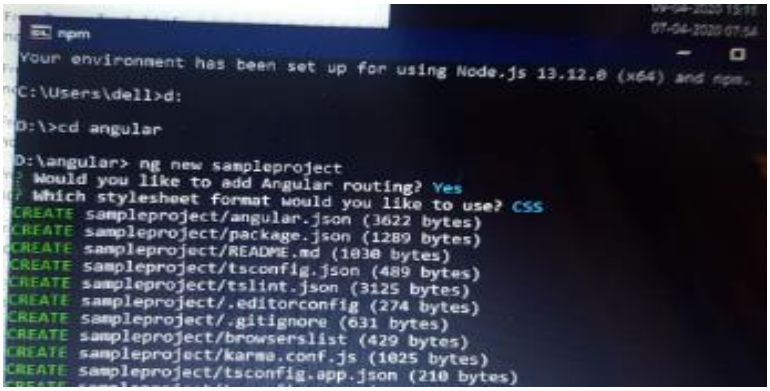
5. create angular project

   ng new projectname

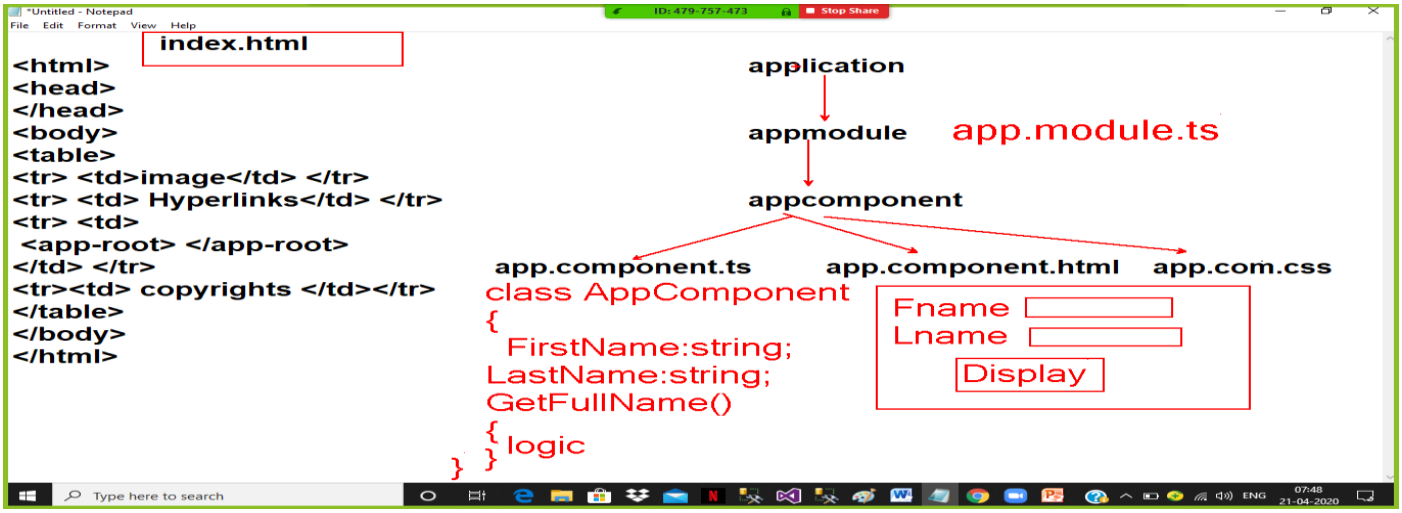
   ng new sampleproject

6. open visualstudiocode and open folder : F:\Ag7am\sampleproject

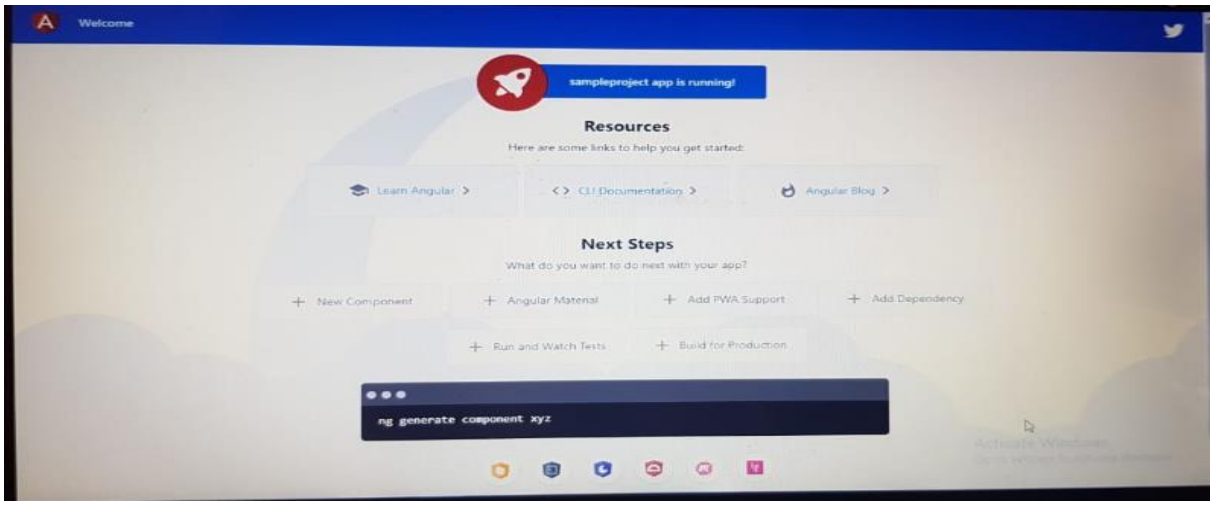
7. goto--->terminal--->newterminal F:\Ag7am\sampleproject> **ng serve –o**



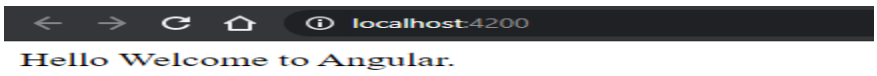
Angular is a Single page application. i.e Whenever we click on link desired page will come without refreshing page.



Output on running the basic Project with out adding any code to project created.



By replacing All code present in **app.component.html** with below code to print ”Hello Welcome to Angular”



Create a project which take fname, lname from user and print the full name.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent,  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
     FormsModule ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.html

ngModel: it is used to catch the value entered at runtime on the html page text box.

```
Enter FirstName<input type="text" [(ngModel)]="FirstName"><br>
Enter LastName<input type="text" [(ngModel)]="LastName"><br>
<input type="button" (click)="GetFullName()" value="Display"><br>
<span>{{FullName}}</span>
<router-outlet></router-outlet>
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
   FirstName:string;
```

LastName:string;

FullName:string;

public GetFullName():void

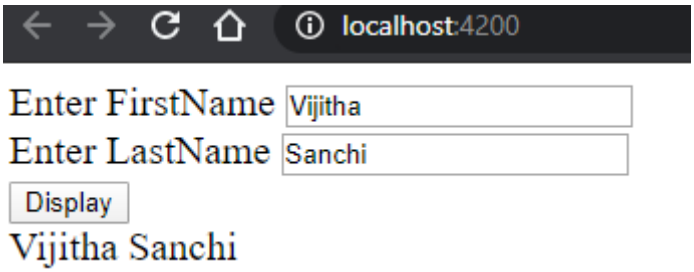
{

this.FullName=this.FirstName+this.LastName; }

title = 'cp';

}

Output:



2.**Write a project to display the sum or diff based on button clicked?**

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent,  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
     FormsModule ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.html

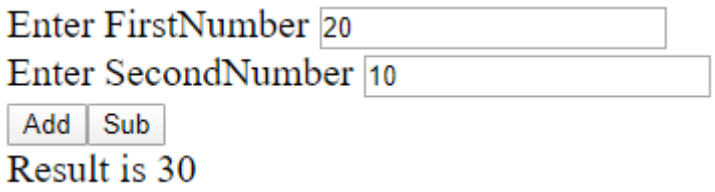
```
Enter FirstNumber
<input type="number" [(ngModel)]="FirstNumber"><br>
Enter SecondNumber
```

```
<input type="number" [(ngModel)]="SecondNumber"><br>
<input type="button"(click)="Add()" value="Add">
<input type="button" (click)=Sub() value="Sub">
<br>
<span>Result is {{Result}}</span>
<router-outlet></router-outlet>
```

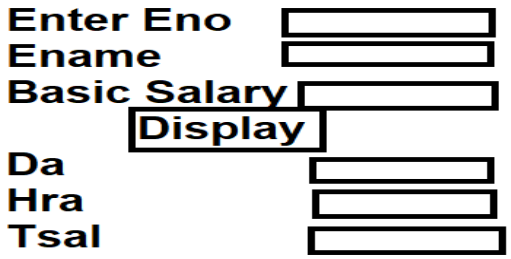
app.component.ts

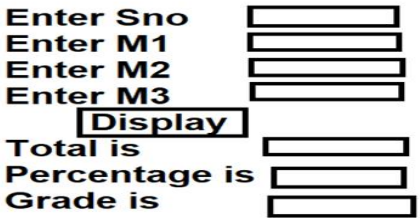
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  FirstNumber:number;
  SecondNumber:number;
  Result:number;
  public Add(){
    this.Result=this.FirstNumber+this.SecondNumber;
  }
  public Sub(){
    this.Result=this.FirstNumber-this.SecondNumber;
  }
  title = 'addsub2no';
}
```

Output:



Tasks:





Task1: Create a project to display Da,Hra,Tsal based on the Basic Salary entered by the user.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { AppRoutingModule } from './app-routing.module';
```

```
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent,  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
     FormsModule ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

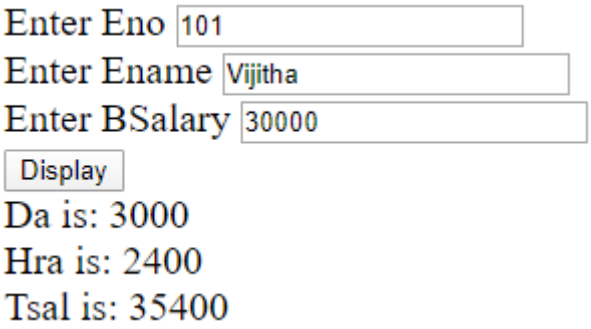
app.component.html

```
Enter Eno <input type="number" [(ngModel)]="Eno"><br>
Enter Ename <input type="string" [(ngModel)]="Ename"><br>
Enter BSalary <input type="number" [(ngModel)]="BSalary"><br>
<input type="button" (click)="Display()" value="Display"><br>
<span>Da is: {{Da}} </span><br>
<span>Hra is: {{Hra}} </span><br>
<span>Tsal is: {{Tsal}} </span><br>
<router-outlet></router-outlet>
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  Eno:number;
  Ename:string;
  BSalary:number;
  Da:number;
  Hra:number;
  Tsal:number;
  public Display(){
    this.Da=(10/100)*this.BSalary;
    this.Hra=(8/100)*this.BSalary;
    this.Tsal=this.BSalary+this.Da+this.Hra;
  }
  title = 'employeesalary';
}
```

Output:



Task2: Create a project to display Total, Percentage,Grade based on the 3 different marks entered by the user.

app.module.ts

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

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

import {FormsModule} from '@angular/forms'

import { AppRoutingModule } from './app-routing.module';

import { AppComponent } from './app.component';
@NgModule({

declarations: [

AppComponent,],

imports: [

BrowserModule,

AppRoutingModule,

FormsModule],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  Sno:number;
  M1:number;
  M 2:number;
  M3:number;
  Total:number;
```

```
  Percentage:number;
  public Display(){
    this.Total=this.M1+this.M2+this.M3;
    this.Percentage=this.Total/3;
  }
  title = 'Studentdetails';
}
```

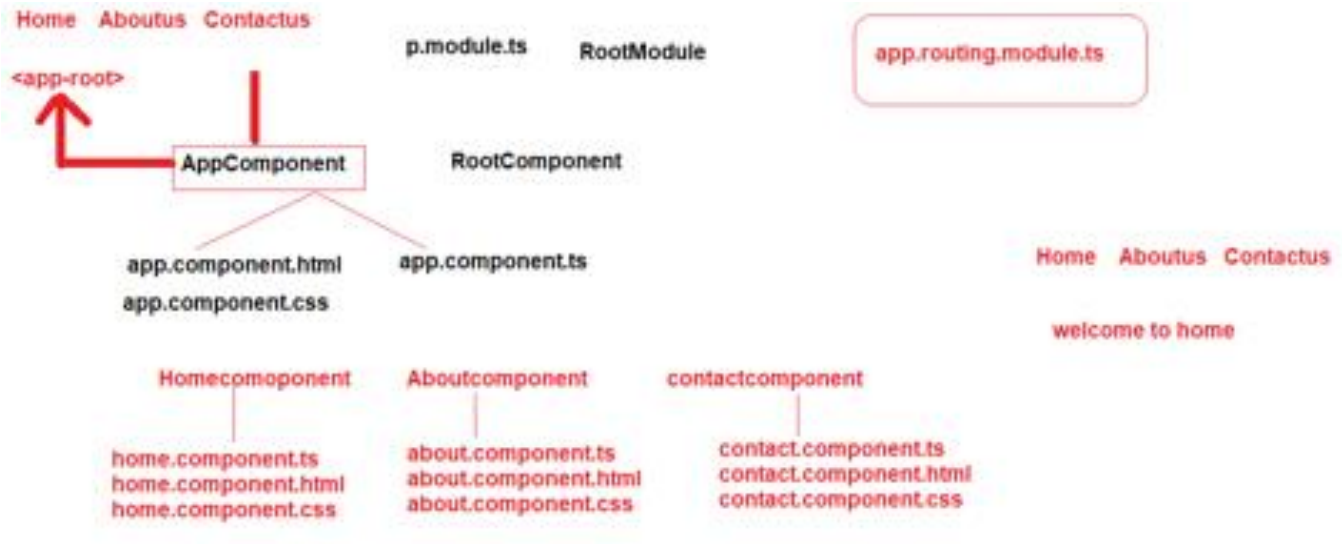
app.component.html

```
Enter Sno <input type="number" [(ngModel)]="Sno"><br>
Enter M1 <input type="number" [(ngModel)]="M1"><br>
Enter M2 <input type="number" [(ngModel)]="M2"><br>
Enter M3 <input type="number" [(ngModel)]="M3"><br>
<input type="button" (click)="Display()" value="Display"><br>
<span>Total is: {{Total}} </span><br>
<span>Percentage is: {{Percentage}} </span><br>
<router-outlet></router-outlet>
```

Output:

Enter Sno 598
Enter M1 89
Enter M2 76
Enter M3 99
Display
Total is: 264
Percentage is: 88

Example: Routing Module (hyper links)



Step 1: Install the Angular project.

Before Routing first, you need to install the Angular project for that use the below command to create your new Angular project.

ng new router (project name)

Step 2: Create new three components in your Angular app.

Now create a new three components as home, about, contact in your Angular app. Follow the below commands which help you to create components.

ng g c home

ng g c about

ng g c contact

Step 3: Import created three components.

Import created three components inside the **src >> app >> app.module.ts** Below you have the code use like as same. Also the router needs a <base href="/"> to be set in the index.html section.

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

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

import { AppRoutingModule } from './app-routing.module';

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

import { HomeComponent } from './home/home.component'; //Home Component (comes by default)

import { AboutComponent } from './about/about.component'; //About Component

import { ContactComponent } from './contact/contact.component'; //Contact Component

@NgModule({

declarations: [

AppComponent,

HomeComponent, //Home Component

AboutComponent, //About Component

ContactComponent //Contact Component (comes by default)

],

imports: [

BrowserModule,

AppRoutingModule

],

providers: [],

bootstrap: [AppComponent]

})

export class AppModule { }

Note: If you are using the Angular latest version don't worry, the components automatically import the components in your app.module.ts

Step 4: Configure routing in your app-routing.module.ts

Now setup routes path in your src >> app >> app-routing.module.ts as like the below code.

app-routing.module.ts

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';
import { HomeComponent } from './home/home.component';
// Define Routing Path
const routes: Routes = [
 { path:"",component: HomeComponent },
{ path: 'home', component: HomeComponent },
{ path: 'about', component: AboutComponent },
{ path: 'contact', component: ContactComponent}
];
@NgModule({
imports: [RouterModule.forRoot(routes)],
exports: [RouterModule]
})
export class AppRoutingModule { }
```

Above you can see, we define the created component routes path and path name with the component and component name.

Step 5: Define the Router with active links

Now open and add the routerLink in your app.component.html as like the below.

```
<div>
<h1>
Welcome to {{ title }}!
</h1>
</div>
<ul>
<li>
<a routerLink="home" routerLinkActive="active">Home</a>
</li>
<li>
```

```
<a  routerLink="about" routerLinkActive="active">About</a>
</li>
<li>
<a routerLink="contact" routerLinkActive="active">Contact</a>
</li>
</ul>
<router-outlet></router-outlet>
```

about.component.html

```
<p>Welcome to About Page! </p>
```

contact.component.html

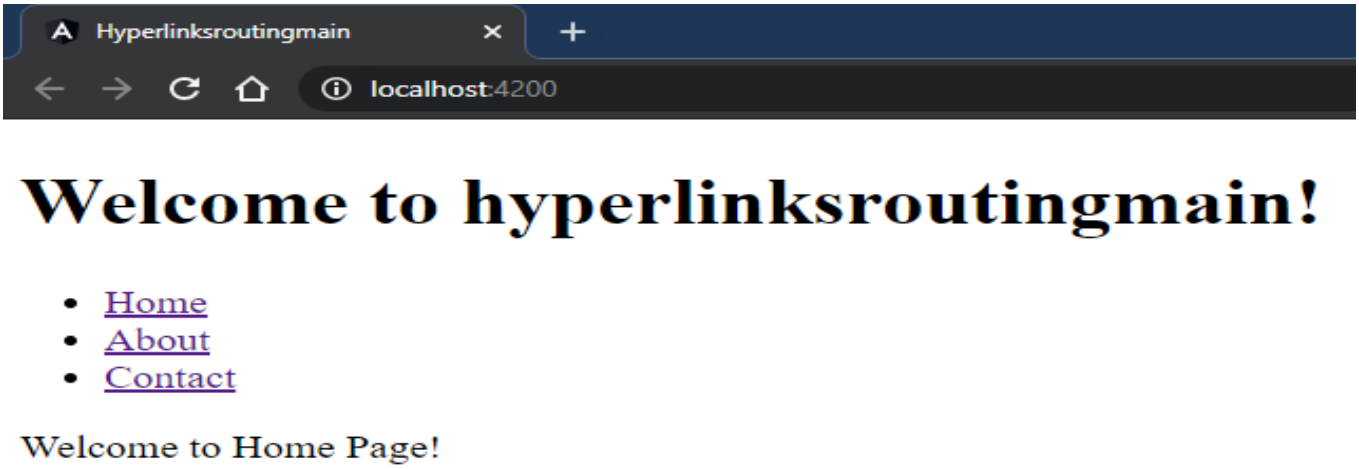
```
<p>Welcome to Contact Page! </p>
```

home.component.html

```
<p>Welcome to Home Page! </p>
```

- In the normal HTML site, we use like About but here instered of that, we going to use Angular route concept with routerLinkActive, like this <a routerLink="about" routerLinkActive="active">About
- Once you completed all the steps now run the command ng serve --open and see the output with an active link. For your reference below you have the output how it looks like.

  OUTPUT:



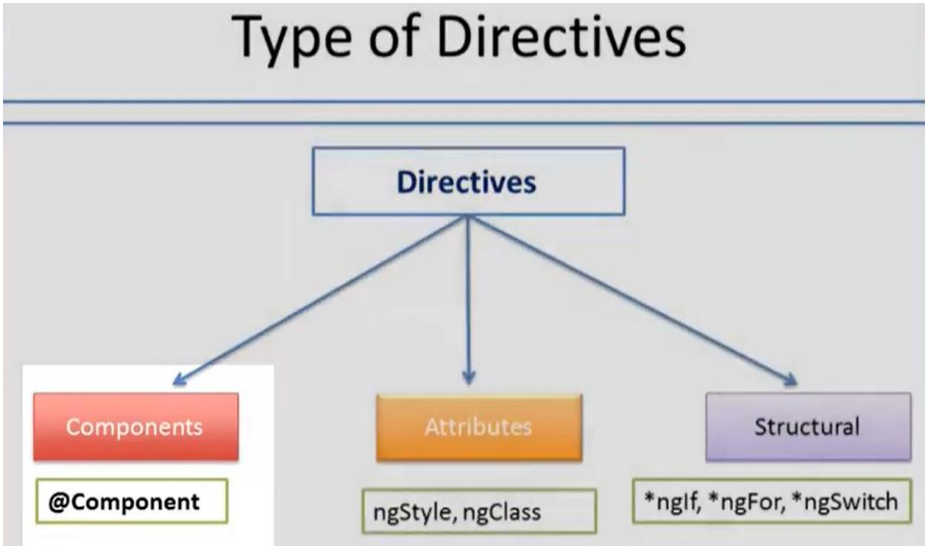
Directives

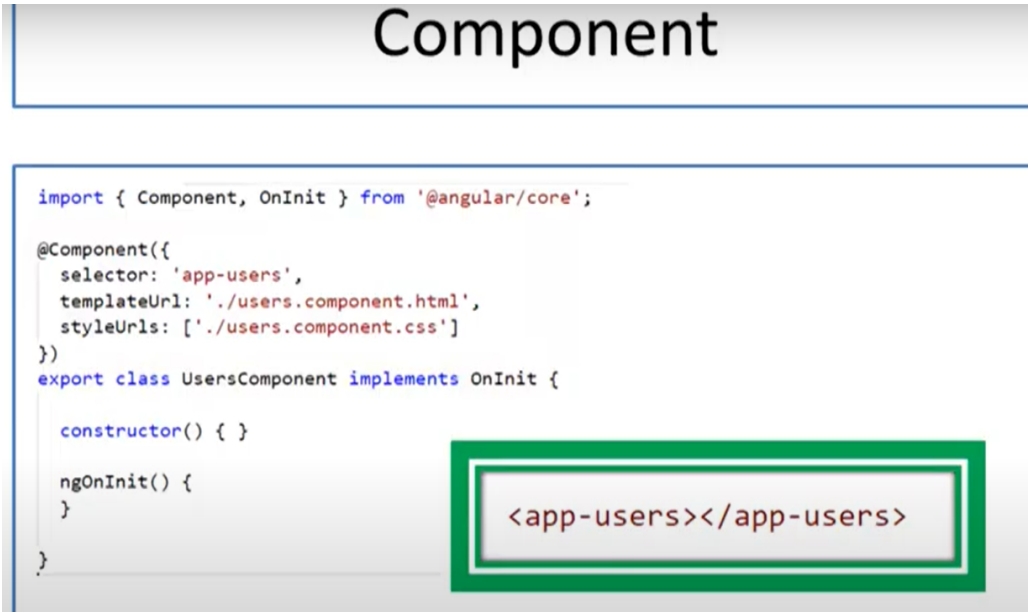
Q) What is Directives?

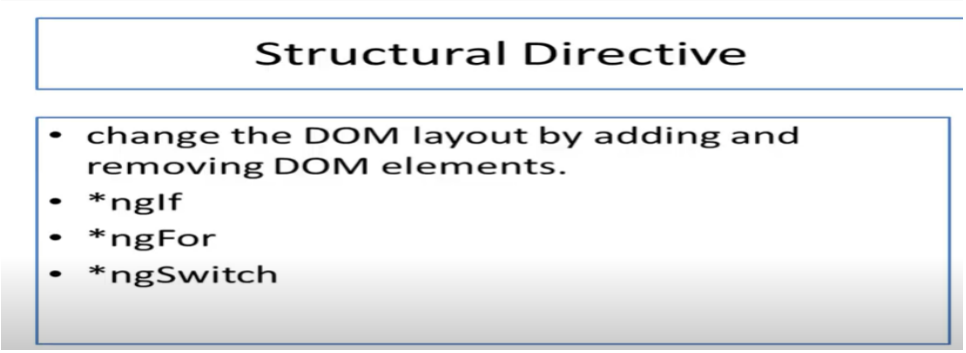
Directives are one of the important and cool feature of angular application and its play a important role in angular projects. With the help of directive we can easily manipulate our DOM layout.

<div **hidden**>

<p **hidden** ></p>







ngIf else

To begin with, **Angular 9/8/7 Ngif** works like regular if else only. It will evaluate the expression and it will show or hide the element based on the result of that evaluation.

You need to know that **Ngif** basic syntax is effective and simple. Here you need to prefix an asterisk (*) with the directive name. Then you can include it inside the template, wherever you wish to have it. Well, we are definitely going to address why use an asterisk as we explore ngIf in Angular.

// * + ngIf = *ngIf Directive formula

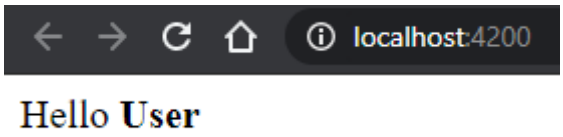
As far as using *ngIf is concerned, there are 4 main ways. We will start with the basic form first below:

1st way:

app.

```
@Component({
  selector: 'my-app',
  template: `    <div *ngIf="userLoggedIn">
     Hello <strong>User</strong>    </div> `,
})
export class AppComponent  {
  userLoggedIn = true;
}
```

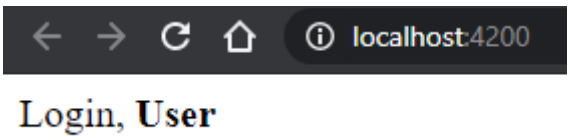
Output:



2nd way: Using Logical NOT (!) Operator with NgIf in Angular

```
@Component({
  selector: 'my-app',
  template: `    <div *ngIf="!userLoggedIn">
     Login, <strong>user</strong>    </div>  `,
})
export class AppComponent  {
  userLoggedIn = false;
}
```

Output:



3rd way:Using Logical AND (&&) Operator with Angular NgIf. Check out the && operator example in Angular with ***ngIf**.

```
@Component({
  selector: 'my-app',
  template: `    <input type="checkbox" [(ngModel)]="checked"/>
    <input type="checkbox" [(ngModel)]="checked2" />
    <span *ngIf="checked && checked2">
      Show this text when both checkboxes are checked!    </span>`,
})
export class AppComponent  {
  alert('Check both the checkboxes to show message')
}
```

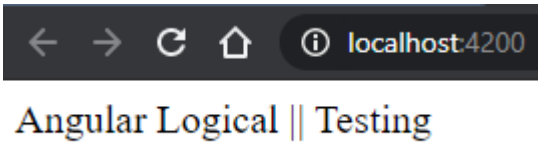
4th Way:

Using Logical OR (||) Operator with Angular NgIf

Check out the || operator example in Angular with ***ngIf**.

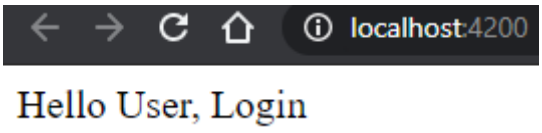
```
@Component({
  selector: 'my-app',
  template: `  <div *ngIf="x == 5 || y == 5">
    Angular Logical || Testing  </div>  `,
})
export class AppComponent {
  x = 5;
  y = 9;
}
```

Output;



Else statement is an excellent addition to **Angular 9/8/7**. It has its roots in Angular JS. There is nothing complicated about this though. If this, execute this **else** execute something else.

```
@Component({
  selector: 'my-app',
  template: `    <div *ngIf="userLoggedIn; else userloggedOut">
      Hello User
    </div>
    <ng-template #userloggedOut>
      Hello User, Login
    </ng-template>  `,
})

export class AppComponent {
  userLoggedIn = false;
  userloggedOut = true;
}
```



Here the thinking in the flow of statements has been accustomed to the syntax. It aligns well with our thoughts, like this:

```
ngIf = condition ? then : else;
```

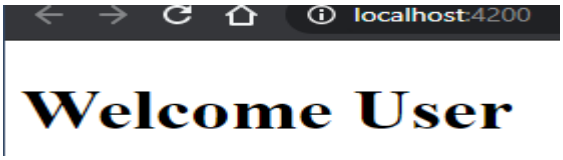
ngIf(Appcomponent.ts)(add forms module in app.module.ts)

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

  selector: 'app-root',

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

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

})

export class AppComponent {    isUserLoggedIn:boolean=true;}
```

app.component.html

```
<div *ngIf="isUserLoggedIn">

  <h1>Welcome  User</h1></div>

<router-outlet></router-outlet>
```

Output:



app.component.ts:-

```
export class AppComponent {
   isUserLoggedIn:boolean=false;
}
```

app.component.html

```
<div *ngIf="isUserLoggedIn;else logout">
  <h1>Welcome  User</h1>
</div>
<ng-template #logout>
  <h1>Please Login</h1>
</ng-template>
<router-outlet></router-outlet>
```

OutPut:

Please Login

---

app.component.ts:-

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'myd';
  X:number;
  visible:boolean=false;
  CheckEven()
  {
    this.visible=this.X%2==0;
  }
  }
```

app.component.html

```
<!--The content below is only a placeholder and can be replaced.-->
Enter a no
<input type="number" [(ngModel)]="X"><br>
<input type="button" (click)="CheckEven()" value="Check"/>
<div *ngIf="visible;then Even else Odd"></div>
<ng-template #Even>Even no</ng-template>
<ng-template #Odd>Odd no</ng-template>
<router-outlet></router-outlet>
```

Output:

Without displayed Default Text

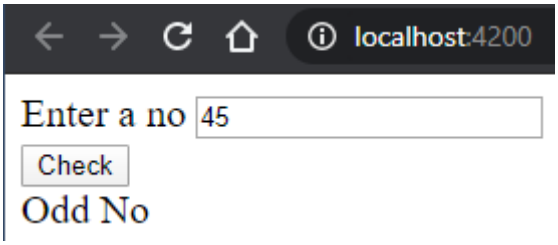
```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'myd';
  X:number;
  IsEven:boolean=false;
  IsOdd:boolean=false;
  CheckEven()
  {
    if(this.X%2==0)
    {
      this.IsEven=true;
      this.IsOdd=false;
    }
```

```
  if(this.X%2!=0)
  {
    this.IsEven=false;
    this.IsOdd=true;
  }
 }
 }
```

app.component.html

```
<!--The content below is only a placeholder and can be replaced.-->
Enter a no
<input type="number" [(ngModel)]="X">
<br>
<input type="button" (click)="CheckEven()" value="Check"/>
<div *ngIf="IsEven">
  <span>Even no</span>
</div>
<div *ngIf="IsOdd">
<span>Odd No</span>
</div>
<router-outlet></router-outlet>
```

Output:



Example 1: Write a project to validate the username and password of login page?

Step 1: Install the Angular project.
Before Routing first, you need to install the Angular project for that use the below command to create your **new Angular project.**
ng new router (project name)

Step 2: Create new three components in your Angular app.

Now create a new three components as home, about, contact in your Angular app. Follow the below commands which help you to create components.

ng g c home

ng g c about

ng g c contact

ng g c login

Step 3: Import created three components.

Import created three components inside the **src >> app >> app.module.ts** Below you have the code use like as same. Also the router needs a <base href="/"> to be set in the index.html section.

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutComponent } from './about/about.component';
import { ContactComponent } from './contact/contact.component';
import { LoginComponent } from './login/login.component';
import {from} from 'rxjs';
@NgModule({
  declarations: [
    AppComponent,
    HomeComponent,
    AboutComponent,
    ContactComponent,
    LoginComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Step 4: Configure routing in your app-routing.module.ts

Now setup routes path in your src >> app >> app-routing.module.ts as like the below code.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import {HomeComponent} from './home/home.component';
import{AboutComponent} from './about/about.component';
import{ContactComponent} from './contact/contact.component';
import {LoginComponent} from './login/login.component';
const routes: Routes = [
  {path:"",component:HomeComponent},
  {path:'home',component:HomeComponent},
  {path:'about',component:AboutComponent},
  {path:'contact',component:ContactComponent},
  {path:'login',component:LoginComponent}
];
@NgModule({
```

```
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

Above you can see, we define the created component routes path and path name with the component and component name.

Step 5: Define the Router with active links

Now open and add the routerLink in your **app.component.html** as like the below.

```
<div>
  <h1>
    Welcome to {{title}}!
  </h1>
</div>
<ul>
  <li>
    <a routerLink="home" routerLinkActive="active">Home</a>
  </li>
  <li>
    <a routerLink="login" routerLinkActive="active">Login</a>
  </li>
  <li>
    <a routerLink="contact" routerLinkActive="active">Contact</a>
  </li>
  <li>
    <a routerLink="about" routerLinkActive="active">About</a>
  </li>
</ul>
<router-outlet></router-outlet>
```

home.component.html

```
<p>Welcome to Home Page!</p>
```

about.component.html

```
<p>Welcome to About Page!</p>
```

contact.component.html

```
<p>Welcome to Contact Page!</p>
```

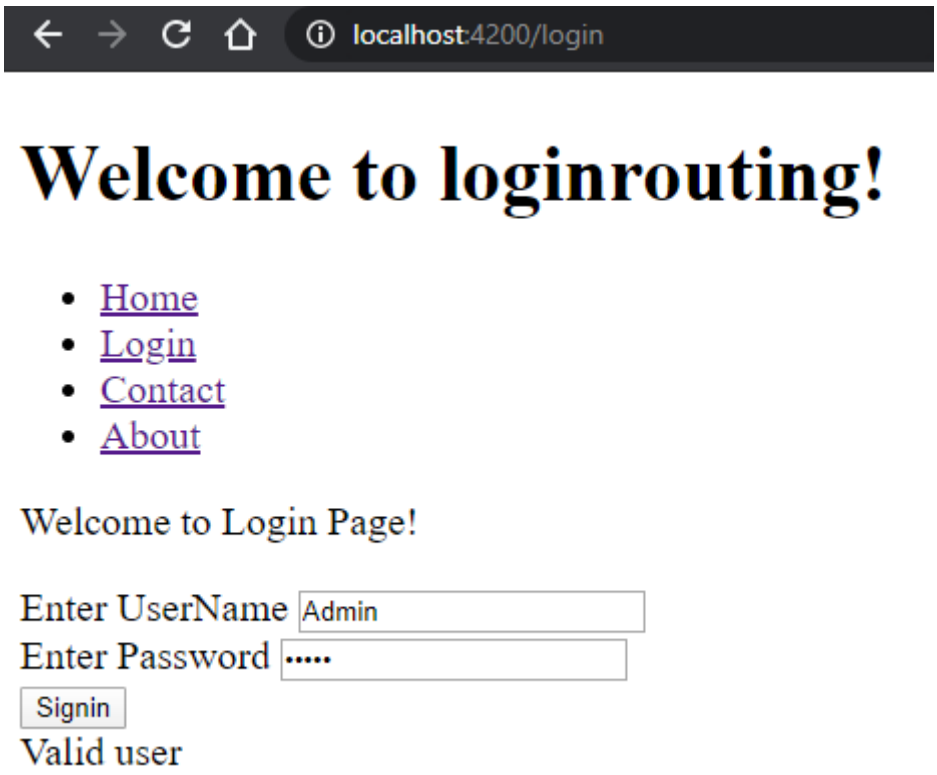
login.component.html

```
<p>Welcome to Login Page!</p>
<div>
    Enter UserName <input type="text" [(ngModel)]="UserName"><br>
    Enter Password <input type="password" [(ngModel)]="Password"><br>
    <input type="button" value="Signin" (click)="CheckUser()"><br>
    <div *ngIf="isvalid">Valid user</div>
    <div *ngIf="isinvalid">inValid user</div>
</div>
```

Login.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit {
  UserName:string;
```

```
  Password:string;
  isvalid:boolean;
  isinvalid:boolean;
  constructor() { }
  ngOnInit(): void {  }
  CheckUser(){
    if(this.UserName=="Admin" && this.Password=="Admin"){
      this.isvalid=true;
      this.isinvalid=false;
    }
    else{
      this.isvalid=false;
      this.isinvalid=true;
    }
  }
}
```



Displaying Data with Angular 9/8 ngFor Directive

What is the role of ngFor and how it is used in Angular?

We are going to learn about one such a directive, which is known as **ngFor**. The ***ngFor="…"** syntax is used to iterate over an array or even an object to produce a template of list items.

How to display data using Angular ngFor syntax?

Now, we are going to display the employee data using the given below JavaScript array of objects.

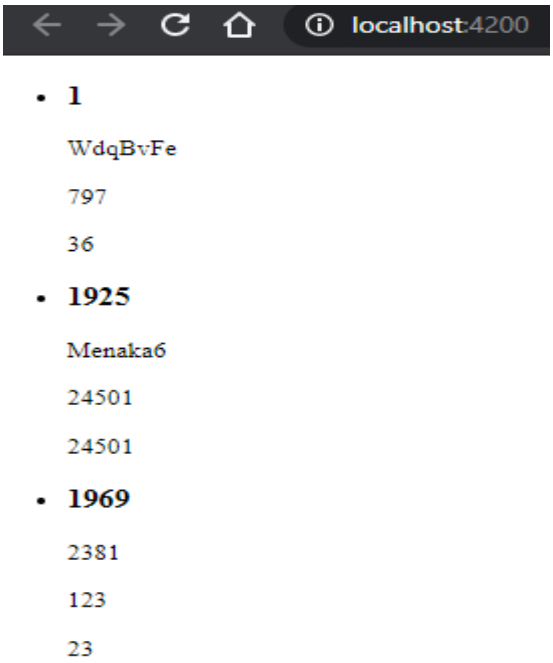
```
// Employee Data in app.component.ts
const Employee = [
   {"id":"1","employee_name":"WdqBvFe","employee_salary":"797","employee_age":"36","profile_image":""},
 {"id":"1925","employee_name":"Menaka6","employee_salary":"24501","employee_age":"24501","profile_image":""},
   {"id":"1969","employee_name":"2381","employee_salary":"123","employee_age":"23","profile_image":""},
   {"id":"1970","employee_name":"6132","employee_salary":"123","employee_age":"23","profile_image":""},
   {"id":"1972","employee_name":"2022","employee_salary":"123","employee_age":"23","profile_image":""},
```

```
  {"id":"1973","employee_name":"4604","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"1976","employee_name":"Shylu","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"1977","employee_name":"8221","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"1981","employee_name":"111test","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"1996","employee_name":"test709","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"1997","employee_name":"test-654","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"1999","employee_name":"test-127","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"2001","employee_name":"test-301","employee_salary":"123","employee_age":"23","profile_image":""},
  {"id":"2003","employee_name":"1769","employee_salary":"123","employee_age":"23","profile_image":""}
]
```


With the help of the ngFor directive, we are going to iterate the Employees array and show the data in the Angular template using the HTML table or unordered or ordered list (ul li or ol li).

```
<ul>
  <li *ngFor="let employee of Employee ">
    <h3>{{employee.id}}</h3>
    <p> {{employee.employee_name}} </p>
    <p> {{employee.employee_salary}} </p>
    <p> {{employee.employee_age}} </p>
    <p> {{employee.profile_image}} </p>
  </li>
</ul>
```

Output: Same as below output one below thee other.




```
<table>
  <tr>
    <th>Eid</th>
    <th>Ename</th>
    <th>ESalary</th>
    <th>eage</th>
  </tr>
  <tr *ngFor="let employee of Employee">
    <td>{{employee.id}}</td>
    <td> {{employee.employee_name}} </td>
    <td> {{employee.employee_salary}} </td>
    <td> {{employee.employee_age}} </td>
    <td> {{employee.profile_image}} </td>
  </tr>
</table>
```

Output:

Eid	Ename	ESalary	eage
1	WdqBvFe	797	36
1925	Menaka6	24501	24501
1969	2381	123	23
1970	6132	123	23
1972	2022	123	23
1973	4604	123	23
1976	Shylu	123	23
1977	8221	123	23
1981	111test	123	23
1996	test709	123	23
1997	test-654	123	23
1999	test-127	123	23
2001	test-301	123	23
2003	1769	123	23

What is the ngFor directive scope?

When we are working with the ngFor directive, we should also be aware of its scope.

The variable we use to display the data with ngFor is only available to its scope. It similarly works the way other method works to iterate over a data collection in different programming languages.

For example, this will be a valid syntax:

```
<ul>
   <li *ngFor="let employee of Employee ">
      <h3>{{employee.id}}</h3>
      <p> {{employee.employee_name}} </p>
   </li>
</ul>
```

Output : like this all will be printed.

- **1**

  WdqBvFe

- **1925**

  Menaka6

- **1969**

  2381

- **1970**

  6132

- **1972**

  2022

- **1973**

  4604

1.Write a project to display data in table format with out using SERVICES??

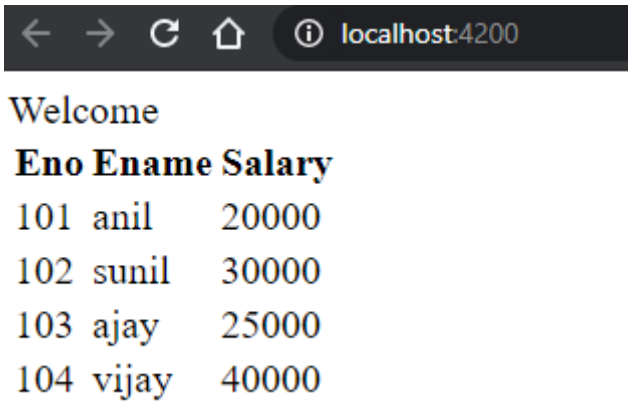
app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'myd';
  users:any[]=[
        {eno:101,ename:"anil",salary:20000},
        {eno:102,ename:"sunil",salary:30000},
        {eno:103,ename:"ajay",salary:25000},
        {eno:104,ename:"vijay",salary:40000}
      ];
}
```

app.component.html

```
Welcome
<div>
  <table>
      <tr>
        <th>Eno</th>
        <th>Ename</th>
        <th>Salary</th>
      </tr>
      <tr *ngFor="let user of users">
        <td>{{user.eno}}</td>
        <td>{{user.ename}}</td>
        <td>{{user.salary}}</td>
      </tr>
    </table>
</div>
<router-outlet></router-outlet>
```

Output:



Angular 8|9 NgSwitch Directive Tutorial with Examples

Directives	Description
ngSwitch	It's a structural directive, all the possible values defined inside of it. It gets the switch value based on the matched statement among multiple possibilities.
ngSwitchCase	The ngSwitchCase statement is declared inside the ngSwitch directive with the specific value.
ngSwitchDefault	This statement executes when the expression doesn't match with any of the defined value.

Using NgSwitch Directive in Angular 8|9

In the below example, we will try to understand the NgSwitch expression in a simple manner. In order to set the ngSwitch directive in Angular HTML template. Add the [ngSwitch] directive along with the desired value on the main container:

```
<ul [ngSwitch]="superhero"></ul>
```

Now, ngSwitch has been added to the main container along with `superhero` value. Then, add the *ngSwitchCase directive inside the NgSwitch statement and the same way you can place as many as **ngSwitchCase** inside the NgSwitch statement.

```
<li *ngSwitchCase="'Groot'">Groot</li>
```

Lastly, we need to declare the *ngSwitchDefault directive. The ngSwitchDefault directive will show the default result if the **ngSwitchCase** statement unable to fetch the result.

```
<li *ngSwitchDefault>Batman</li>
```

Angular 8|9 NgSwitch Directive Example

You can see below how we used NgSwitch directive with HTML elements to show the matching result from the multiple choices:

```
<ul [ngSwitch]="superhero">
  <li *ngSwitchCase="'Groot'">Groot</li>
  <li *ngSwitchCase="'Ironman'">Ironman</li>
  <li *ngSwitchCase="'Hulk'">Hulk</li>
```

```
  <li *ngSwitchCase="'Thor'">Thor</li>
  <li *ngSwitchCase="'Spiderman'">Spiderman</li>
  <li *ngSwitchDefault>Batman</li>
</ul>
```

Define ngSwitch expression value in app.component.ts:

```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  templateUrl: './app.component.html',
  styleUrls: [ './app.component.css' ]
})
export class AppComponent  {
  superhero = 'Spiderman';
}
```

The [ngSwitch] directive will return `Spiderman` as a result from the multiple possibilities because we defined `Spiderman` as the superhero's value.

Using NgFor with NgSwitch in Angular 9

In this example, we will understand how to use ***ngFor directive** with ***ngSwitch** directive. We mainly use *ngFor directive to iterate over a data collection. In the below example we will create a Cars array and show the cars data on front-end using Angular 8|9 *ngFor and *ngSwitch directive. This way, we will assign dynamic color classes to the HTML elements.

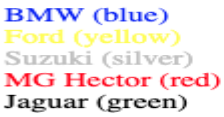
```
import { Component } from '@angular/core';
@Component({
  selector: 'my-app',
  template: `
    <div *ngFor="let car of Cars" [ngSwitch]="car.color">
      <div *ngSwitchCase="'blue'"          class="blue">
        {{ car.name }} ({{ car.color }})
      </div>
      <div *ngSwitchCase="'yellow'"          class="yellow">
        {{ car.name }} ({{ car.color }})
      </div>
      <div *ngSwitchCase="'silver'"          class="silver">
        {{ car.name }} ({{ car.color }})
      </div>
      <div *ngSwitchCase="'red'"          class="red">
        {{ car.name }} ({{ car.color }})
      </div>
      <div *ngSwitchDefault          class="text-warning">
        {{ car.name }} ({{ car.color }})
      </div>
    </div>`
})

export class AppComponent {
  Cars: any[] = [
    {      "name": "BMW",      "average": 12,      "color": 'blue'    },
    {      "name": "Ford",      "age": 15,      "color": 'yellow'    },
    {      "name": "Suzuki",      "age": 18,      "color": 'silver'    },
    {      "name": "MG Hector",      "age": 14,      "color": 'red'    },
    {      "name": "Jaguar",      "age": 8,      "color": 'green'    }
  ];
}
```

Add some styling in styles.css:

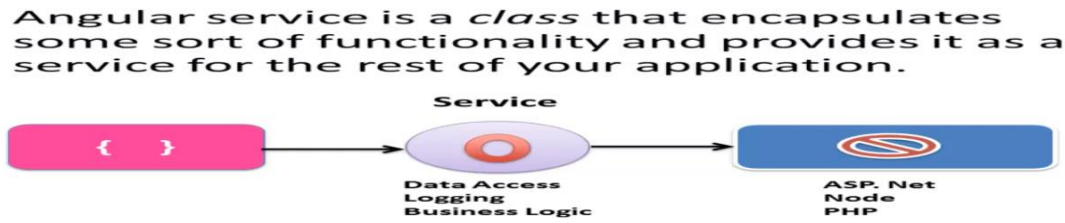
```
.blue {  color: blue}
```

```
.yellow {  color: yellow}
.silver {  color: silver}
.red {  color: red}
.blue {  color: blue}
```

Now, above Angular NgSwitch expression will apply various color classes to the HTML element using NgFor directive:

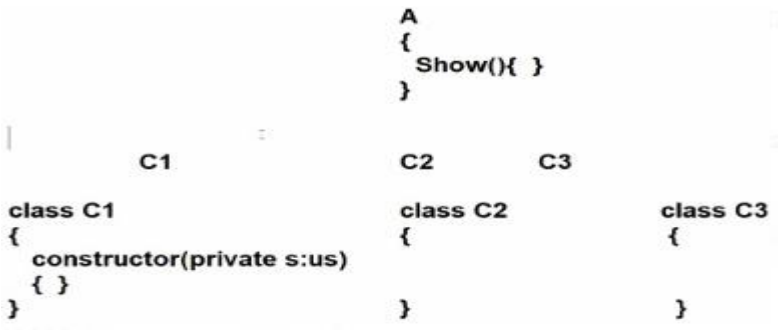


Services



Sharing of data is one of the requirement in angular application. Sometimes we want to use same data across components. So to do so you have 2 approaches

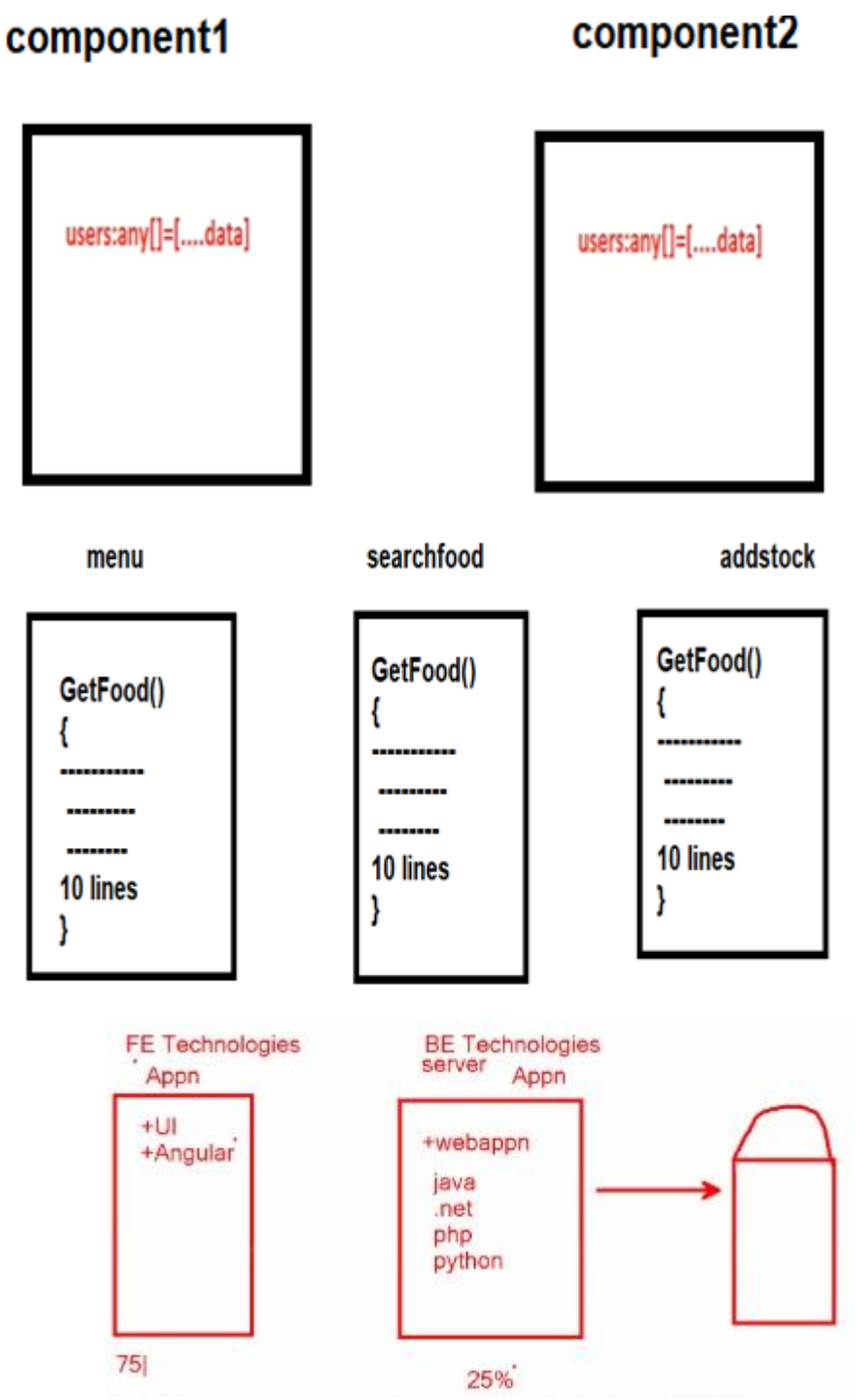
1. create that data in each component individually - but there is drawback with that approach we are re-writing the code which is against the approach of DRY(DONT REPEAT YOURSELF) and code also become lengthy , complex and less testable. So next approach is Services. Services follows singleton object.



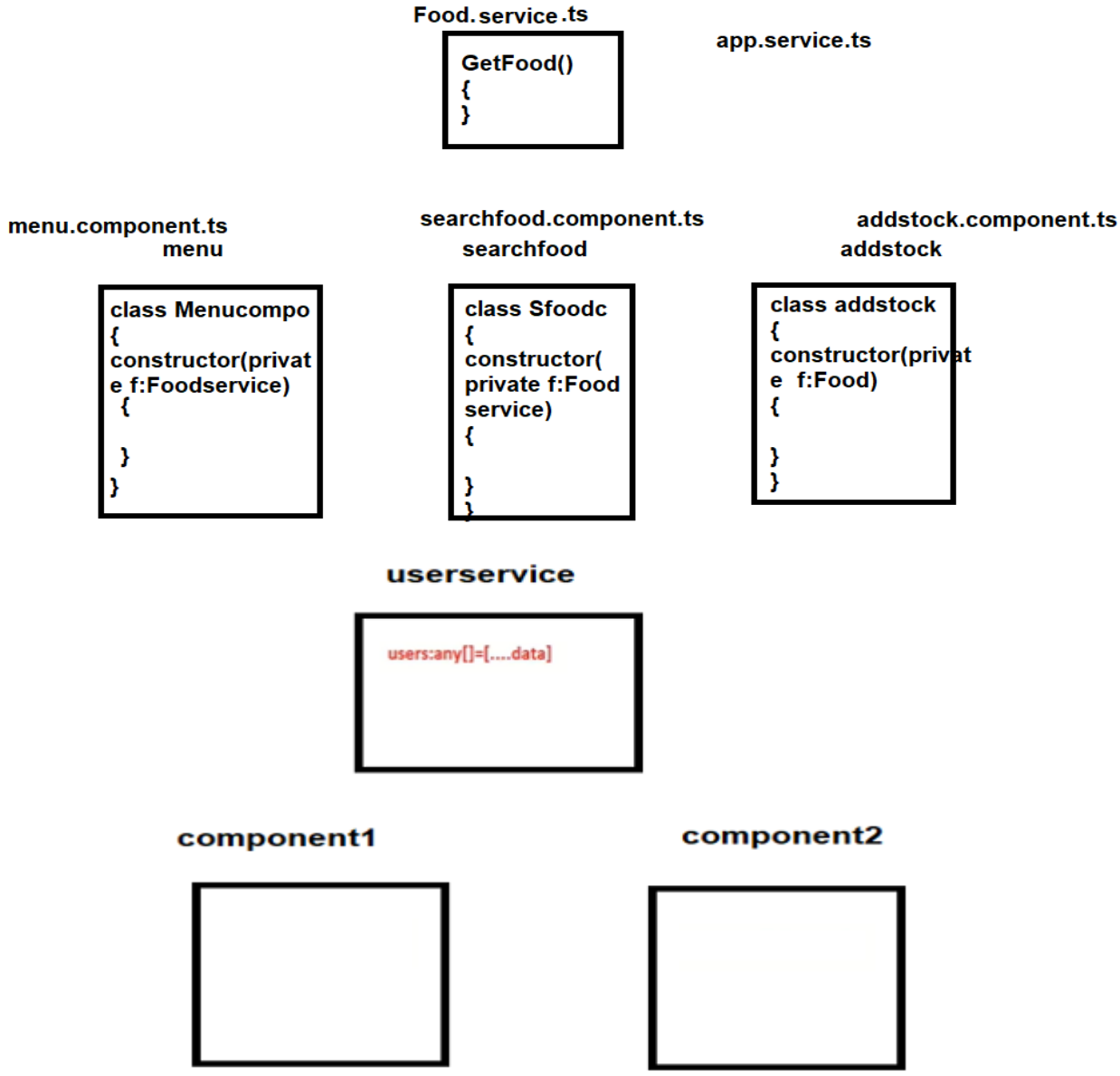
Classes before services implemented. multiple objects were to be created.

Problems of writing repeated code in multiple components

- Code Redundency
- Code Modifications is Difficult
- Code Testing
- Code Maintainance is difficult

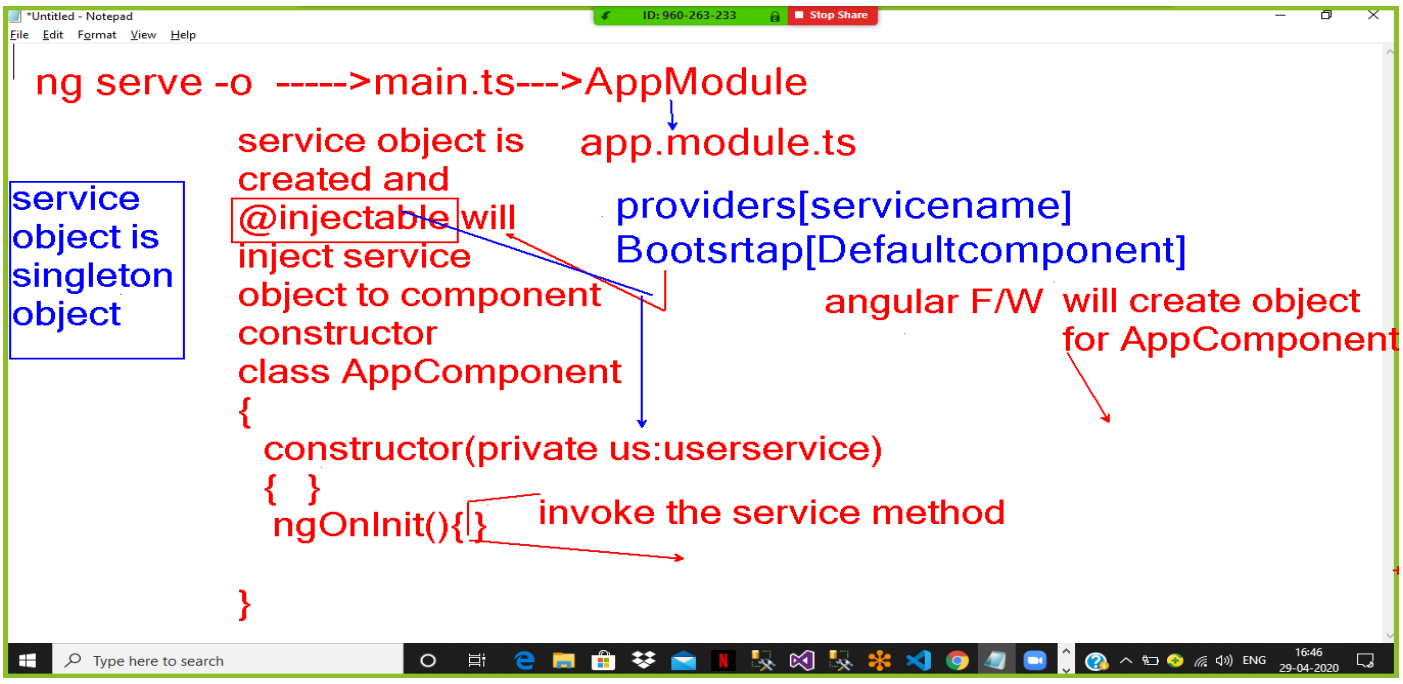


2) Service is a centralized location in angular application where you can keep your data which you want to share across components. Any modifications done in service will automatically reflected back to all the components who is injecting this service because services are singleton (Single instance of that service will get created). To create service use ng generate command: **ng g s services/userService**



All services class will use Injectable() decorator because we can inject service in any component

After that we need to register our service the way we have registered our component to app.module.ts in declarations. All service will registered in app.module.ts providers[] array.



After registration we need to use DI(Dependency Injection) to inject the reference of that service in any component .After DI we can call all methods define in that service class.Services are one of the part of angular which basically use for sharing of data across the angular application and we also use it to make HTTP Call. Command to create services: **ng g s services/users**

ng serve -o

main.ts class will be loaded ---->search for RootModule (AppModule)

app.module.ts:-

@NgModule({

declarations:[], register components

imports:[], register imported modules

providers:[UserService], register services

bootstrap:[AppComponent], default componentname

})

export class AppModule{ }

object is created for UserService by angular Framework and this object is singleton object. singleton object means single instance(object) was created for the class and this object is consumed in multiple components,object is created for AppComponent. whenever object is created then constructor() will gets invoked, here we need to pass the userservice as parameter to AppComponent constructor.

class AppComponent implements OnInit{

constructor(private us:userservice) { }

ngOnInit(){}

}

in Angular constructors are used to inject the service objects within the components

ngOnInit() :- it is the implemented method of OnInit interface

it is the Angular Lifecycle Hook method

This method will invoke automatically after executing the constructor

within ngOnInt() we are invoking the service method getAllUsers()

and we are assigning the result of the method to users variable

and finally the o/p will be rendered with Html Table of app.component.html

and finally <app-root> index.html

1.Write a project to display data in table format using SERVICES??

users.service.ts

```
import { Injectable } from '@angular/core';
@Injectable({
  providedIn: 'root'
})
```

```
export class UsersService {
  constructor() { }

  getAllUsers()

  {

     return [

          {eno:101,ename:"anil",salary:20000},

          {eno:102,ename:"sunil",salary:30000},

          {eno:103,ename:"ajay",salary:25000},

          {eno:104,ename:"vijay",salary:40000}

        ];

  }

}
```

Register service in **app.module.ts**

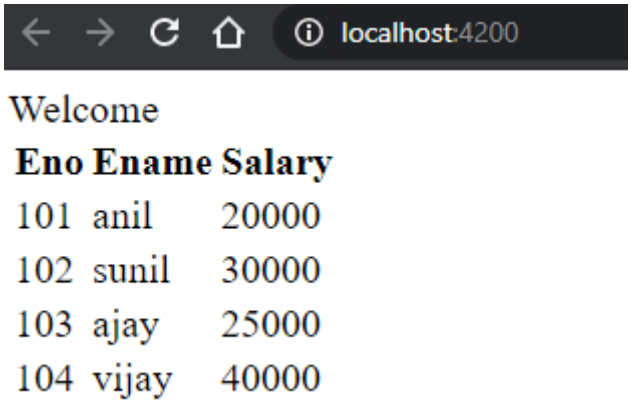
```
import { BrowserModule } from '@angular/platform-browser';

import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';

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

import {UsersService} from './services/users.service';

@NgModule({

  declarations: [

    AppComponent,

      ],

  imports: [

    BrowserModule,

    AppRoutingModule

  ],

  providers: [UsersService],

  bootstrap: [AppComponent]

})

export class AppModule { }
```

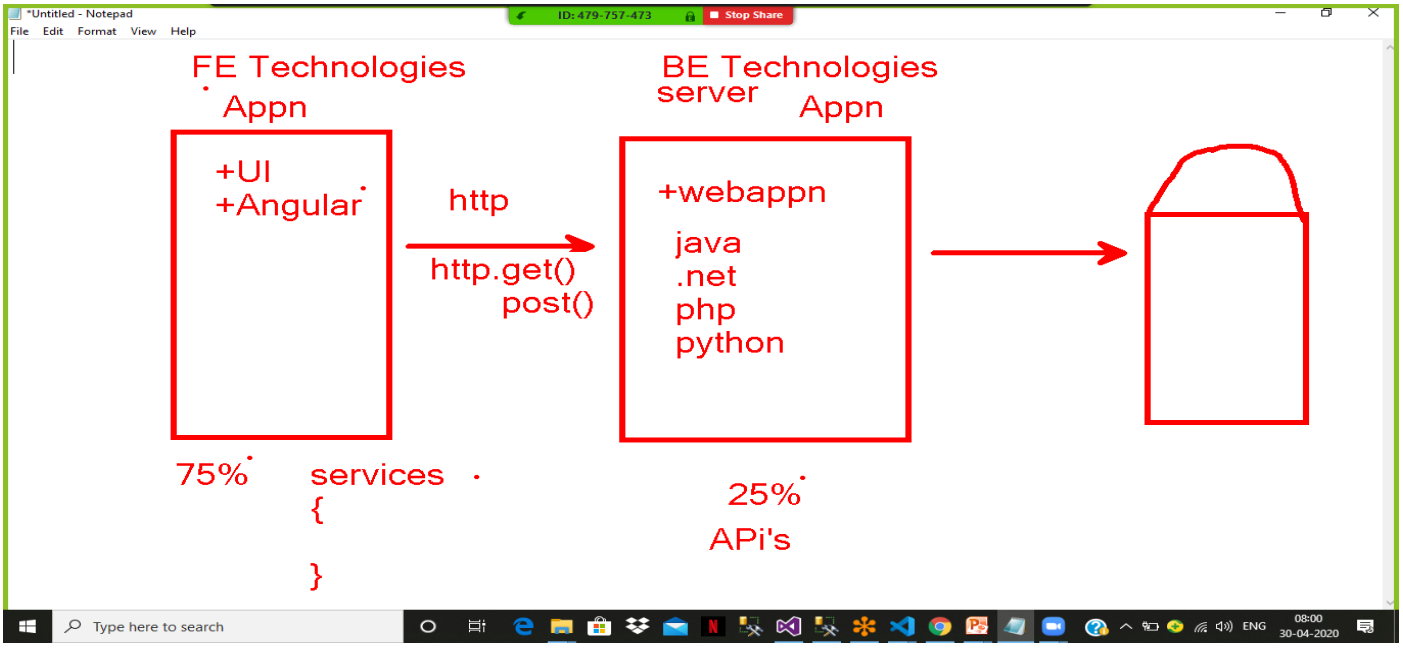
app.component.ts

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

import { UsersService } from './services/users.service';
@Component({

  selector: 'app-root',
```

```
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'myd'; //titleof project
  constructor(private userservice:UsersService)  {  }
  ngOnInit()
  {      this.users=this.userservice.getAllUsers();      }
  users:any[];
}
```

Output:

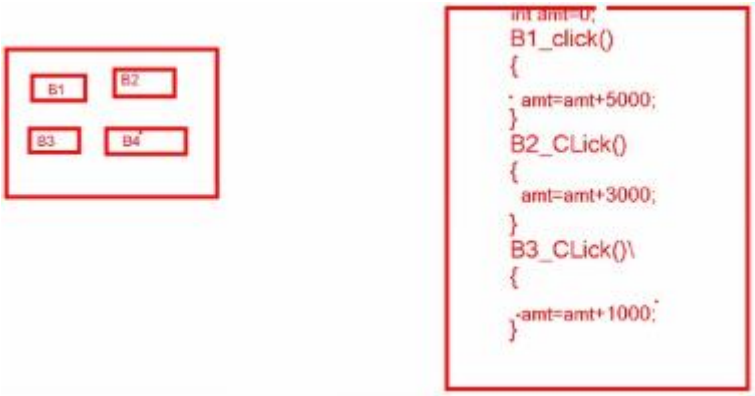




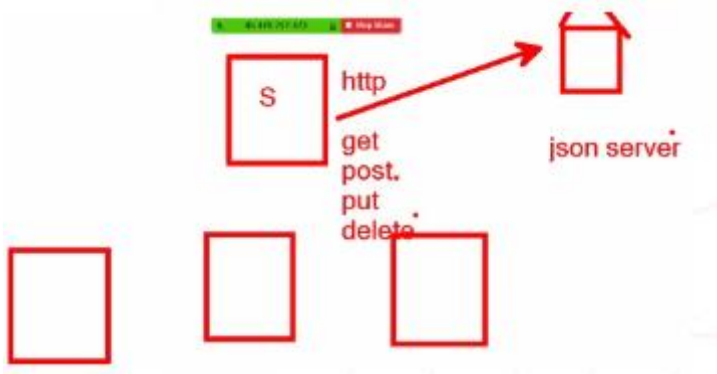
Most front end applications communicate with backend services over HTTP protocol. HTTP services is the important part of any application which communicates with server.

Till Now we are getting a dummy users data but now we want to get live data somewhere from server
Now we require remote data server which provides HTTP Web Service, which we can call using Angular Front End.
You can create Web Service in any language like Java, PHP, .Net or NodeJS.
We can consume this web service using HttpClient in an angular app.
To keep this demo easy to understand by everyone we will consume a FAKE online web service and you can also test at your end easily :
So use this FAKE online REST API :
http://jsonplaceholder.typicode.com/u...
To use HTTP service first step is to enable HTTP Service
.For this we need to include a HttpClientModule in imports array in app.module.ts and this module is defined in library angular/common/http

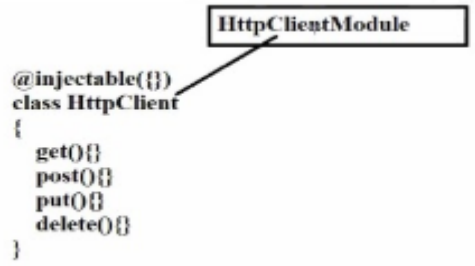
Stateless Management: every time when the method is called amount is reinitialized with 0.



Httpclient methods



post: insert data, put: update data, delete: delete data



Program Execution process:

Main.ts -> search for module name then control goes to module(app.module..ts)

In that we hav to register our service in providers:[] and its path also should be imported(HttpClientModule)

app.module.ts

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

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import {UsersService} from './services/users.service';
import { HttpClientModule } from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent,
      ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    HttpClientModule
  ],
  providers: [UsersService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Users.service.ts

```
import { Injectable } from '@angular/core';
import { HttpClientModule, HttpClient } from '@angular/common/http';

@Injectable({
  providedIn: 'root'
})
export class UsersService {
  constructor(private httpservice:HttpClient) { }
  getAllUsers()  {  return this.httpservice.get("http://jsonplaceholder.typicode.com/users");  }
}
```

app.component.ts

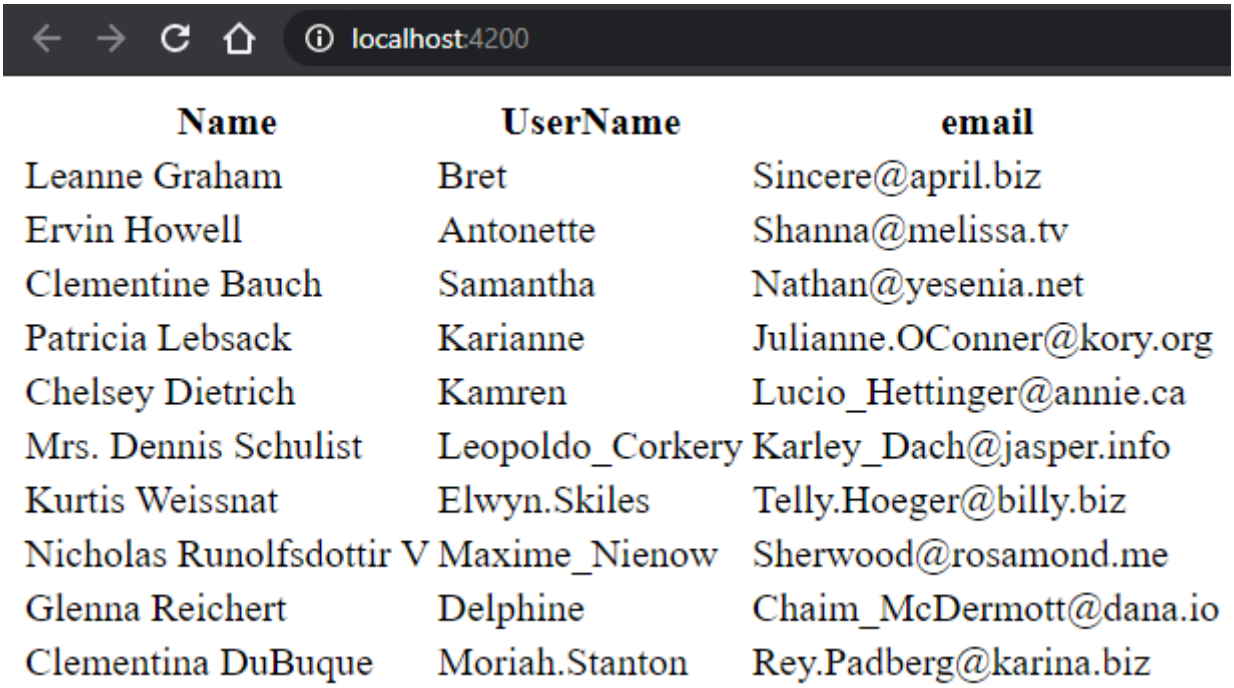
```
import { Component, OnInit } from '@angular/core';
import { UsersService } from './services/users.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'myd';
  users:any;
  constructor(private userservice:UsersService)
  {
  }
  ngOnInit()
  {
     this.userservice.getAllUsers().subscribe((data)=>
     {
       this.users=data;
     });
  }
}
```

app.component.html

```
<div>
  <table>
     <tr>
       <th>Name</th>
       <th>UserName</th>
       <th>email</th>
     </tr>
     <tr *ngFor="let user of users">
       <td>{{user.name}}</td>
       <td>{{user.username}}</td>
```

```
      <td>{{user.email}}</td>
    </tr>
  </table>
</div>
<router-outlet></router-outlet>
```

Output:



JSON: JavaScript Object Notation

JSON stands for JavaScript Object Notation. JSON is a syntax for storing and exchanging data. JSON is text, written with JavaScript object notation. JSON objects are used for transferring data between server and client, XML serves the same purpose.

JSON will maintain data in key-value pair format
var emp= { "firstName" : "Chaitanya", "lastName" : "Singh", "age" : "22" };

Features of JSON:

- It is light-weight
- It is language independent
- Easy to read and write
- Text based, human readable data exchange format
- <students> <student> <name>John</name> <age>23</age> <city>Agra</city> </student> <student> <name>Steve</name> <age>28</age> <city>Delhi</city> </student> <student> <name>Peter</name> <age>32</age> <city>Chennai</city> </student> <student> <name>Chaitanya</name> <age>28</age> <city>Bangalore</city> </student> </students>
- {"students":

  [

{"name":"John", "age":"23", "city":"Agra"}, {"name":"Steve", "age":"28", "city":"Delhi"},
{"name":"Peter", "age":"32", city":"Chennai"}, {"name":"Chaitanya", "age":"28",city":"Ban"}

]

}

JSON data structure types and how to read them:

- JSON objects
- JSON objects in array
- Nesting of JSON objects

JSON objects

var emp=

{

"name" : "Chaitanya Singh",

"age" : "28",

"website" : "beginnersbook“

};

JSON objects in array

- In the above example we have stored the information of one person in a JSON object suppose we want to store the information of more than one person; in that case we can have an array of objects.

var students =

[{ "name" : "Steve", "age" : "29", "gender" : "male" },

{ "name" : "Peter", "age" : "32", "gender" : "male" },

{ "name" : "Sophie", "age" : "27", "gender" : "female" }];

To access the information out of this array, we do write the code like this:
document.writeln(students[0].age); //output would be: 29 document.writeln(students[2].name); //output: Sophie

Nesting of JSON objects:

another way of doing the same thing that we have done above.

var students =

{ "steve" : { "name" : "Steve", "age" : "29", "gender" : "male" },

"pete" : { "name" : "Peter", "age" : "32", "gender" : "male" },

"sop" : { "name" : "Sophie", "age" : "27", "gender" : "female" } }

document.writln(students.steve.age);

//output: 29 document.writeln(students.sop.gender); //output: female

How to read data from json file and convert it into a JavaScript object?

We have two ways to do this.
1) Using eval function, but this is not suggested due to security reasons (malicious data can be sent from the server to the client and then eval in the client script with harmful effects).
2) Using JSON parser: No security issues plus it is faster than eval. Here is how to use it:

- var emp= { "name" : "Chaitanya Singh", "age" : "28", "website" : "beginnersbook" };
- **We are converting the above JSON object to javascript object using JSON parser**:
- var myJSObject = JSON.parse(emp);

- **How to convert JavaScript object to JSON text?**
  By using method stringify
- var jsonText= JSON.stringify(myJSObject);

https://www.npmjs.com/package/json-serverr

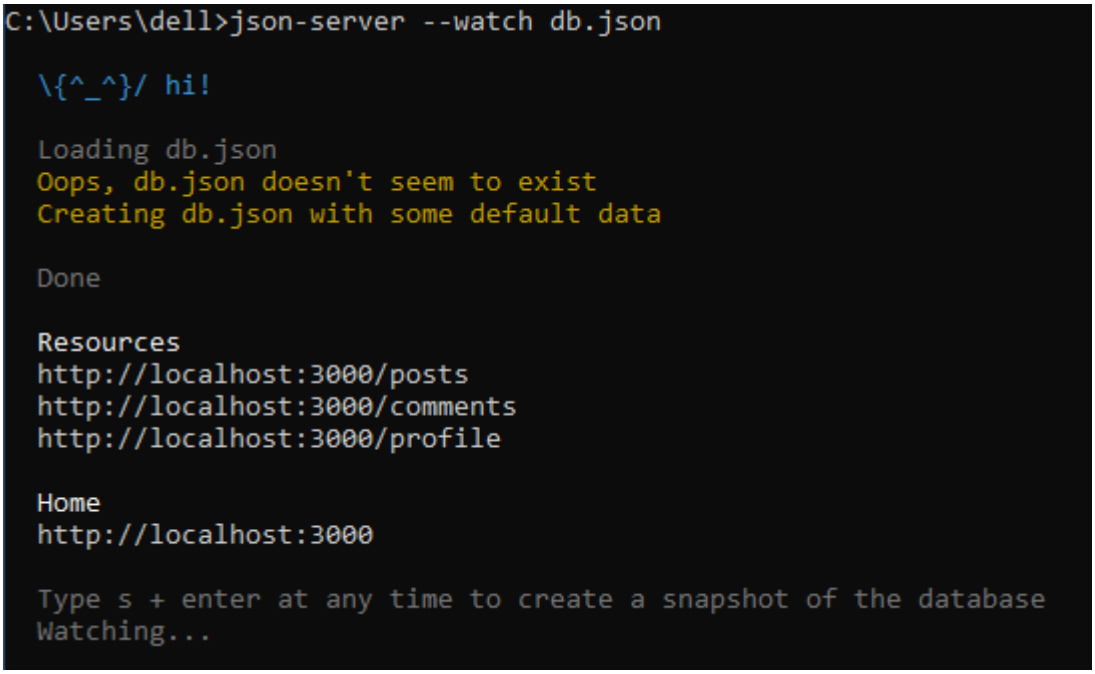
- We need to create a sample database with name db.json
- Inorder to create a database we need to follow to 2 steps:-

1. Install json server

npm install -g json-server

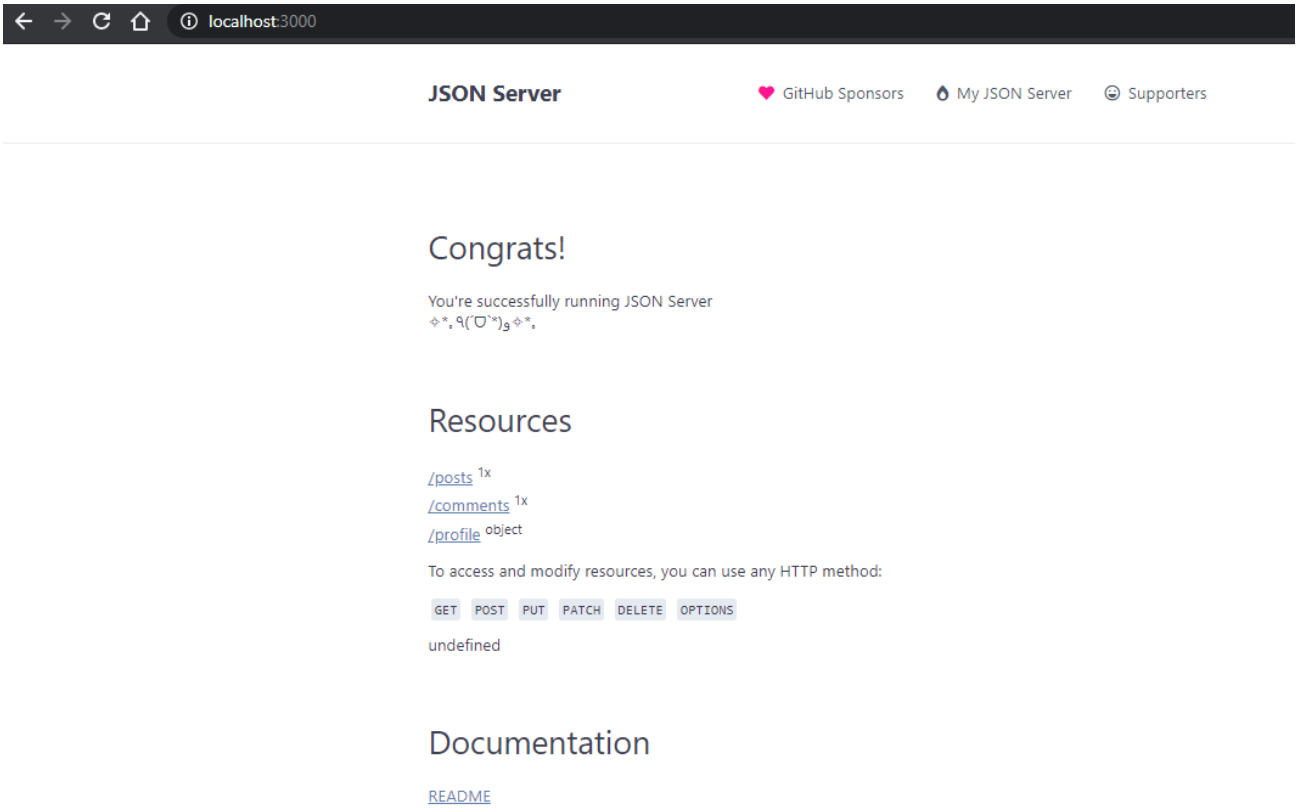
2 Start json server

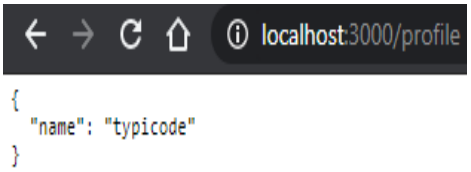
json-server --watch db.json

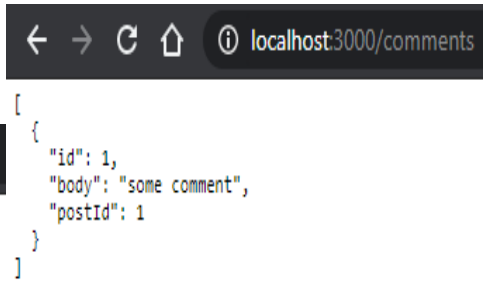


Copy the above link and paste it in browser.

the following window is opened as shown below







Create a project to demonstrate the working of json server with an login page??

Index.html:-

```
<!doctype html>
<html lang="en">
<head>
  <meta charset="utf-8">
  <title>Cp</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>
  <table>
```

```
    <tr>
      <td>
      <img src="./assets/1a.png" height="70" width="300">
      </td>
    </tr>
    <tr>
      <td>
      </td>
    </tr>
    <tr>
      <td>
        <app-root></app-root>
      </td>
    </tr>
    <tr>
      <td></td>
    </tr>
  </table>
</body>
</html>
```

app.module.ts:-

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms'
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { HomeComponent } from './home/home.component';
import { AboutusComponent } from './aboutus/aboutus.component';
import { ContactComponent } from './contact/contact.component';
import { LoginComponent } from './login/login.component';
import { LoginService } from './login.service';
import {HttpClient,HttpClientModule} from '@angular/common/http';
@NgModule({
```

```
  declarations: [
    AppComponent,
    HomeComponent,
    AboutusComponent,
    ContactComponent,
    LoginComponent,
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
     FormsModule,
     HttpClientModule
  ],
  providers: [LoginService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app-routing.module.ts:-

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { HomeComponent } from './home/home.component';
import { AboutusComponent } from './aboutus/aboutus.component';
import { ContactComponent } from './contact/contact.component';
import { LoginComponent } from './login/login.component';

const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'home', component: HomeComponent },
  { path: 'about', component: AboutusComponent },
  { path: 'contact', component: ContactComponent},
  { path: 'login', component: LoginComponent}
];
```

```
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

(no changes in app.component.ts)

App.component.html:-

```
<a routerLinkActive="active" routerLink="/home">Home</a> |
<a routerLinkActive="active" routerLink="/about">Aboutus</a> |
<a routerLinkActive="active" routerLink="/contact">Contactus</a> |
<a routerLinkActive="active" routerLink="/login">Login</a> |
<router-outlet></router-outlet>
```

login.service.ts:-

```
import { Injectable } from '@angular/core';
import {HttpClient} from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class LoginService {
  constructor(private http:HttpClient) { }
  GetStatus()
  {
    debugger
   return this.http.get("http://localhost:3000/login");
  }
}
```

login.component.ts:-

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

@Component({
  selector: 'app-login',
  templateUrl: './login.component.html',
  styleUrls: ['./login.component.css']
})
export class LoginComponent implements OnInit{
  title = 'jsonlogin';
  UserName:string;
```

```
  Password:string;
  IsValid:boolean;
  IsInValid:boolean;
  ngOnInit(){  }
  OnSubmit()   {
     if(this.UserName=="Admin" && this.Password=="Admin")
     {
       this.IsValid=true;
       this.IsInValid=false;
     }
     else
     {
       this.IsInValid=true;
       this.IsValid=false;
     }
  }
}
```

login.component.html:-

```
<div>
Enter UserName <input type="text" [(ngModel)]="UserName"> <br>
Enter Password
<input type="password" [(ngModel)]="Password">
<br>
<button (click)="OnSubmit()">Signin</button>
<span *ngIf="IsValid">
Valid User
</span>
<span *ngIf="IsInValid">
   InValid User
</span>
</div>
```

db.json:-

```
{
  "login": [
    {
      "username": "Admin",
      "password": "Admin"
    }
  ]
```

}

This has to be run (json-server –watch db.json) and then run the angular project.

db.json

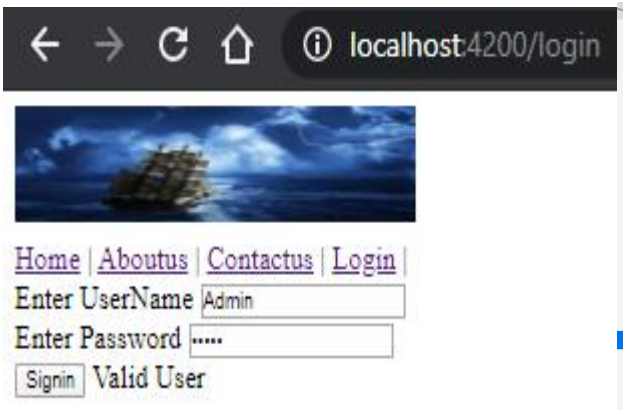
```
{
  "login": [
    {
      "username": "Admin",
      "password": "Admin"
    }
  ]
}
```

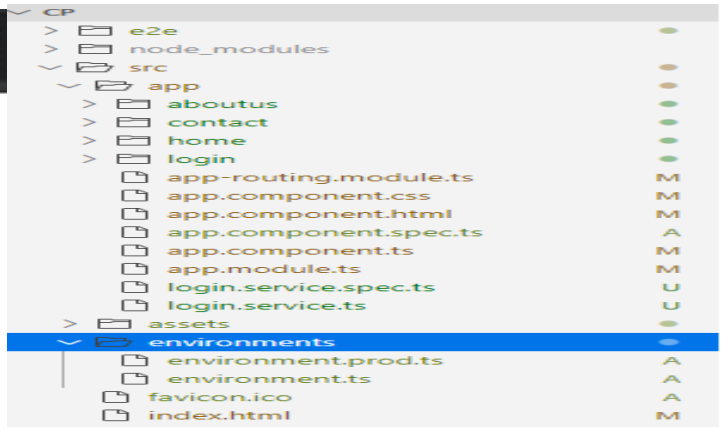
```
DEBUG CONSOLE  PROBLEMS  OUTPUT  TERMINAL                1: node
PS D:\angular\jsonlogin> json-server --watch db.json

  \{^_^}/ hi!

  Loading db.json
  Done

  Resources
  http://localhost:3000/login

  Home
  http://localhost:3000

  Type s + enter at any time to create a snapshot of the database
  Watching...

GET /login 200 14.304 ms - 60
```

localhost:3000/login

```
[
  {
    "username": "Admin",
    "password": "Admin"
  }
]
```

Output: **If username== password then home page will open.**





Create a project with name empdetailsproject to store data in json server as user enters in reg form?

1. create register component: **ng g c register**
2. create a service: **ng g s register**
3. install json server: **npm install -g json-server**
4. start json server: **json-server --watch db.json**
5. edit db.json

```
{
 "emp": [
  {
```

```
    "id": 101,
    "ename": "anil",
    "salary": 20000
  }
 ]
}
```

code for **app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { RegisterComponent } from './register/register.component';
import {FormsModule} from '@angular/forms';
import {RegisterService} from './register.service';
import { from } from 'rxjs';
import{HttpClientModule} from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent,
    RegisterComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [RegisterService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

code for **app.component.html**

```
<app-register></app-register>
<router-outlet></router-outlet>
```

code for **register.service.ts**

```
import { Injectable } from '@angular/core';
import{HttpClientModule,HttpClient} from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class RegisterService {
  constructor(private http:HttpClient) { }
  SaveUser(empformdata:any)
  {
    debugger
    return this.http.post("http://localhost:3000/emp",empformdata);
  }
  GetUser()
  {
    debugger
    return this.http.get("http://localhost:3000/emp");
```

```
   }
}
```

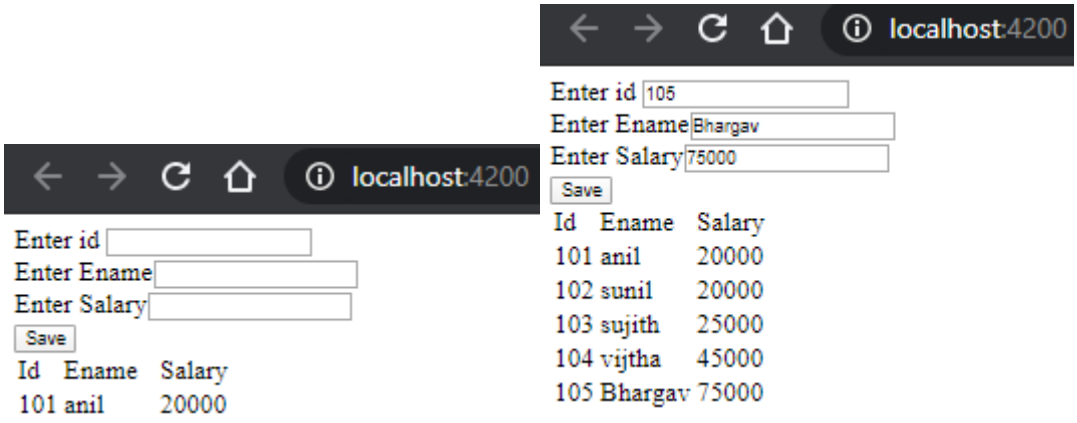
code for **register.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { RegisterService } from '../register.service';
@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  id:number;
  ename:string;
  salary:number;
  emplist:any;
  formdata:any;
  constructor(private reg:RegisterService) { }
  ngOnInit(): void
  {
    debugger
    this.GetUser();
  }
  GetUser(){
    this.reg.GetUser().subscribe(res=>
      {
        this.emplist=res;
    });
  }
  OnSubmit()
  {
    debugger
    this.formdata={id:this.id,ename:this.ename,salary:this.salary}
    this.reg.SaveUser(this.formdata).subscribe(res=>
      {
        this.GetUser();
      });
  }
}
```

code for **register.component.html**

```
Enter id <input type="number" [(ngModel)]="id"><br>
Enter Ename<input type="text" [(ngModel)]="ename"><br>
Enter Salary<input type="number" [(ngModel)]="salary"><br>
<input type="button" (click)="OnSubmit()" value="Save"><br>
<table>
    <tr>
        <td>Id</td>
        <td>Ename</td>
        <td>Salary</td>
    </tr>
        <tr *ngFor="let user of emplist">
```

```
            <td>{{ user.id }}</td>
            <td>{{ user.ename}}</td>
            <td>{{user.salary}}</td>
        </tr>
</table>
```



Code for **db.json** after adding the data:

```
{
  "emp": [
    {
      "id": 101,
      "ename": "anil",
      "salary": 20000
    },
    {
      "id": 102,
      "ename": "sunil",
      "salary": 20000
    },
    {
      "id": 103,
      "ename": "sujith",
      "salary": 25000
    },
    {
      "id": 104,
      "ename": "vijtha",
      "salary": 45000
    },
    {
      "id": 105,
      "ename": "Bhargav",
      "salary": 75000
    }
  ]
}
```

Debugging

Fn+f12→ Inspect
F8→ directly goto debugger
F10 or f11 → move to next line

Create a project with name empdetailseditdel to store data in json server as user enters in reg form and to perform updation, deletion of data as per requirement?

1. create register component: **ng g c register**
2. create a service: **ng g s register**
3. install json server: **npm install -g json-server**
4. start json server: **json-server --watch db.json**
5. edit db.json

```
{
  "emp": [
    {
      "id": 101,
      "ename": "anil",
      "salary": 20000
    },
    {
      "id": 102,
      "ename": "sunil",
      "salary": 20000
    },
    {
      "id": 103,
      "ename": "sujith",
      "salary": 25000
    },
    {
      "id": 104,
      "ename": "vijtha",
      "salary": 45000
    },
    {
      "id": 105,
      "ename": "Bhargav",
      "salary": 75000
    }
  ]
}
```

code for **app.module.ts**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { RegisterComponent } from './register/register.component';
import {FormsModule} from '@angular/forms';
import {RegisterService} from './register.service';
import { from } from 'rxjs';
import{HttpClientModule} from '@angular/common/http';
@NgModule({
  declarations: [
```

```
    AppComponent,
    RegisterComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    HttpClientModule
  ],
  providers: [RegisterService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

code for **app.component.html**

```
<app-register></app-register>
<router-outlet></router-outlet>
```

code for **register.service.ts**

```
import { Injectable } from '@angular/core';
import{HttpClientModule,HttpClient} from '@angular/common/http';
@Injectable({
  providedIn: 'root'
})
export class RegisterService {
  constructor(private http:HttpClient) { }
  SaveUser(empformdata:any)
  {
    debugger
    return this.http.post("http://localhost:3000/emp",empformdata);
  }
  GetUser()
  {
    debugger
    return this.http.get("http://localhost:3000/emp");
  }
   SearchEmpById(id:number)
  {
       return this.http.get("http://localhost:3000/emp/"+id);
  }
   DeleteEmp(id:number)
  {
     return this.http.delete("http://localhost:3000/emp/"+id);
  }
  updateuser(formdata:any)
  {
     debugger
     return this.http.put("http://localhost:3000/emp/"+formdata.id,formdata);
  }
}
```

code for **register.component.ts**

```
import { Component, OnInit } from '@angular/core';
import { RegisterService } from '../register.service';
```

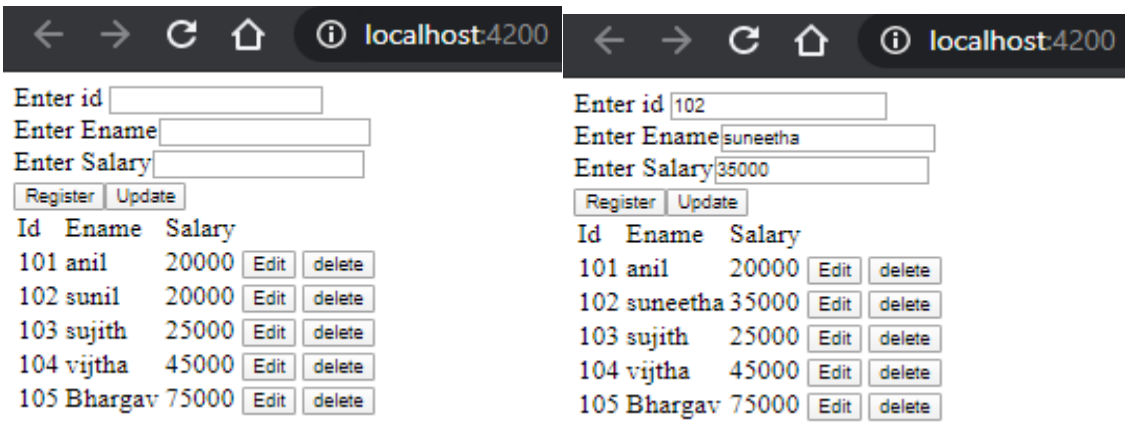
```
@Component({
  selector: 'app-register',
  templateUrl: './register.component.html',
  styleUrls: ['./register.component.css']
})
export class RegisterComponent implements OnInit {
  id:number;
  ename:string;
  salary:number;
  emplist:any;
  formdata:any;
  constructor(private reg:RegisterService) { }
  ngOnInit(): void
  {
    debugger
    this.GetUser();
  }
  GetUser(){
    this.reg.GetUser().subscribe(res=>
      {
        this.emplist=res;
    });
  }
  OnSubmit()
  {
    debugger
    this.formdata={id:this.id,ename:this.ename,salary:this.salary}
    this.reg.SaveUser(this.formdata).subscribe(res=>
      {
        this.GetUser();
      });
  }
  GetEmpById(id)
  {
    this.reg.SearchEmpById(id).subscribe(res=>
      {
        debugger
        this.emplist=res;
        this.id=this.emplist.id;
        this.ename=this.emplist.ename;
        this.salary=this.emplist.salary;
      });
  }
  DeleteEmp(id)
  {
  this.reg.DeleteEmp(id).subscribe(r=>
  {
    this.GetUser();
  });
  }
  OnUpdate()
  {
    debugger
    this.formdata={id:this.id,ename:this.ename,salary:this.salary};
```

```
    this.reg.updateuser(this.formdata).subscribe(r=>
      {
        this.GetUser();
     });
  }
}
```

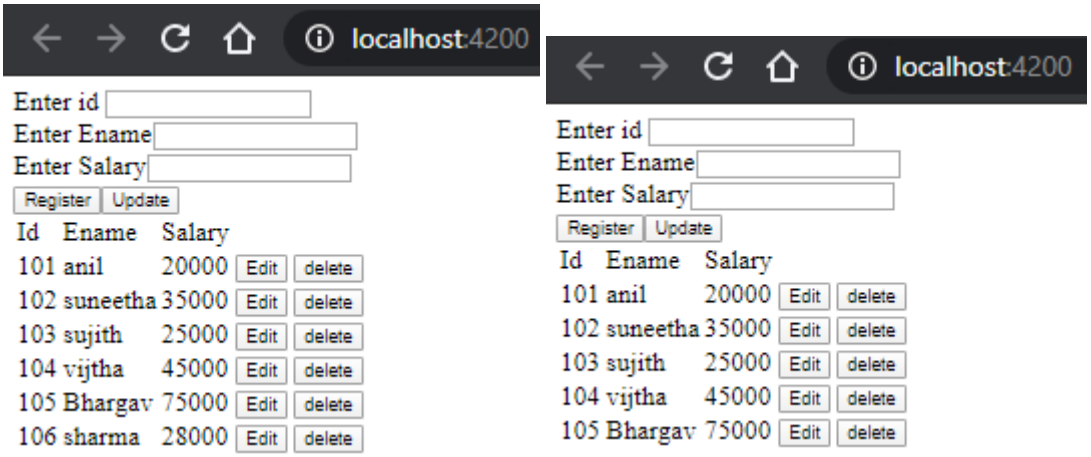
code for **register.component.html**

```
Enter id <input type="number" [(ngModel)]="id"><br>
Enter Ename<input type="text" [(ngModel)]="ename"><br>
Enter Salary<input type="number" [(ngModel)]="salary"><br>
<input type="button" (click)="OnSubmit()" value="Register">
<input type="button" (click)="OnUpdate()" value="Update"><br>
<table>
    <tr>
        <td>Id</td>
        <td>Ename</td>
        <td>Salary</td>
    </tr>
        <tr *ngFor="let user of emplist">
            <td>{{ user.id }}</td>
            <td>{{ user.ename}}</td>
            <td>{{user.salary}}</td>
            <td><input type="button" (click)="GetEmpById(user.id)" value="Edit"></td>
            <td><input type="button" (click)="DeleteEmp(user.id)" value="delete"></td>
        </tr>
</table>
```

Output before and after updation:



Output before and after deletion:



Pipes or Filters

Angular JS 1.x has filters which are used for many common uses like formatting dates, string display in uppercase or lowercase etc. These filters are knows as "Pipes" in Angular

Pipes allows us to change the data before display to the user.

Normally a pipe takes the data and transforms this input to the desired output. There are two types of pipes in Angular

1) built-in pipes

2) Custom pipes

Built in pipes are already created in angular ,we just need to call that if we want to use it. If Built-in pipes are not sufficient for project needs we are free to create custom pipes

There are 10 built in pipes in angular

1) Uppercase

2)Lowercase

3)Titlecase

4)date

5)Currency

6)Number

7)date

8)slice

9)json

10)async

Syntax : {{ expression | pipename:arguments }}

Whenever you want to use pipe just use pipe name followed by pipe(|) character. Pipes are methods behind the scene , so methods can take arguments in same manner pipe can also take argument which you will pass after colon(:)

Chaining of Pipe: Sometime we want to use more than one pipe on same expression so this is called chaining of pipe. So here suppose we want to apply date and uppercase together on dob express

{{dob | date | uppercase}}

In this case output of date pipe will act like a input for uppercase pipe

Pipes in Angular:-

Pipes (|) in Angular are used to transform the data before displaying it in a browser. Angular provides a lot of built-in pipes to translate the data before displaying it into the browser

Angular lets us extend its features we can even create custom pipes in Angular. So in this article, we are going to learn the built-in pipes which are provided by Angular

Pipes can be applied to:-

- Pipes which we can apply to Strings
- Pipes which we can apply to Numbers
- Pipes which we can apply to Date

Pipes which we can apply to String

- **Uppercase** - We can use this pipe to convert the data-bound expression to uppercase.
- **Lowercase** - We can use this pipe to convert the data-bound expression to lowercase.
- **Slice** – We can use this pipe to extract the substring from the data-bound expression. This pipe takes two parameters - starting index of a string and the last index of the string.
- **Titlecase:** Pipes which we can apply to Numbers. We can use these pipes to format the numbers
- **Number –** We can use this pipe to convert the number to a different format.
- **Currency** – We can use this pipe to show currency symbol. This pipe takes two parameters - currency i.e USD, EURO, and GTP; and the Boolean value which decides whether we want to display the currency symbol or not. See the below code and output screenshot for more information.

NOTE: Filters or pipes are used to format the output before displaying on th browser.

Example 1:

app.component.ts

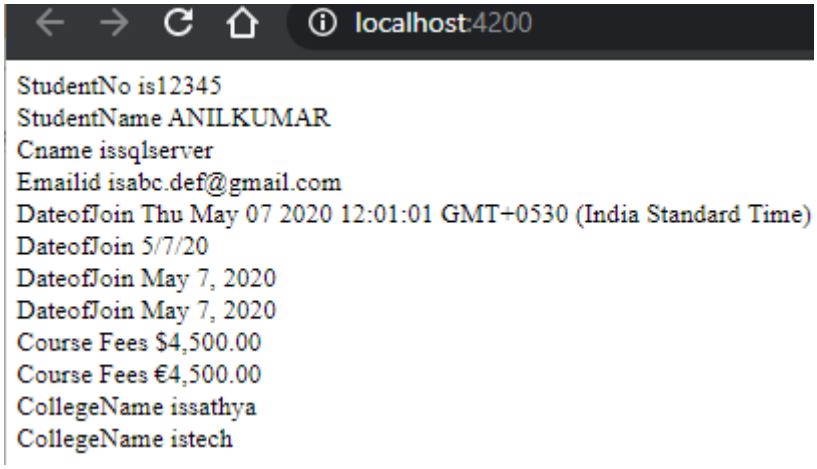
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'pipedemo';
  public Sno:number=12345;
  public Sname:string="anilkumar";
  public Cname:string="SQLSERVER";
  public Emailid:string="abc.def@gmail.com";
```

```
  public JoinDate:Date=new Date();
  public Fees:number=4500;
  public CollegeName:string="sathyatechnologies";

}
```

app.component.html:-

```
<div>
  StudentNo is{{Sno}}  <br>
  StudentName {{Sname|uppercase}}<br>
Cname is{{Cname|lowercase}}<br>
Emailid is{{Emailid}}<br>
DateofJoin {{JoinDate}}<br>
DateofJoin {{JoinDate| date:'shortDate'}}<br>
DateofJoin {{JoinDate| date:'longDate'}}<br>
DateofJoin {{JoinDate| date:'mediumDate'}}<br>
Course Fees {{Fees|currency:'USD':true}}<br>
Course Fees {{Fees|currency:'EUR':true}}<br>
CollegeName is{{CollegeName|slice: 0: 6}}<br>
CollegeName is{{CollegeName|slice: 6: 10}}
</div>
<router-outlet></router-outlet>
```

Output:



Example 2:

app.component.ts

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

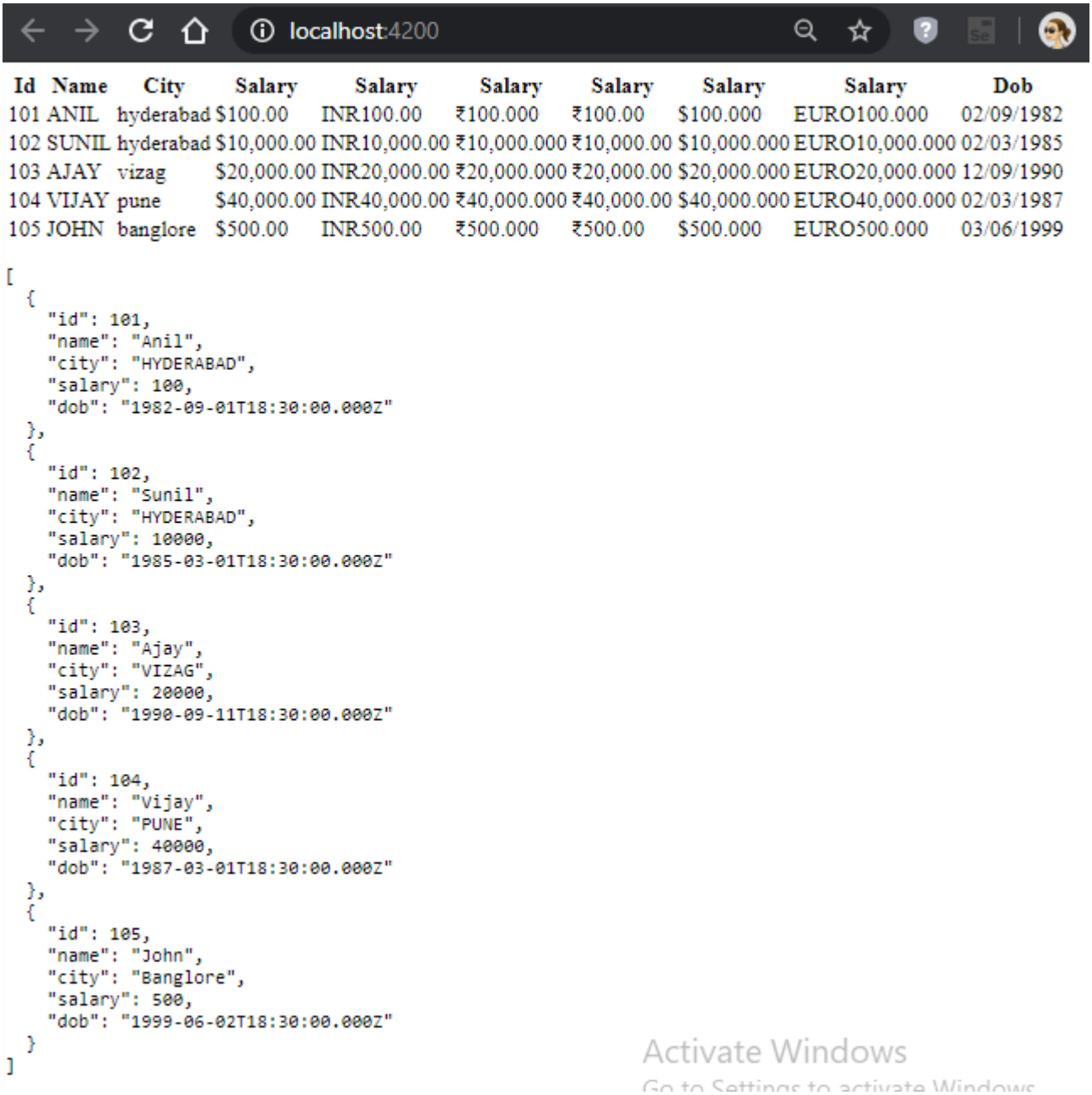
```
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'myp';
  users:any[]=[
    {id:101,name:'Anil',city:'HYDERABAD',salary:100,dob:new Date("09/02/1982")},
    {id:102,name:'Sunil',city:'HYDERABAD',salary:10000,dob:new Date("03/02/1985")},
    {id:103,name:'Ajay',city:'VIZAG',salary:20000,dob:new Date("09/12/1990")},
    {id:104,name:'Vijay',city:'PUNE',salary:40000,dob:new Date("03/02/1987")},
    {id:105,name:'John',city:'Banglore',salary:500,dob:new Date("06/03/1999")},
  ];
}
```

app.component.html

```
<table>
  <tr>
    <th>Id</th>
    <th>Name</th>
    <th>City</th>
    <th>Salary</th>
    <th>Salary</th>
    <th>Salary</th>
    <th>Salary</th>
    <th>Salary</th>
    <th>Salary</th>
    <th>Dob</th>
  </tr>
  <tr *ngFor="let user of users">
<td>{{user.id}}</td>
<td>{{user.name|uppercase}}</td>
<td>{{user.city|lowercase}}</td>
<td>{{user.salary|currency}}</td>
<td>{{user.salary|currency:'INR':'code'}}</td>
<td>{{user.salary|currency:'INR':'symbol':'3.3'}}</td>
<td>{{user.salary|currency:'INR'}}</td>
<td>{{user.salary|currency:'USD':'symbol':'3.3'}}</td>
<td>{{user.salary|currency:'EURO':'symbol':'3.3'}}</td>
<td>{{user.dob|date:'dd/MM/yyyy'}}</td>
  </tr>
</table>
<pre>
{{users|json}}
</pre>
```

```
<router-outlet></router-outlet>
```

Output:



custom pipes:- The pipes that was created by the programmer depending on the user requirement are called as ciustom pipes

syn to create pipe:-

ng g p pipes/pipename

ex:ng g p pipes/age

create a project to display age based on dob given by user.

1. Create a pipe with name age

   ng g p pipes/age

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { AgePipe } from './pipes/age.pipe';
```

```
@NgModule({
  declarations: [
    AppComponent,
    AgePipe
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'userloginngif';
  users:any[]=[
    {id:101,name:'Anil',city:'HYDERABAD',salary:100,dob:new Date("09/02/1982")},
    {id:102,name:'Sunil',city:'HYDERABAD',salary:10000,dob:new Date("03/02/1985")},
    {id:103,name:'Ajay',city:'VIZAG',salary:20000,dob:new Date("09/12/1990")},
    {id:104,name:'Vijay',city:'PUNE',salary:40000,dob:new Date("03/02/1987")},
    {id:105,name:'John',city:'Banglore',salary:500,dob:new Date("06/03/1999")},
  ];
}
```

App.component.html

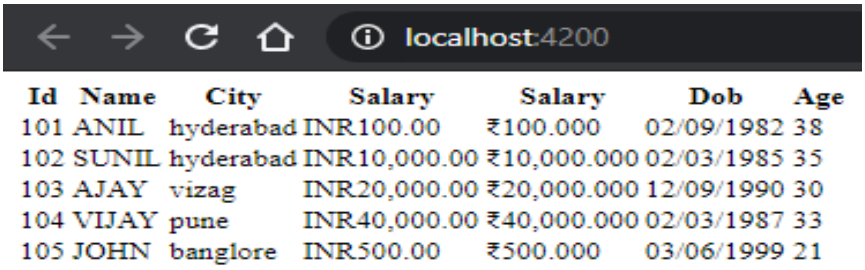
```
<table>
  <tr>
    <th>Id</th>
    <th>Name</th>
    <th>City</th>
    <th>Salary</th>
    <th>Salary</th>
    <th>Dob</th>
    <th>Age</th>
  </tr>
  <tr *ngFor="let user of users">
    <td>{{user.id}}</td>
    <td>{{user.name|uppercase}}</td>
    <td>{{user.city|lowercase}}</td>
    <td>{{user.salary|currency:'INR':'code'}}</td>
    <td>{{user.salary|currency:'INR':'symbol':'3.3'}}</td>
   <td>{{user.dob|date:'dd/MM/yyyy'}}</td>
    <td>{{user.dob|age}}</td>
  </tr>
</table>
```

```
<pre>
{{users|json}}
</pre>
<router-outlet></router-outlet>


<router-outlet></router-outlet>
```

age.pipe.ts

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'age'
})
export class AgePipe implements PipeTransform {

  transform(value: any, args?: any): any {
    let currentyear:any=new Date().getFullYear();
    let uy:any=new Date(value).getFullYear();
    let userage=currentyear-uy;
    return userage;
  }

}
```

Output:




```
[
  {
    "id": 101,
    "name": "Anil",
    "city": "HYDERABAD",
    "salary": 100,
    "dob": "1982-09-01T18:30:00.000Z"
  },
  {
    "id": 102,
    "name": "Sunil",
    "city": "HYDERABAD",
    "salary": 10000,
    "dob": "1985-03-01T18:30:00.000Z"
  },
  {
    "id": 103,
    "name": "Ajay",
    "city": "VIZAG",
    "salary": 20000,
    "dob": "1990-09-11T18:30:00.000Z"
  },
  {
    "id": 104,
    "name": "Vijay",
    "city": "PUNE",
    "salary": 40000,
    "dob": "1987-03-01T18:30:00.000Z"
  },
  {
    "id": 105,
    "name": "John",
    "city": "Banglore",
    "salary": 500,
    "dob": "1999-06-02T18:30:00.000Z"
  }
]
```


Angular Forms

Formtags:-

Redirection,Validation,submissions,Reset,Clear

Prerequisites:-

1. HTML
2. Javascript
3. CSS
4. Angular Components,Services,Templates,Databinding

Forms can be difficult to style. Not only do form elements display differently in different browsers, but most browsers limit the visual changes you can make to form controls.

These are of 2 types:

1. **Template Driven Forms**: Most of the code written in component template.

Step1: The Component template contains the html to collect

Step 2: The component class handles data binding to the collected data.

Step3: The collected data is sent to the server through the service.

2.Reactive Forms (Model Driven Approach): Most of the code written in component class.

Template-driven Forms	Reactive Forms
Easy to use	More flexible→More Complex scenaries
Similar to angular 1	Immutable data model
Two way data binding→Minimal Component code	Easier to perform an action on a value change.
Automatically tracks forms and input element state	Reactive Trasformations→Debounce tiem or distinct changed
	Easily add input elements dynamically
	Easier unit testing.

Template Driven Forms

- Easy to use and similar to angular JS forms.
- Two way data Binding with ngModel
- Bulky HTML and minimal Component code.
- Automatically tracks the form anddd form elements state and validity
- Unit testing is a challenge.
- Readability decreases with complex forms an validations
- Suitabl for simple scenarios.

Why Template Driven Forms?

In Normal forms, we only handlewith user data, but here in our template driven forms we get an object in that object we holding all form data like properties,Elements, Information,…etc.

class
{
  variable store the value
  method operation
}
==========
class :-consists of variables and methods
variable :- store the value
Method :- perform operation
object :- Allocate memory for variable
Reference :- accessability
object is collection of variables
collection is group of objects
==========
ngForm:- is a predefined class

Development Process of Template Driven Forms

Every First Thing if we want to handle any form in angular.
STEP 1: We need to import “formsmodule” from Angular/forms into “app.module.ts” & at the same time we need to place that “FormsModule” In import section then we are ready to access all directives, Properties related to our Form & Methods, Event Handling we can access.

create a project to display data as 2 different groups of ingormation (Student Details, Course Details)?

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule} from '@angular/forms';
```

```
import { AppRoutingModule } from ‘./app-routing.module’;
import { AppComponent } from ‘./app.component’;
import { from } from ‘rxjs’;

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

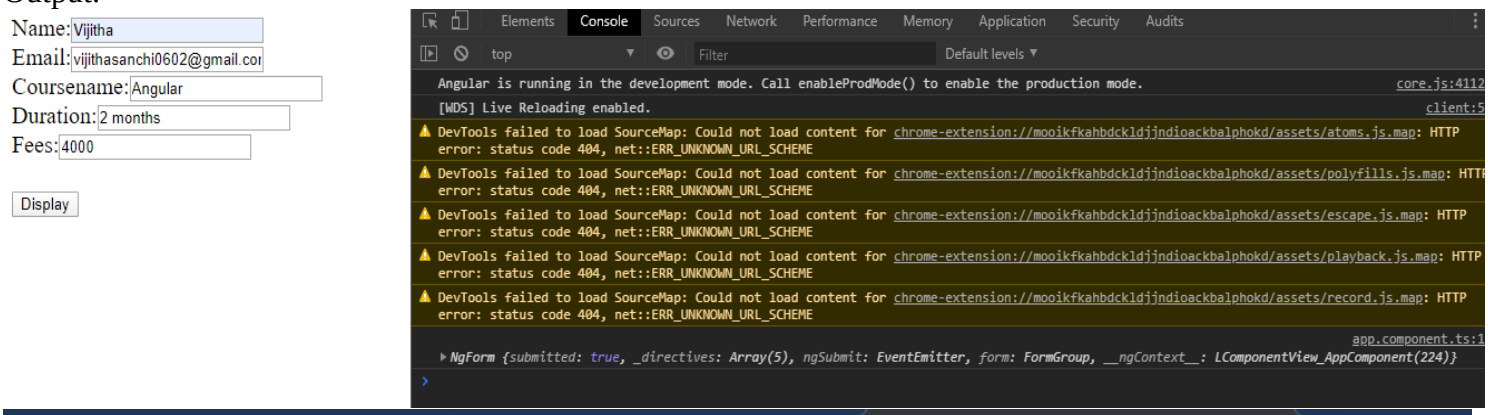
```
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'tfdemo';
  name:string;
  SaveData(data){
    console.log(data);
  }
}
```

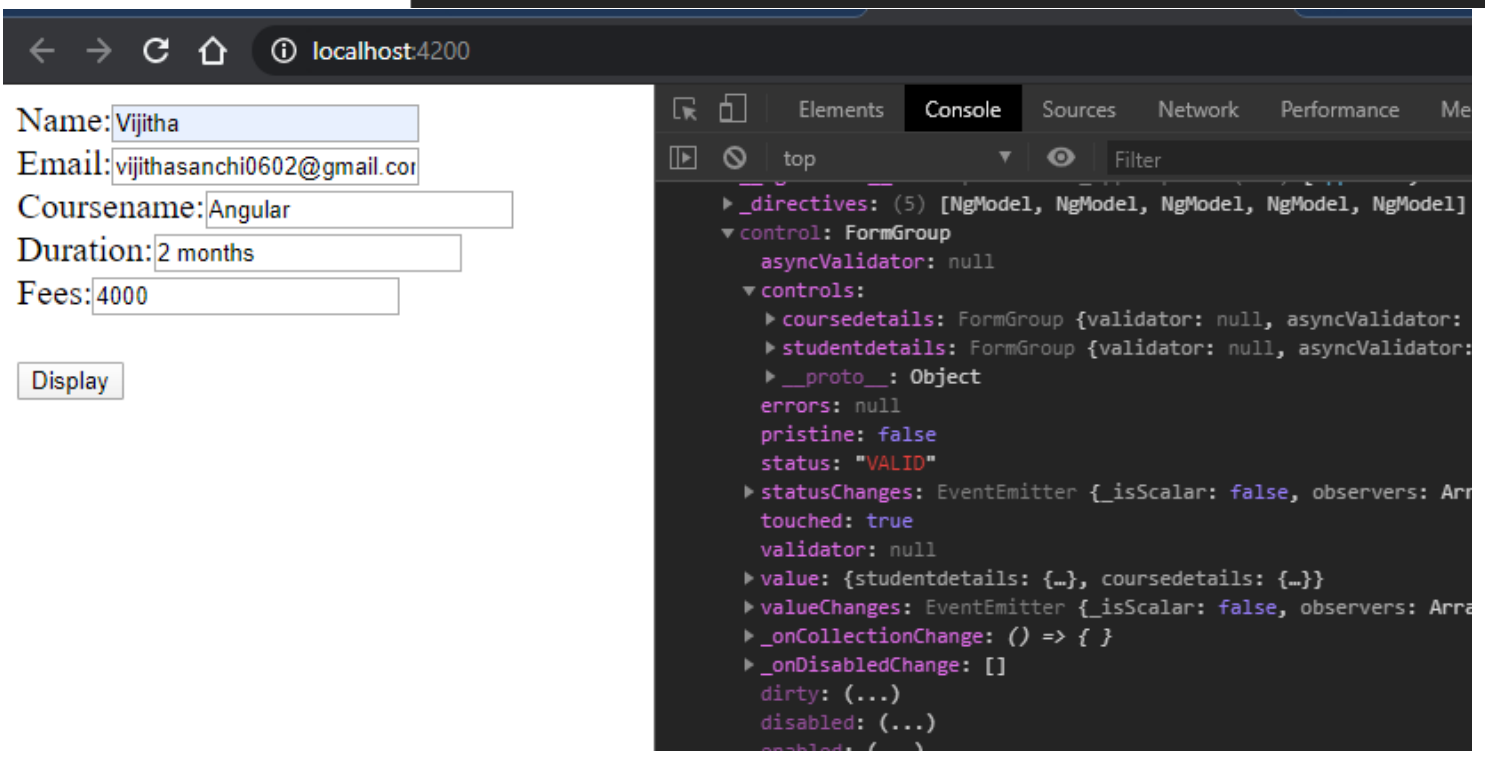
app.component.html

```
<form #myformref="ngForm" (ngSubmit)="SaveData(myformref)">
  <div ngModelGroup="studentdetails">
  <div class="row">
  <label for="name">Name:</label>
  <input type="text" name="Name" [(ngModel)]="name">
  </div>
  <div class="row">
  <label for="email">Email:</label>
  <input type="text" name="Email" ngModel>
  </div>
</div>
  <div ngModelGroup="coursedetails">
  <div class="row">
    <label for="Coursename">Coursename:</label>
    <input type="text" name="Coursename" ngModel>
    </div>
    <div class="row">
      <label for="Duration">Duration:</label>
      <input type="text" name="Duration" ngModel>
      </div>
      <div class="row">
```

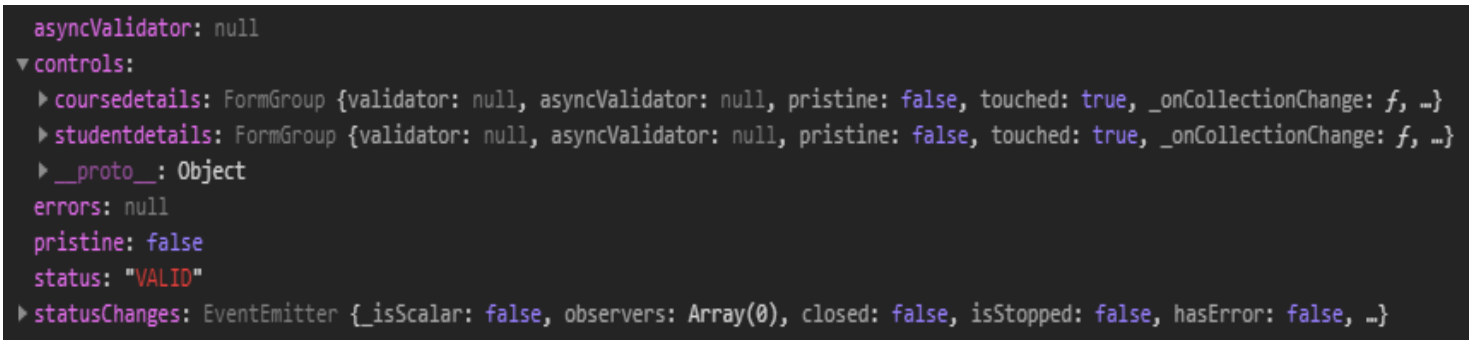
```
        <label for="Fees">Fees:</label>
        <input type="text" name="Fees" ngModel>
        </div>
      </div>
  <br>
  <button type="submit">Display</button>
  </form>
<router-outlet></router-outlet>
```

Output:

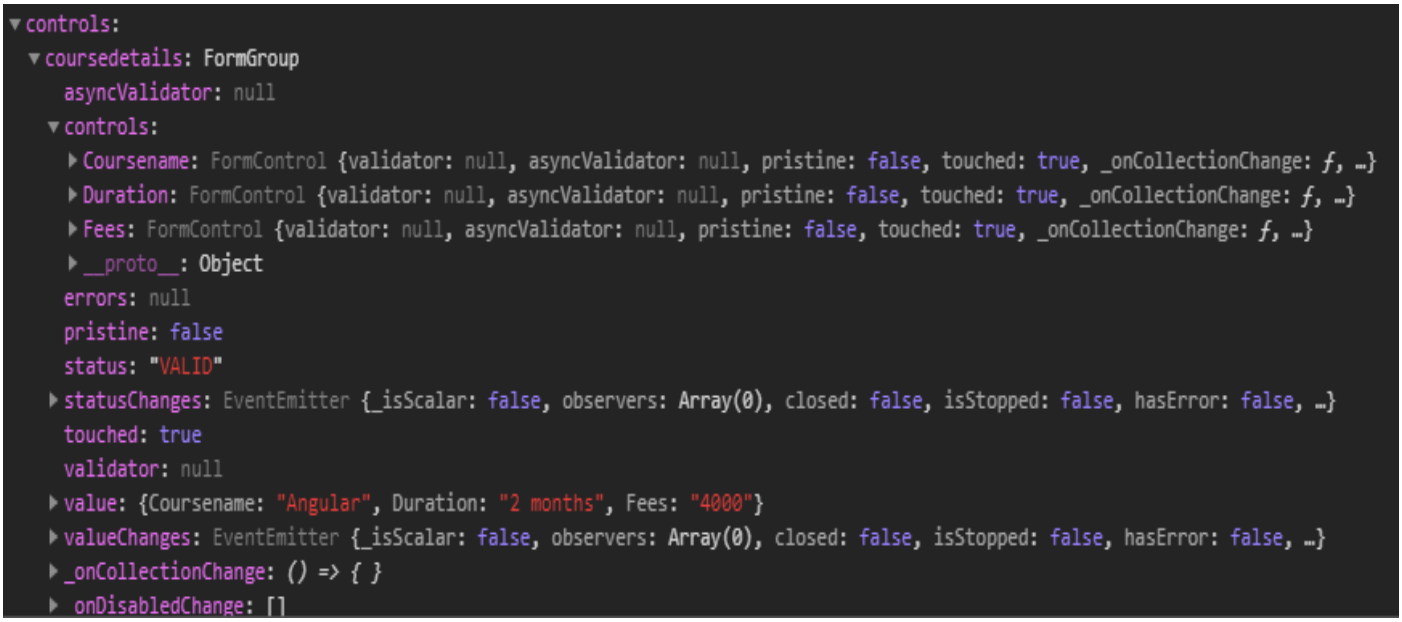




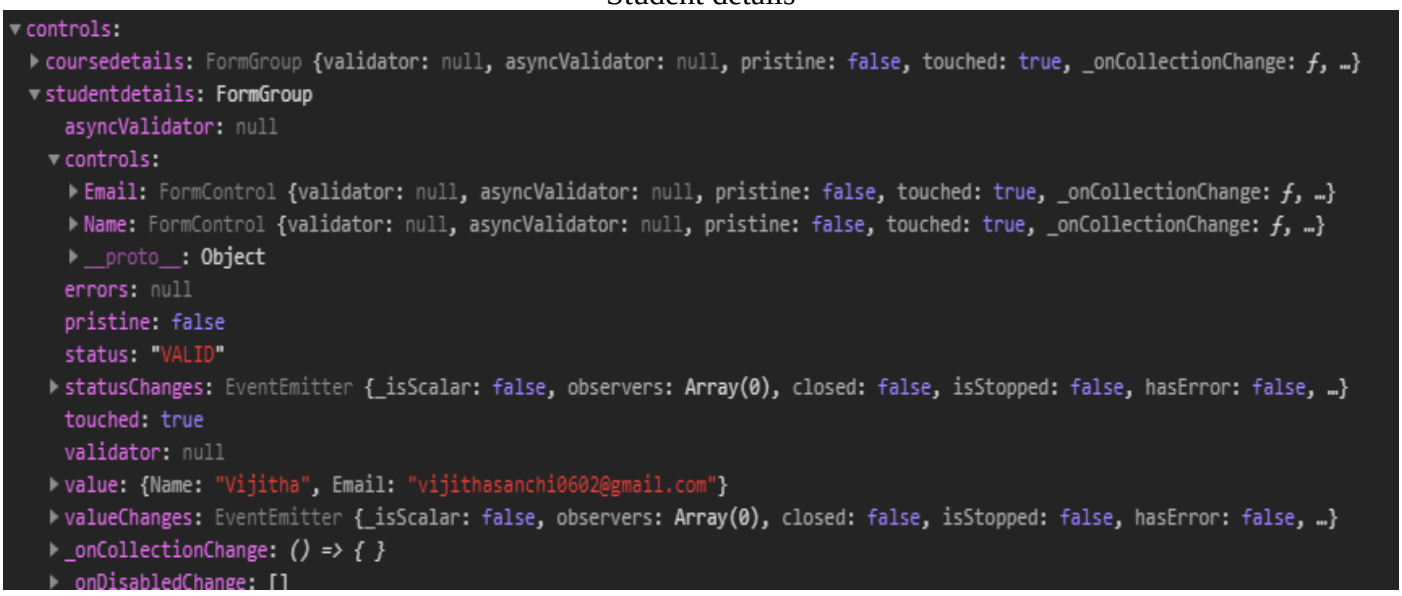
Press fn+f12 to view all this.



Course name



Student details

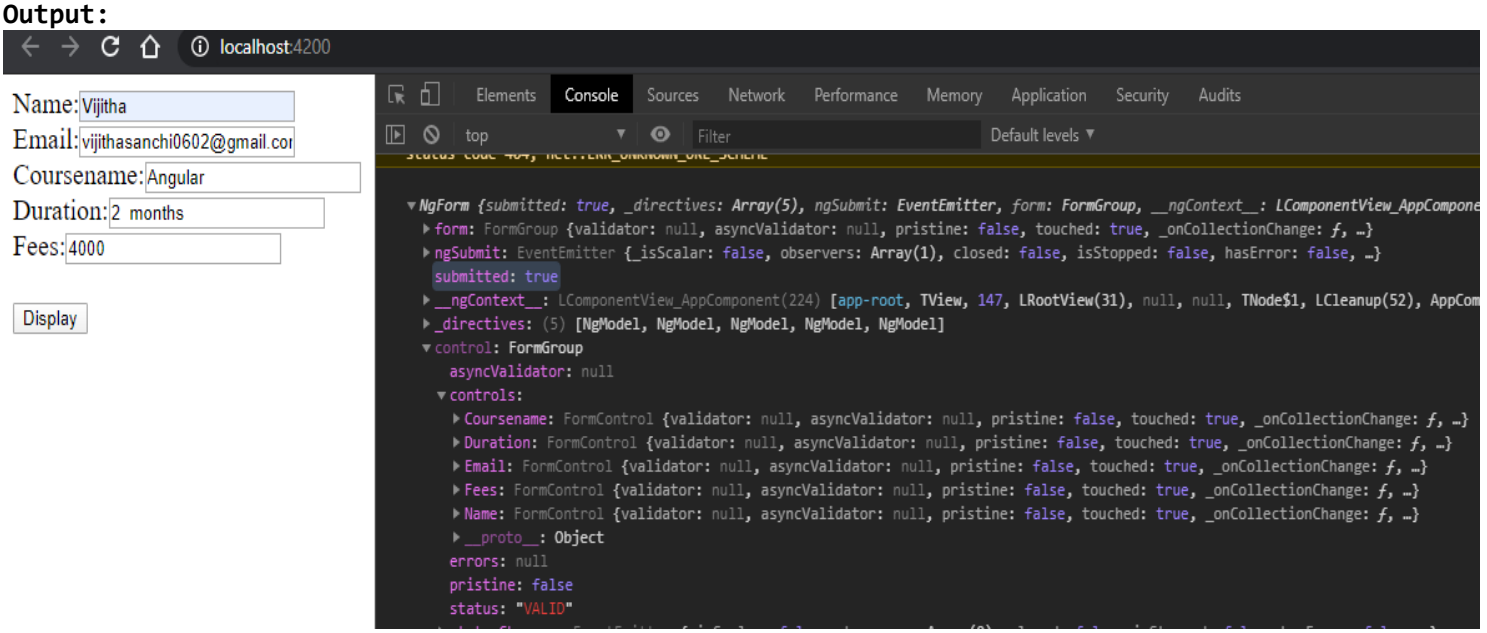


Another way of storing same data as a single group:

```
<form #myformref="ngForm" (ngSubmit)="SaveData(myformref)">
  <div class="row">
  <label for="name">Name:</label>
  <input type="text" name="Name" [(ngModel)]="name">
  </div>
  <div class="row">
  <label for="email">Email:</label>
  <input type="text" name="Email" ngModel>
  </div>
  <div class="row">
    <label for="Coursename">Coursename:</label>
    <input type="text" name="Coursename" ngModel>
    </div>
    <div class="row">
      <label for="Duration">Duration:</label>
```

```
      <input type="text" name="Duration" ngModel>
      </div>
      <div class="row">
        <label for="Fees">Fees:</label>
        <input type="text" name="Fees" ngModel>
        </div>
  <br>
  <button type="submit">Display</button>
  </form>
<router-outlet></router-outlet>
```

Output:



Reactive Forms

Create a project to display a reactive form which display the name,age,email and store all its values in object?

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule,ReactiveFormsModule} from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { from } from 'rxjs';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

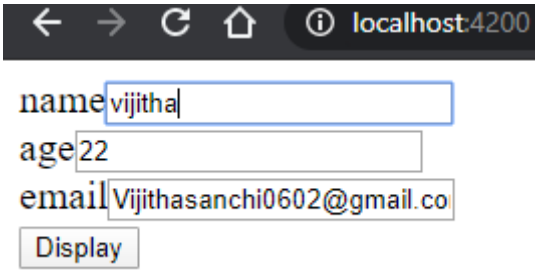
```
import { Component } from '@angular/core';
```

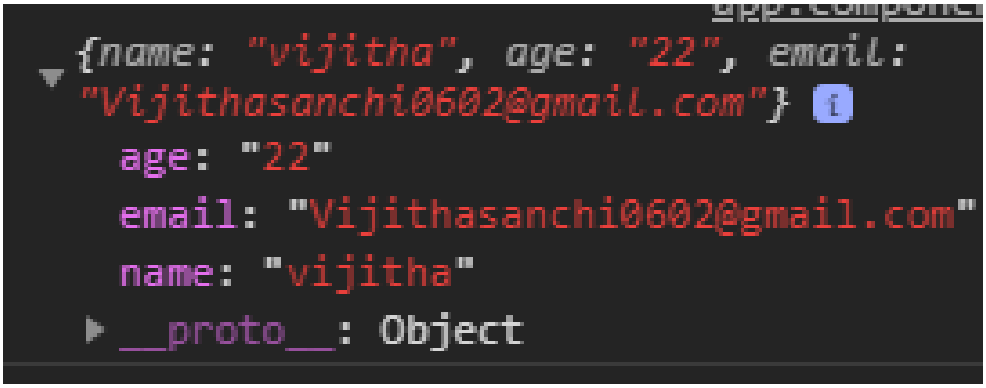
```
import {FormGroup, FormControl} from '@angular/forms';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'rfdemo';
  myform=new FormGroup(
    {
      name:new FormControl(),
      age:new FormControl(),
      email:new FormControl(),
    }
  );
  SaveData(){
    console.log(this.myform.value);
  }
}
```

app.component.html

```
<form [formGroup]="myform" (ngSubmit)="SaveData()">
  <div class="row">
    <label for="name">name</label>
    <input type="text"name="name" id="name" formControlName="name">
  </div>
  <div class="row">
    <label for="age">age</label>
    <input type="text"name="age" id="age" formControlName="age">
  </div>
  <div class="row">
    <label for="email">email</label>
    <input type="text"name="email" id="email" formControlName="email">
  </div>
  <button type="submit">Display</button>
</form>
<router-outlet></router-outlet>
```

Output:





Validations

Create a project to display a reactive form which display the name,age,email and store all its values in object and validate the data?

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule,ReactiveFormsModule} from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { from } from 'rxjs';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.ts

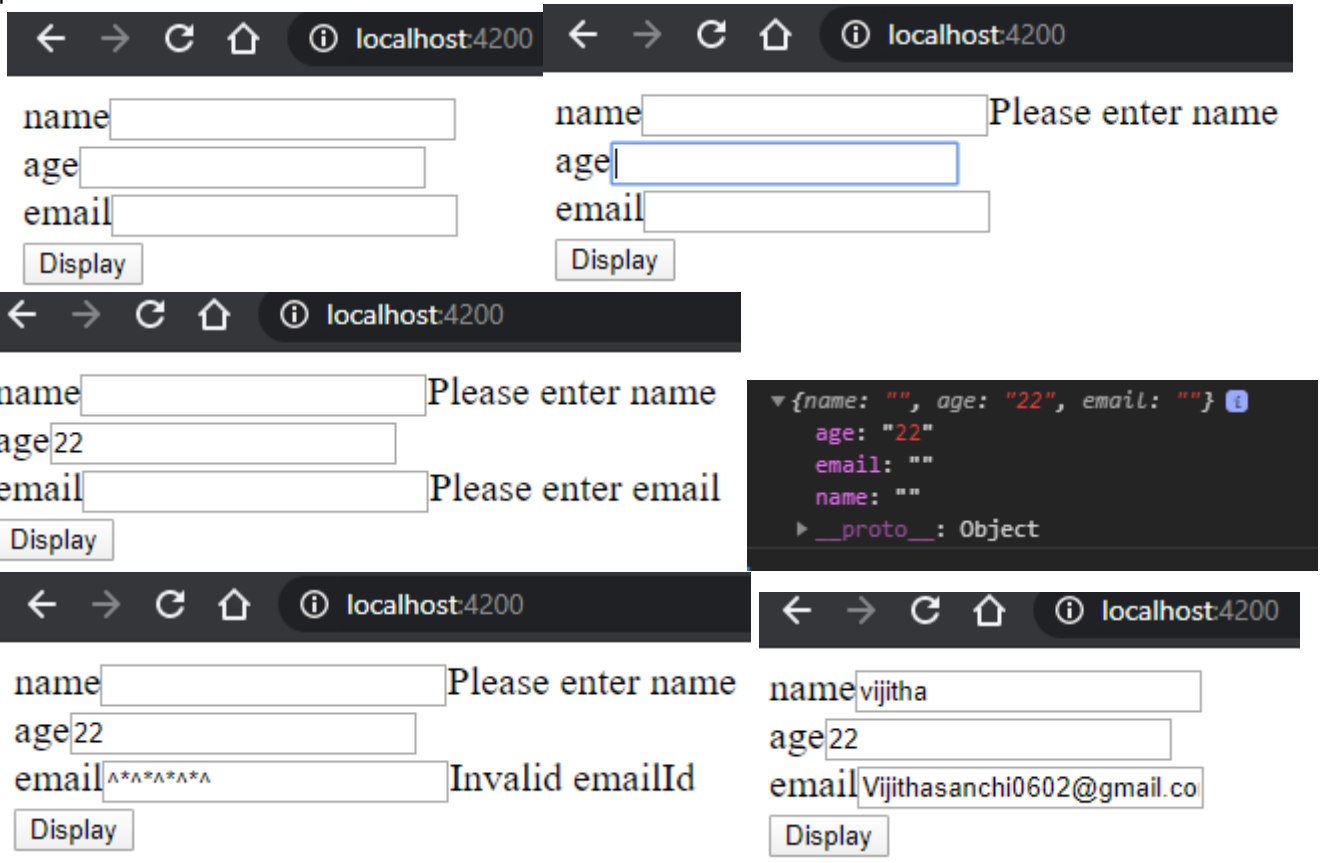
```
import { Component } from '@angular/core';
import {FormGroup, FormControl, Validators} from '@angular/forms';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'rfdemo';
  myform=new FormGroup(
    {
      name:new FormControl("",[Validators.required]),
      age:new FormControl("",[Validators.required]),
      email:new FormControl("",[Validators.required,Validators.email]),
    }
```

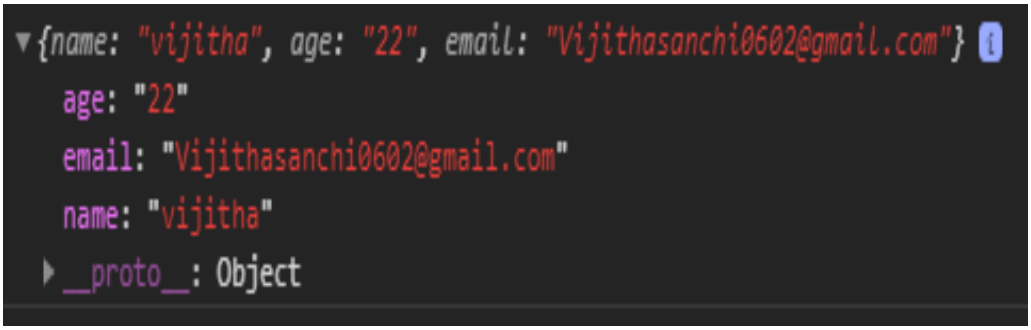
```
  );
  SaveData(){
    console.log(this.myform.value);
  }
}
```

app.component.html

```
<form [formGroup]="myform" (ngSubmit)="SaveData()">
  <div class="row">
    <label for="name">name</label>
    <input type="text"name="name" id="name" formControlName="name">
    <span *ngIf="myform.controls.name.invalid && myform.controls.name.touched">Please ente
r name</span>
  </div>
  <div class="row">
    <label for="age">age</label>
    <input type="text"name="age" id="age" formControlName="age">
  </div>
  <div class="row">
    <label for="email">email</label>
    <input type="text"name="email" id="email" formControlName="email">
    <span *ngIf="myform.controls.email.invalid && myform.controls.email.touched && myform.
controls.email.errors.required">Please enter email</span>
    <span *ngIf="myform.controls.email.invalid && myform.controls.email.touched && myform.
controls.email.errors.email">Invalid emailId</span>
  </div>
  <button type="submit">Display</button>
</form>
<router-outlet></router-outlet>
```

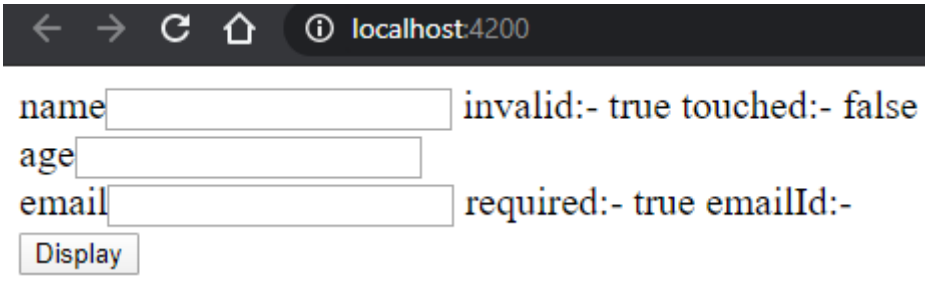
Output:

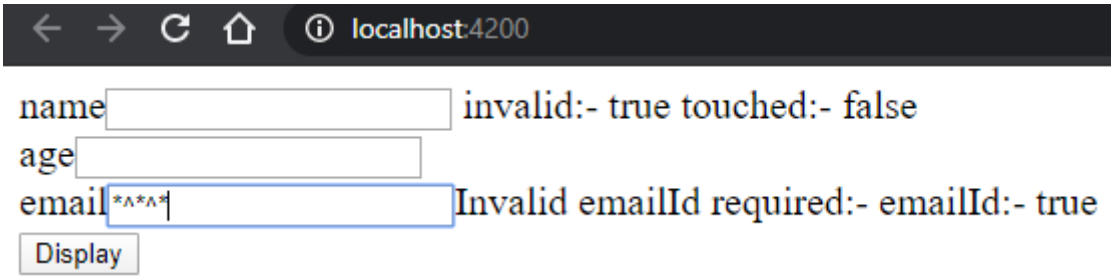


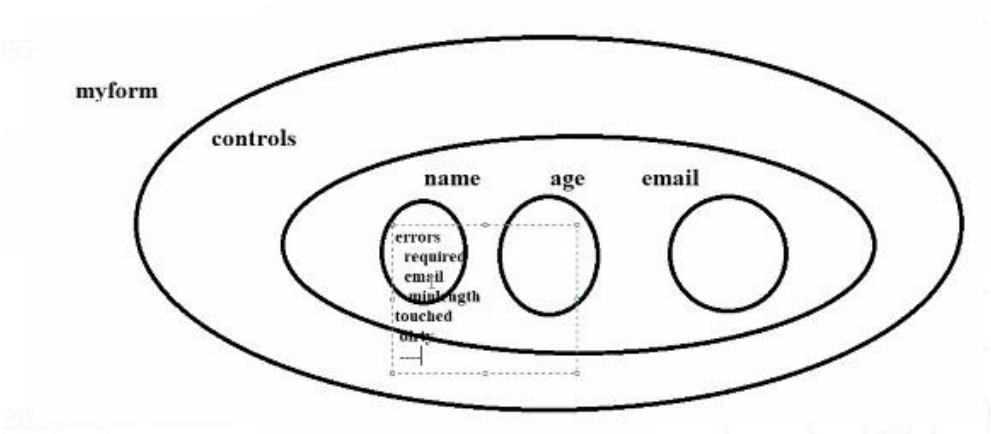


To check the use of myform.controls.name.invalid, myform.controls.name.touched, myform.controls.email.invalid,myform.controls.email.touched, myform.controls.email.errors.required, myform.controls.email.errors.email?

```
<form [formGroup]="myform" (ngSubmit)="SaveData()">
  <div class="row">
    <label for="name">name</label>
    <input type="text"name="name" id="name" formControlName="name">
    <span *ngIf="myform.controls.name.invalid && myform.controls.name.touched">Please ente
r name</span>
    invalid:- {{myform.controls.name.invalid}}
    touched:- {{myform.controls.name.touched}}
  </div>
  <div class="row">
    <label for="age">age</label>
    <input type="text"name="age" id="age" formControlName="age">
  </div>
  <div class="row">
    <label for="email">email</label>
    <input type="text"name="email" id="email" formControlName="email">
    <span *ngIf="myform.controls.email.invalid && myform.controls.email.touched && myform.
controls.email.errors.required">Please enter email</span>
    <span *ngIf="myform.controls.email.invalid && myform.controls.email.touched && myform.
controls.email.errors.email">Invalid emailId</span>
    required:- {{myform.controls.email.errors.required}}
    emailId:- {{myform.controls.email.errors.email}}
  </div>
  <button type="submit">Display</button>
</form>
<router-outlet></router-outlet>
```







Custom Validations

Command to create a class:**ng g cl Agerange**

Create a project to add custom validations to age of previous project?

app.module.ts

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {FormsModule,ReactiveFormsModule} from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { from } from 'rxjs';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    FormsModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

app.component.html

```
<form [formGroup]="myform" (ngSubmit)="SaveData()">
  <div class="row">
    <label for="name">name</label>
    <input type="text"name="name" id="name" formControlName="name">
    <span *ngIf="myform.controls.name.invalid && myform.controls.name.touched">Please ente
r name</span>
    </div>
  <div class="row">
    <label for="age">age</label>
```

```
    <input type="text"name="age" id="age" formControlName="age">
    <span *ngIf="myform.get('age').invalid && myform.get('age').touched">Please enter age
between 18-25</span>
  </div>
  <div class="row">
    <label for="email">email</label>
    <input type="text"name="email" id="email" formControlName="email">
    <span *ngIf="myform.controls.email.invalid && myform.controls.email.touched && myform.
controls.email.errors.required">Please enter email</span>
    <span *ngIf="myform.controls.email.invalid && myform.controls.email.touched && myform.
controls.email.errors.email">Invalid emailId</span>
   </div>
  <button type="submit">Display</button>
</form>
<router-outlet></router-outlet>
```

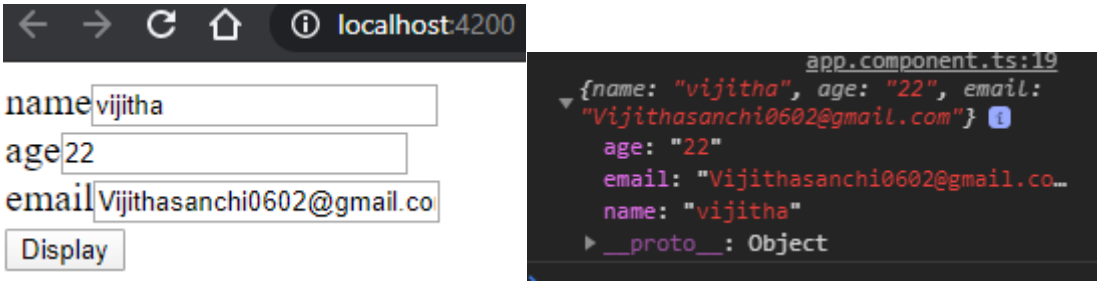
app.component.ts

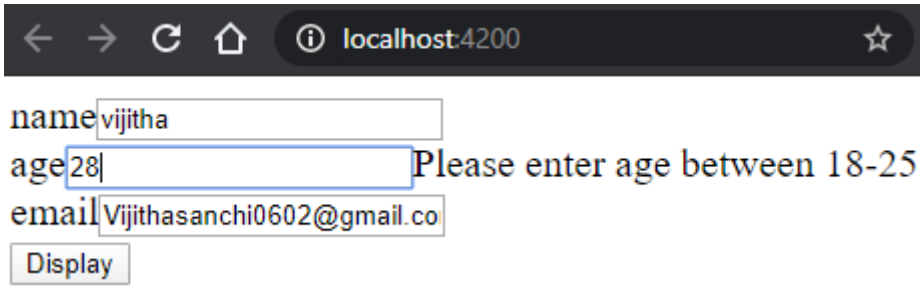
```
import { Component } from '@angular/core';
import {FormGroup, FormControl, Validators} from '@angular/forms';
import {Agerange} from './agerange';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'rfdemo';
  myform=new FormGroup(
    {
      name:new FormControl("",[Validators.required]),
      age:new FormControl("",[Agerange]),
      email:new FormControl("",[Validators.required,Validators.email]),
    }
  );
  SaveData(){
    console.log(this.myform.value);
  }
}
```

agerange.ts

```
import { AbstractControl } from '@angular/forms';
export function Agerange(control:AbstractControl) {
    if (control.value>=18 && control.value<=25) {
        return null;
    }
    else{
        return {range:true}
    }
}
```

Output:





Method example:

```
function f1()
{
  console.log("i am function");
}
 f1();
function f2(x:number,y:number)
{
 console.log(x+y);
}
f2(6,3);
```

```
class Employee
{
  eno:number;
  ename:string;
}
function f3(e1:Employee)
{
  console.log(e1.eno+e1.ename);
}
var em:Employee=new Employee();
    em.eno=101;     em.ename="anil";
  f3(em);
```

Create a project to display registration form using reactive forms and bootstraps?

FormBuilder

```
class FormControl
{
 variables
 methods
}
class FormGroup
{
 variables
 methods
}
class FormBuilder
{
 methods
 variables     group(){}
```

```
myform=new FormGroup(
{
 Name:new FormControl(),
Age:new FormControl(),
Email:new FormControl(),
});
```

```
class AppComponent
{
constructor(private fb:FormBuilder){}
  this.regform=fb.group(
  {
    Name:["",Validators.required],
    Email:["",Validators.email]
  });
}
```

1. install bootstrap

```
npm install bootstrap --save
```

2. goto angular.json

```
"styles": [
    "src/styles.css",
    "node_modules/bootstrap/dist/css/bootstrap.css"
],
```

app.module.ts

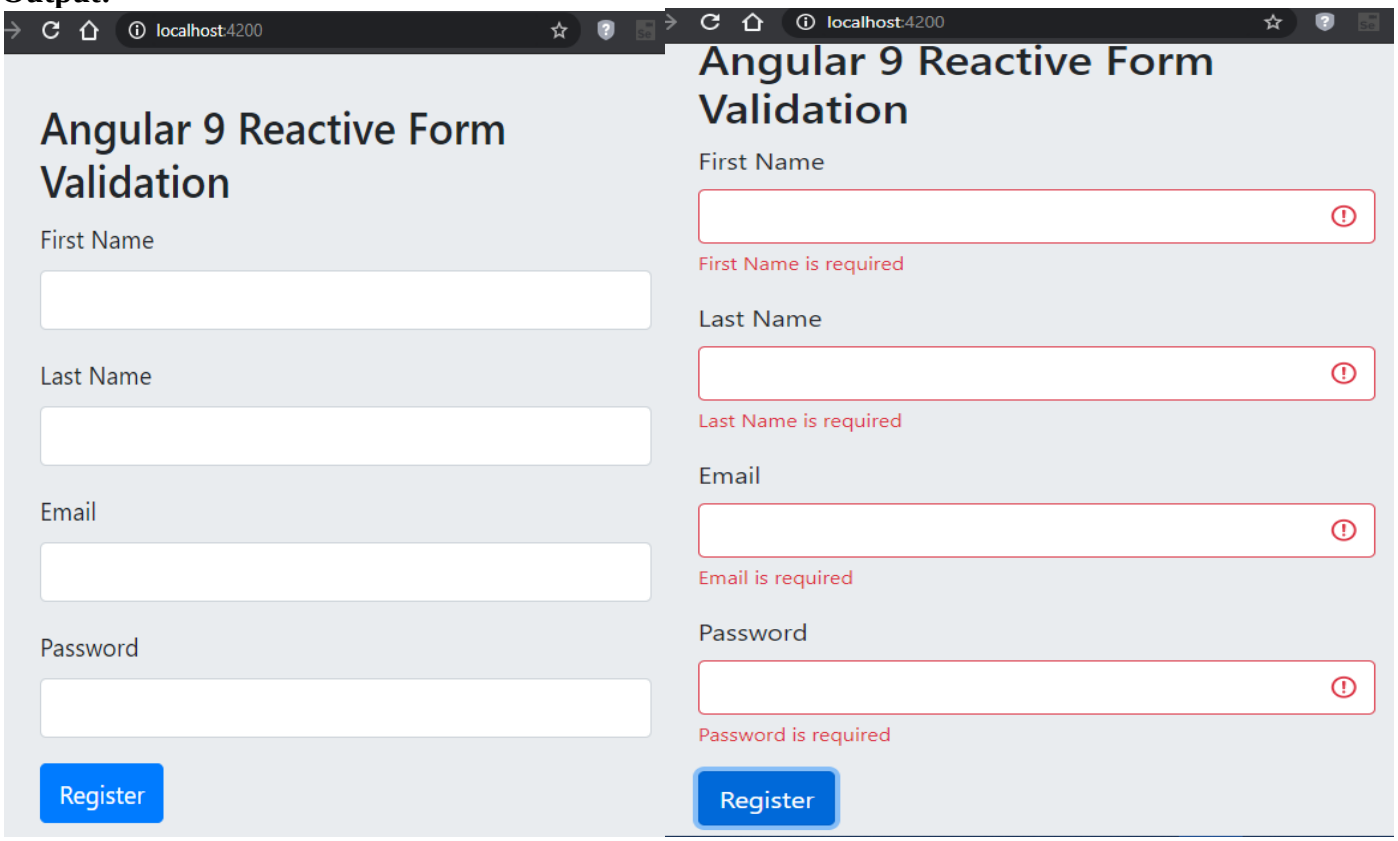
```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import {ReactiveFormsModule} from '@angular/forms';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    ReactiveFormsModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

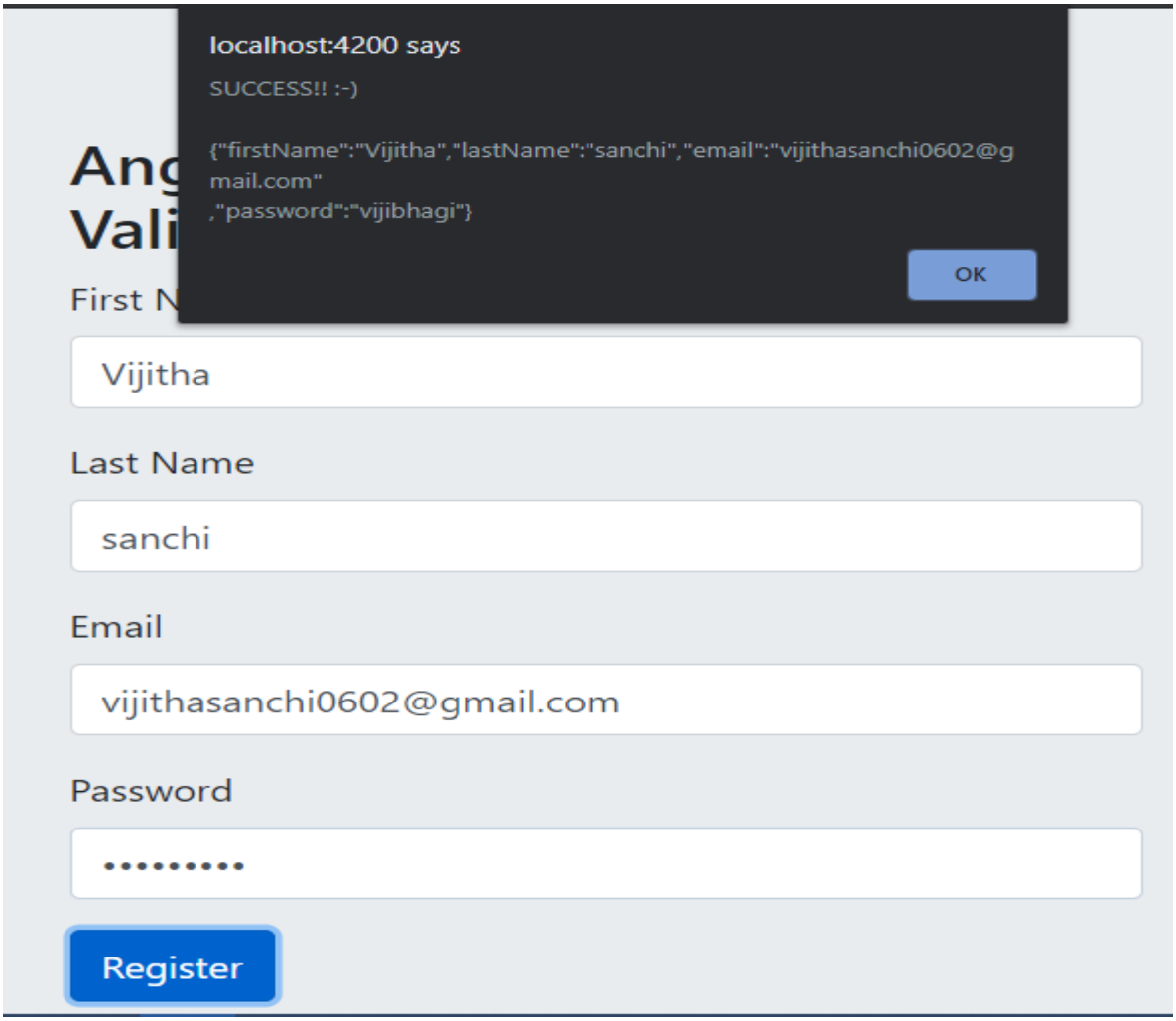
app.component.html

```
<div class="jumbotron">
  <div class="container">
     <div class="row">
        <div class="col-md-6 offset-md-3">
           <h3>Angular 9 Reactive Form Validation</h3>
           <form [formGroup]="registerForm" (ngSubmit)="onSubmit()">
              <div class="form-group">
                 <label>First Name</label>
                 <input type="text" formControlName="firstName" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.firstName.errors }" />
                 <div *ngIf="submitted && f.firstName.errors" class="invalid-feedback">
                    <div *ngIf="f.firstName.errors.required">First Name is required</div>
                 </div>
              </div>
              <div class="form-group">
                 <label>Last Name</label>
                 <input type="text" formControlName="lastName" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.lastName.errors }" />
                 <div *ngIf="submitted && f.lastName.errors" class="invalid-feedback">
                    <div *ngIf="f.lastName.errors.required">Last Name is required</div>
                 </div>
              </div>
              <div class="form-group">
                 <label>Email</label>
                 <input type="text" formControlName="email" class="form-control" [ngClass]="{ 'is-
invalid': submitted && f.email.errors }" />
```

```
                <div *ngIf="submitted && f.email.errors" class="invalid-feedback">
                    <div *ngIf="f.email.errors.required">Email is required</div>
                    <div *ngIf="f.email.errors.email">Email must be a valid email address</div>
                </div>
            </div>
            <div class="form-group">
                <label>Password</label>
                <input type="password" formControlName="password" class="form-
control" [ngClass]="{ 'is-invalid': submitted && f.password.errors }" />
                <div *ngIf="submitted && f.password.errors" class="invalid-feedback">
                    <div *ngIf="f.password.errors.required">Password is required</div>
                    <div *ngIf="f.password.errors.minlength">Password must be at least 6 characters</div>
                </div>
            </div>
            <div class="form-group">
                <button class="btn btn-primary">Register</button>
            </div>
        </form>
    </div>
  </div>
</div>
<router-outlet></router-outlet>
```

Output:





https://stackblitz.com/edit/angular-6-reactive-form-validation?file=app%2Fapp.component.ts

Singleton Objects

Design Pattern:- Design Pattern is a readymade solution for already existing problems
singleton design pattern is a design pattern that restricts the object creation outside the class
1. multiple users connect to database to perform different operations in that case for every user a seperate object is created for db connectivity because of that the no of objects will increase so instead of creating multiple objects create a single instance(object) and that object was consumned by all the users.
2. Angular supports component based architecture if we want to invoke the service method in multiple components we need to create object for the service class

```
class Service
{

}
class C1                class C2                class C3
{                {                {
Service s1=new    Service s1=new    Service s1=new Service();
Service();            Service();
}                }                }
```

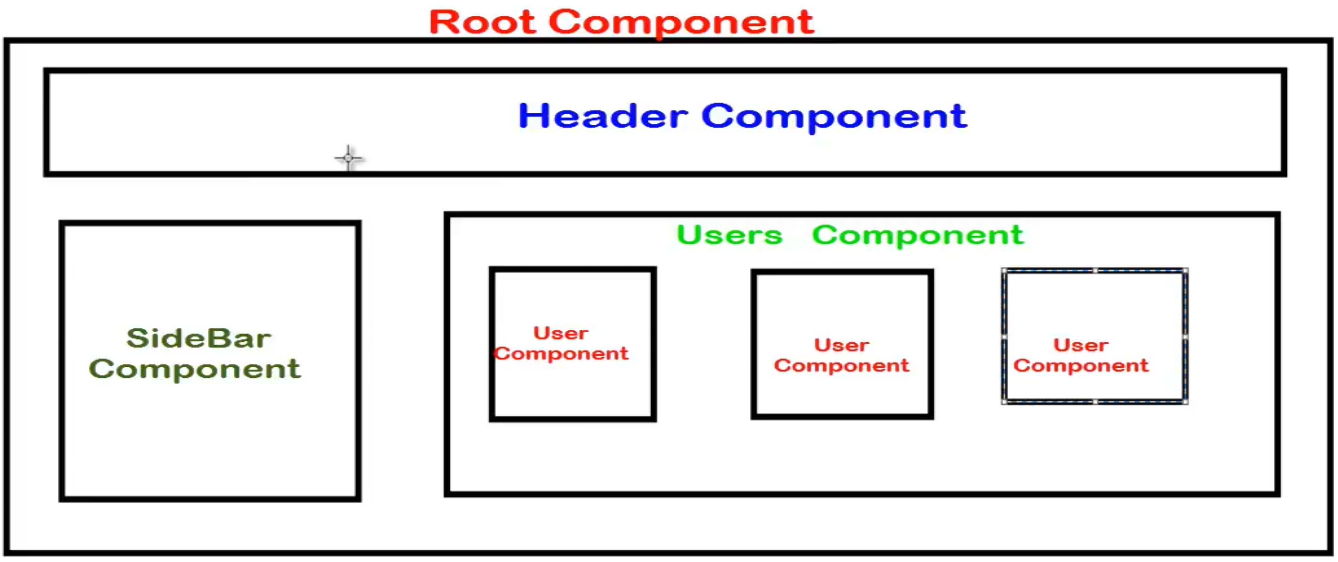
in the above example mutliple objects are created for service class so in order to control object creation internally angular services are implemented as singleton classes in angular single instance was created for service class and that instance was accessable in multiple components
==========

Singleton.ts

```
class College
{
    static c1:College;
    cname:string;
    caddress:string;
    private constructor()
    {
        this.cname="sathya";
        this.caddress="hyd";
    }
    static GetCollege():College
    {
        if(College.c1==null)
        {
            College.c1=new  College();
        }
         return College.c1;
    }
}
var o1:College=College.GetCollege();
console.log(o1.cname+o1.caddress);
var o2:College=College.GetCollege();
console.log(o2.cname+o2.caddress);
var o3:College=College.GetCollege();
console.log(o3.cname+o3.caddress);
```

Output:
sathyahyd
sathyahyd
sathyahyd
==========
Rules:-
1. create a private constructor
2. create a reference variable as static
3. create a static method
4. static method must return single instance

Parent class-Child class Communication



Create a project to display child component inside parent component?
create a component with name 'child' **ng g c child**

child.component.html

```
<div>
    <p>Child Component!</p>
</div>
```

child.component.css

```
div{
    background: red;
    padding: 30px;
}
```

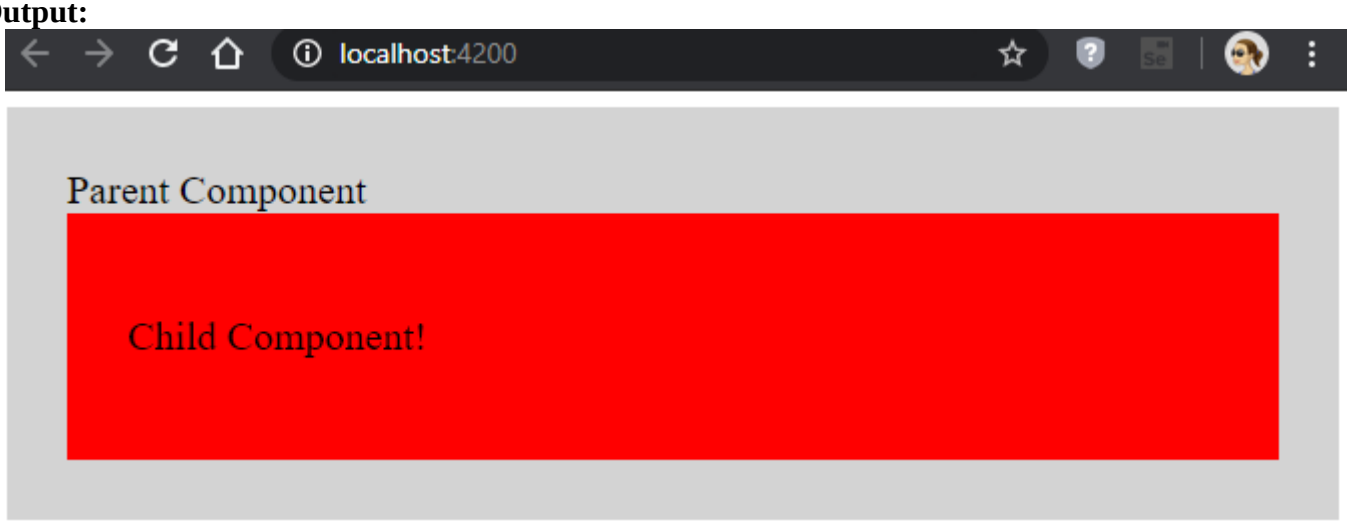
app.component.html

```
<div>
  Parent Component
  <app-child></app-child>
</div>
<router-outlet></router-outlet>
```

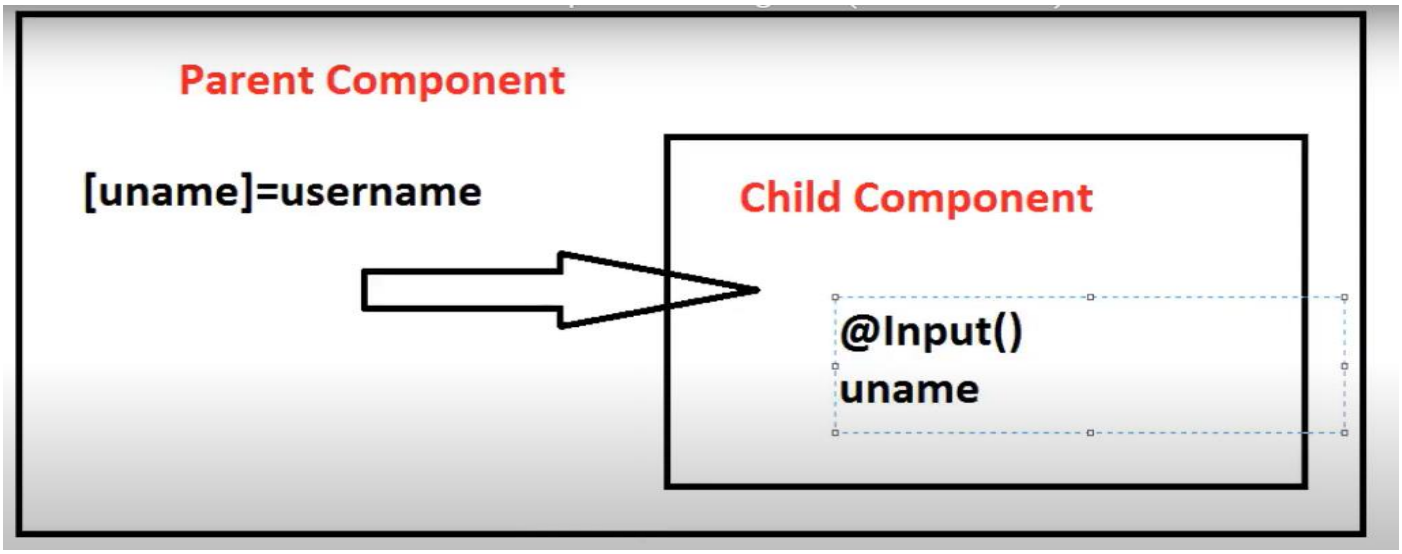
app.component.css

```
div{
    background: lightgrey;
    padding: 30px;
}
```

Output:



Create a project to show the communication between parent class to child class?
create a component with name 'child' **ng g c child**



@input:pass the data from parent to child
@output:pass the data from child to parent

app.component.html

```
<div>
  Parent Component
  <app-child [uname]="UserName"></app-child>
</div>
<router-outlet></router-outlet>
```

child.component.html

```
<div>    <p>Child Component {{uname}}</p>      </div>
```

child.component.ts

```
import { Component, OnInit, Input } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
export class ChildComponent implements OnInit {
@Input()
uname:string;
  constructor() { }
  ngOnInit(): void {  }
}
```

app.component.ts

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'parentchild';
  UserName="Bhargav";
}
```

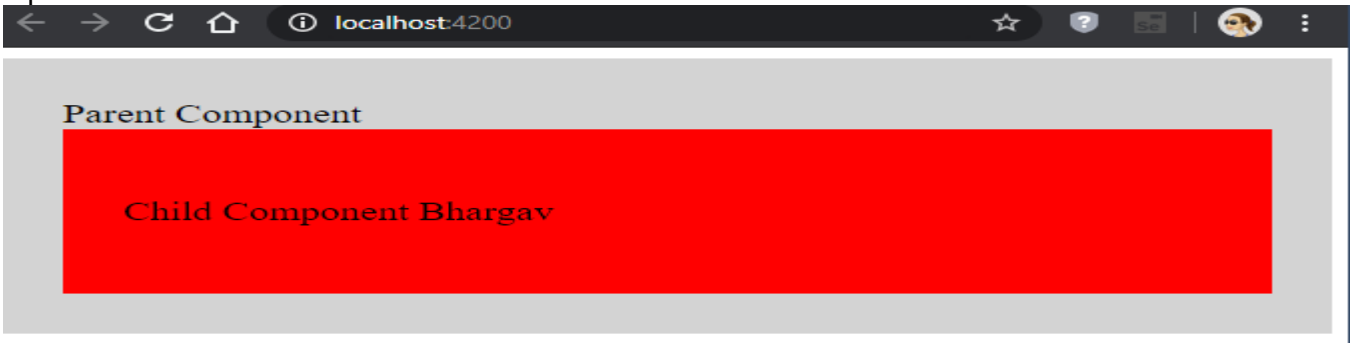
app.component.css

```
div{
    background: lightgrey;
    padding: 30px;
}
```

child.component.css

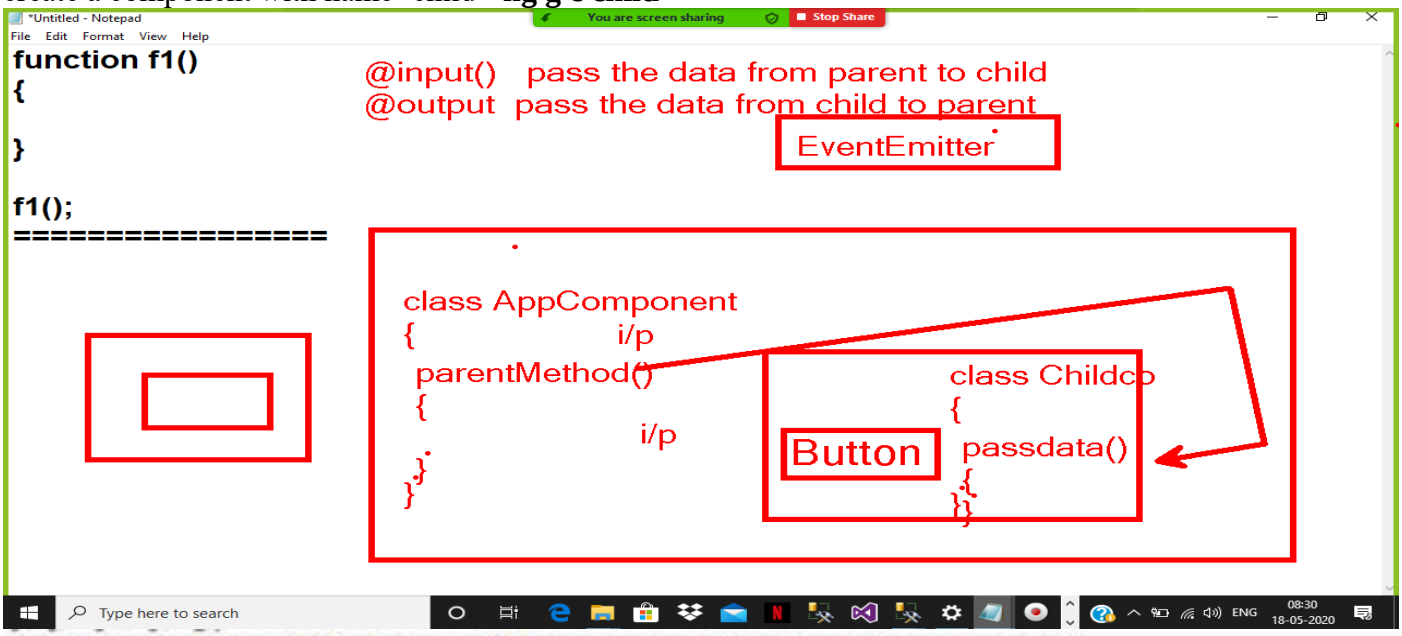
```
div{
    background: red;
    padding: 30px;
}
```

Output:



Create a project to show the communication between child class to parent class?

create a component with name 'child' **ng g c child**



How to pass the Data from child component to Parent component:-

@Output binds a property of the type of angular EventEmitter class. This property name becomes custom event name for calling component. @Output decorator can also alias the property name as @Output(alias) and now this alias name will be used in custom event binding in calling component.

childcomponent to parent communication is Event based communication

app.component.ts

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'childtoparent';
  childdata:string;
  ParentMethod(data){
    this.childdata=data;
  }
}
```

app.component.html

```
<div>
  <h1>Parent Component</h1>
  <p>{{childdata}}</p>
  <app-child (notify)="ParentMethod($event)"></app-child>
</div>
<router-outlet></router-outlet>
```

child.component.ts

```
import { Component, OnInit,Output,EventEmitter } from '@angular/core';
@Component({
  selector: 'app-child',
  templateUrl: './child.component.html',
  styleUrls: ['./child.component.css']
})
```

```
export class ChildComponent implements OnInit {
  @Output()
  notify:EventEmitter<string>=new EventEmitter<string>();
    PassData(){
      this.notify.emit("iam from child");
    }
  constructor() { }
  ngOnInit(): void {
  }

}
```

child.component.html

```
<div>
    <p>Child Component </p>
    <button type="submit" (click)="PassData()">click</button>
</div>
```

app.component.css

```
div{
    background: lightgrey;
    padding: 30px;
}
```

child.component.css

```
div{
    background: red;
    padding: 30px;
}
```

Output:

- whenever user clicks on Button
- Child Method will gets executed
- PassData(){}
- then emit() will invoke parent method and pass the result of childmethod as i/p for parent method
- and we are storing the result in variable and bind in Frontend i.e app.component.html

Depenedency Injection

It can be achieved in 3 ways:

- code without DI
- DI as Design Pattern
- DI as a Framework

Code without DI

```
class Engine{
    constructor(){}
}
class Tires{
    constructor(){}
}
```

```
class Car{
    engine;
    tires;
    constructor()
    {
        this.engine = new Engine();
        this.tires = new Tires();
    }
}
```

```
class Circle
{
   Draw()
   { draw a circle }
}
class Square
{
   Draw()
   { draw a square }
}
class Triangle
{
  Draw(){ draw a triangle }
}
```

```
class Shape
{ c1:Circle;  s1:Square;
  GetShape()
  {
    this.c1=new Circle();
    this.s1=new Square();
  }
}
```

without DI

DI as a design Patern:

DI is a coding pattern in which a class receives its dependencies from external sources rather than creating them itself.

```
class Engine{
    constructor(){}
}
class Tires{
    constructor(){}
}
```

```
class Car{
    engine;
    tires;
    constructor()
    {
        this.engine = new Engine();
        this.tires = new Tires();
    }
}
```

DI as a design pattern contd.

```
var myEngine = new Engine();
var myTires = new Tires();
var myCar = new Car(myEngine, myTires);
```

```
Class Car
{
 constructor(e1:Engine,t1:Tires)
 {

 }
}
```

```
var myEngine = new Engine(parameter);
var myTires = new Tires();
var myCar = new Car(myEngine, myTires);
```

```
var myEngine = new Engine(parameter);
var myTires = new Tires(parameter);
var myCar = new Car(myEngine, myTires);
```

```
var oldEngine = new Engine(oldparameter);
var oldTires = new Tires(oldparameter);
var oldCar = new Car(oldEngine, oldTires);
```

```
var newEngine = new Engine(newparameter);
var newTires = new Tires(newparameter);
var newCar = new Car(newEngine, newTires);
```

```
var myEngine = new Engine();
var myTires = new Tires();
var depA = new dependency();
var depB = new dependency();
var depZ = new dependency();
var myCar = new Car(myEngine, myTires, depA, depB, depZ);
```

```
var myEngine = new Engine();
var myTires = new Tires();
var depA = new dependency();
var depB = new dependency();
var depAB = new dependency();
var depZ = new dependency(depAB);
var myCar = new Car(myEngine, myTires, depA, depB, depZ);
```

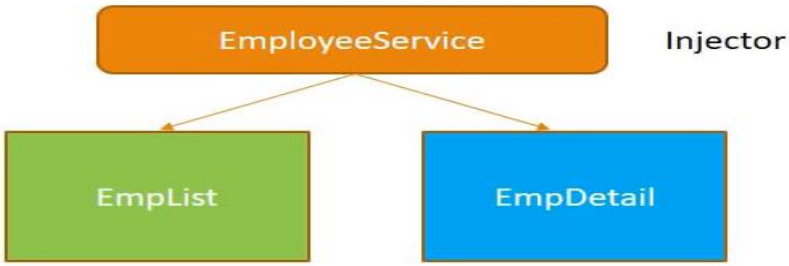
```
interface Ishape
{
  Draw();
}
class Circle implements Ishape
{
   Draw()
   { draw a circle }
}
class Square implements Ishape
{
   Draw()
   { draw a square }
}
class Triangle implements Ishape
{
  Draw(){ draw a triangle }
}
```

With DI as Design Pattern

```
class Shape
{
constructor(Ishape i)
{
  i.Draw();

 }
}
var c1:Circle=new Circle();
var s1:Square=new Square();
  var sh:Shape=new Shape(c1);
```

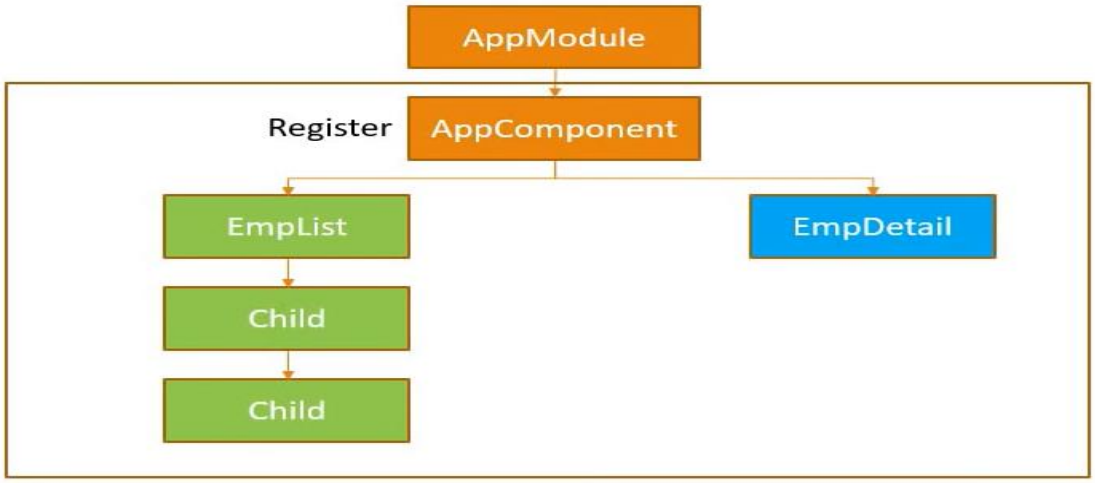
DI As a Framework

1) Define the EmployeeService class
2) Register with Injector
3) Declare as dependency in EmpList and EmpDetail




```
PS F:\Ag7am\didemo> ng g s employee
CREATE src/app/employee.service.spec.ts (343 bytes)
CREATE src/app/employee.service.ts (137 bytes)
PS F:\Ag7am\didemo> ng g c emplist
CREATE src/app/emplist/emplist.component.html (22 bytes)
CREATE src/app/emplist/emplist.component.spec.ts (635 bytes)
CREATE src/app/emplist/emplist.component.ts (273 bytes)
CREATE src/app/emplist/emplist.component.css (0 bytes)
UPDATE src/app/app.module.ts (479 bytes)
PS F:\Ag7am\didemo> ng g c empdetail
CREATE src/app/empdetail/empdetail.component.html (24 bytes)
CREATE src/app/empdetail/empdetail.component.spec.ts (649 bytes)
CREATE src/app/empdetail/empdetail.component.ts (281 bytes)
CREATE src/app/empdetail/empdetail.component.css (0 bytes)
UPDATE src/app/app.module.ts (573 bytes)
PS F:\Ag7am\didemo>
```

Hierarchical DI in Angular

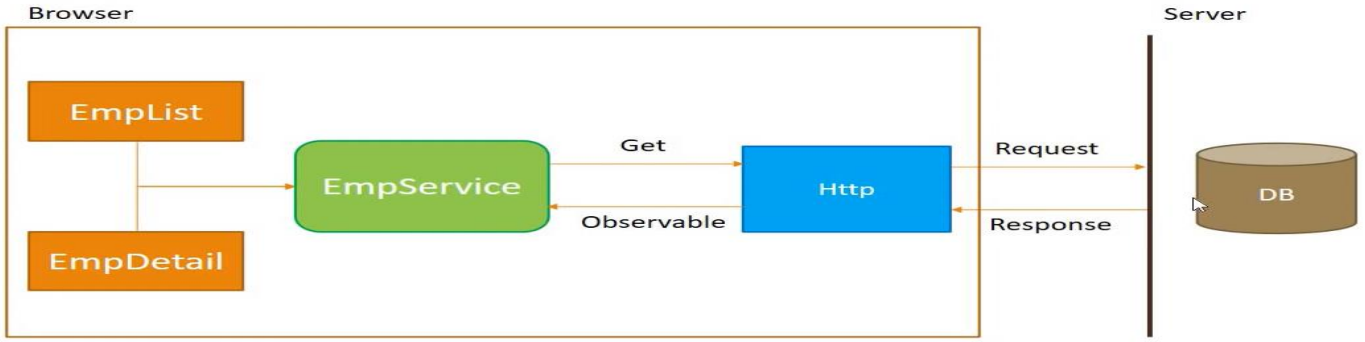


HTTP, Observables and RxJS

1. HTTP Get request from EmpService
2. Receive the observable and cast it into an employee array
3. Subscribe to the observable from EmpList and EmpDetail
4. Assign the employee array to a local variable

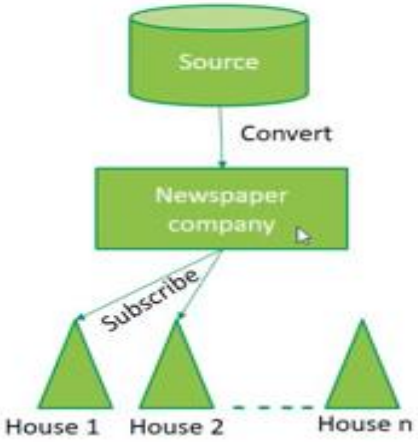
RxJS
- Reactive Extensions for Javascript
- External library to work with Observables

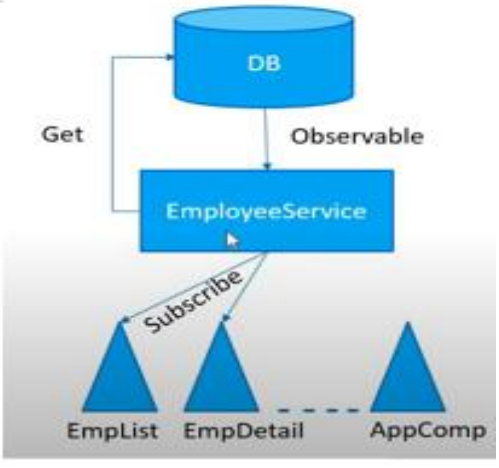
Http Mechanism



Observables

Observables





A sequence of items that arrive asynchronously over time.
HTTP call: single item
Single item: HTTP response

Create a project to demonstrate the use of observables?

1. install json server

```
npm install -g json-server
```

2. start json server

```
json-server --watch db.json
```

code for db.json

```
{
  "employees": [
        {"id":1,"name":"anil","age":30},
        {"id":2,"name":"sunil","age":25},
        {"id":3,"name":"ajay","age":23},
        {"id":4,"name":"vijay","age":24},
        {"id":5,"name":"john","age":26}
  ]
}
```

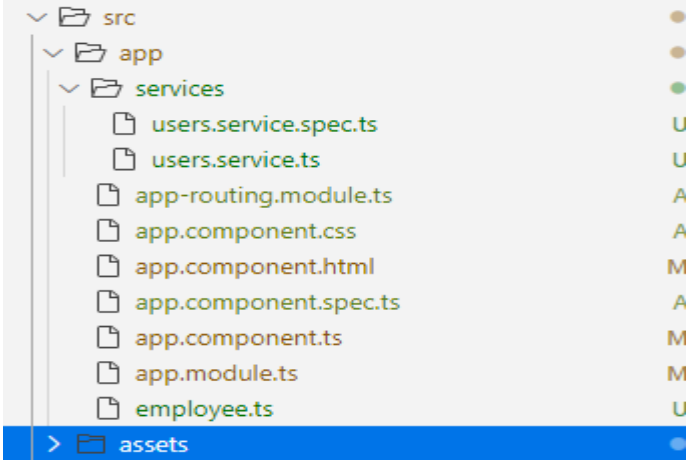
create a user service: ng g s services/users

users.serveice.ts

```
import { Injectable } from '@angular/core';
import{HttpClient} from '@angular/common/http';
import { IEmployee } from '../employee';
import { Observable } from 'rxjs';
@Injectable({
  providedIn: 'root'
})
export class UsersService {
  private _url:string="http://localhost:3000/employees";
  constructor(private http:HttpClient) { }
  GetUsers():Observable<IEmployee[]>
  {
    debugger;
 return this.http.get<IEmployee[]>(this._url);
  }
}
```

employee.ts(create this file under app folder)

```
export interface IEmployee
{
   id:number;
   name:string;
   age:number;
}
```



code for app.component.ts

```
import{Component, OnInit } from '@angular/core';
import{ UsersService } from './services/users.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
  title = 'servicesdemo';
  users:any[];
   constructor(private us:UsersService)   {   }
   ngOnInit()
   {     debugger;
    this.us.GetUsers().subscribe(
      data=>this.users=data);
   }}
```

app.component.html:-

```
<!--The content below is only a placeholder and can be replaced.-->
<div>
  <table>
     <tr>
       <th>id</th>
       <th>Name</th>
       <th>Age</th>
     </tr>
     <tr *ngFor="let user of users">
       <td>{{user.id}}</td>
       <td>{{user.name}}</td>
       <td>{{user.age}}</td>
     </tr>
   </table>
</div>
<router-outlet></router-outlet>
```

code for **app.module.ts:-**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import{UsersService} from './services/users.service';
import{HttpClient, HttpClientModule} from '@angular/common/http';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    HttpClientModule,
    AppRoutingModule
  ],
  providers: [UsersService],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

Output:

id	Name	Age
1	anil	30
2	sunil	25
3	ajay	23
4	vijay	24
5	john	26

HTTP Error Handling

Create a project to demonstrate the exception handling using Oservables?

npm install rxjs-compat

Create a project to demonstrate the Exception handling using observables?

1. install json server

```
npm install -g json-server
```

2. start json server

```
json-server --watch db.json
```

code for db.json

```
{
  "employees": [
        {"id":1,"name":"anil","age":30},
        {"id":2,"name":"sunil","age":25},
        {"id":3,"name":"ajay","age":23},
        {"id":4,"name":"vijay","age":24},
        {"id":5,"name":"john","age":26}
  ]
}
```

create a user service: ng g s services/users

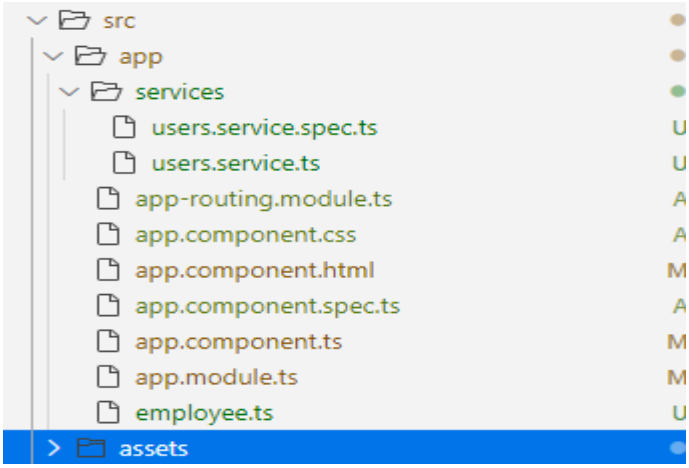
users.serveice.ts

```
import { Injectable } from '@angular/core';
import {HttpClient, HttpErrorResponse} from '@angular/common/http';
import {IEmployee} from './employee';
import {Observable,observable} from 'rxjs';
import 'rxjs/add/operator/catch';
import 'rxjs/add/observable/throw';
@Injectable({
  providedIn: 'root'
})
export class UsersService {
  private _url:string="http://localhost:3000/employees";
  constructor(private http:HttpClient) { }
  GetUsers():Observable<IEmployee[]>
  {
    debugger
    return this.http.get<IEmployee[]>(this._url).catch(this.errorHandler);
  }
```

```
  errorHandler(error:HttpErrorResponse)
  {
     return Observable.throw(error.message || "server Error");
  }
}
```

employee.ts(create this file under app folder)

```
export interface IEmployee
{
  id:number;
  name:string;
  age:number;
}
```



code for app.component.ts

```
import { Component, OnInit } from '@angular/core';
import { UsersService } from './users.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit{
  title = 'observablesexception1';
  public errorMsg;
  users:any[];
  constructor(private us:UsersService){}
  ngOnInit(){
    debugger;
    this.us.GetUsers().subscribe(
      data=>this.users=data,
      error=>this.errorMsg=error
    );
  }
}
```

app.component.html:-

```
<!--The content below is only a placeholder and can be replaced.-->
<div>
 <table>
   <tr>
    <th>id</th>
    <th>Name</th>
    <th>Age</th>
```

```
    </tr>
    <tr *ngFor="let user of users">
      <td>{{user.id}}</td>
      <td>{{user.name}}</td>
      <td>{{user.age}}</td>
    </tr>
  </table>
</div>
<router-outlet></router-outlet>
```

code for **app.module.ts:-**

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import{UsersService} from './services/users.service';
import{HttpClient, HttpClientModule} from '@angular/common/http';
@NgModule({
 declarations: [
   AppComponent
 ],
 imports: [
   BrowserModule,
   HttpClientModule,
   AppRoutingModule
 ],
 providers: [UsersService],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

Output:

id	Name	Age
1	anil	30
2	sunil	25
3	ajay	23
4	vijay	24
5	john	26

Custom Directives

1. Structural Directives(ngIf,ngFor,ngSwitch)

2. Custom Directives:-creating the directives depending on the user requirement

Ex:

If our webpage contains multiple text boxes, if it wants to display the text in “red color” or “accept only 10 numbers” we have to give same code repetitedly. Such repetited code can be placed in custom directives.

Create a project to create custom directive?

create a directive

ng g d allowtendigit

Example 1:

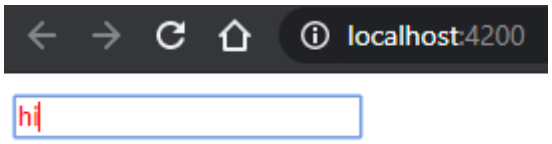
allowtendigits.directive.ts

```
import { Directive, ElementRef, OnInit } from '@angular/core';

@Directive({
  selector: '[appAllowtendigits]'
})
export class AllowtendigitsDirective implements OnInit{

  constructor(private eleref:ElementRef) { }
  ngOnInit(){
    debugger;
    this.eleref.nativeElement.style.color="red";
  }
}
```

app.component.html

```
<div>
  <input appAllowtendigits type="text">
</div>
<router-outlet></router-outlet>
```

Output:



Example 2:

in order to apply style for ElementRef
onInit :- apply style code
Events :- @HostListener
Whenever user press any key in the TextBox then keypress event will fire:-
@HostListener(Inbuilt directive) will observe the Element Events

allowtendigits.directive.ts

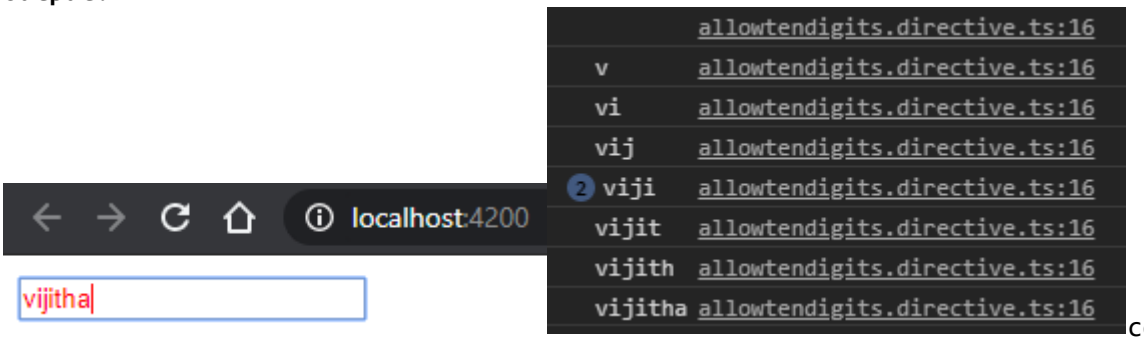
```
import { Directive, ElementRef, OnInit, HostListener } from '@angular/core';
@Directive({
  selector: '[appAllowtendigits]'
})
export class AllowtendigitsDirective implements OnInit{
  constructor(private eleref:ElementRef) { }
  ngOnInit(){
    debugger;
    this.eleref.nativeElement.style.color="red";
  }
  @HostListener('keypress')
```

```
  keypress()
  {
  console.log(this.eleref.nativeElement.value);
  }
}
```

app.componet.html

```
<div>
  <input appAllowtendigits type="text">
</div>
<router-outlet></router-outlet>
```

Output:



console

Example 3:

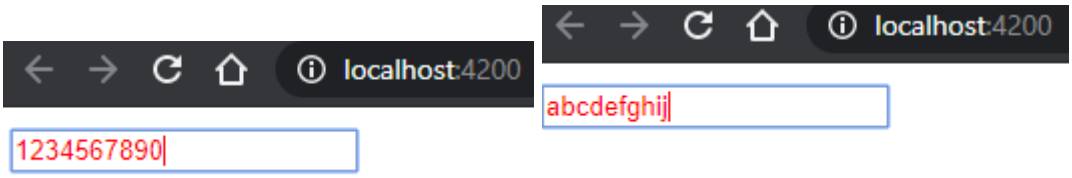
allowtendigits.directive.ts

```
import { Directive, ElementRef, OnInit, HostListener } from '@angular/core';

@Directive({
  selector: '[appAllowtendigits]'
})
export class AllowtendigitsDirective implements OnInit{

  constructor(private eleref:ElementRef) { }
  ngOnInit(){
    debugger;
    this.eleref.nativeElement.style.color="red";
  }
  @HostListener('keypress')
  keypress()
  {
    if(this.eleref.nativeElement.value.length+1>10)
    {
      return false;
    }
    else
    {
      return true;
    }
  }
}
```

app.componet.html

```
<div>
  <input appAllowtendigits type="text">
</div>
<router-outlet></router-outlet>
```

Output:



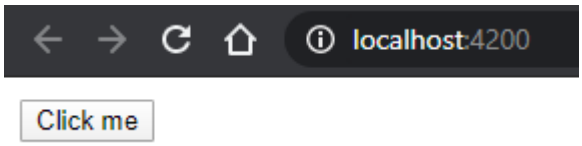
Accepts only 10 characters.

ViewChild

code for app.component.html

```
<div>
 <button type="button">Click me</button>
</div>
<router-outlet></router-outlet>
```

Output:



Example 2:

code for app.component.html

```
<div>
 <button type="button" #btn>Click me</button>
</div>
```

code for app.component.ts

```
import { Component, ViewChild, ElementRef} from '@angular/core';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'viewchilddemo1';
  @ViewChild("btn",{static:true})myvariable:ElementRef;
  ngAfterViewInit()
  {
   console.log(this.myvariable.nativeElement);
    this.myvariable.nativeElement.style.color="red";
    debugger;
  }
}
```

Click me

DOM Manipulations are mentioned in ngAfterViewInit(). DOM is used to manipulate the html elements at runtime.
ngAfterViewInit() is an angular life cycle hook method. it gets executed automatically Once DOM is ready. DOM will come to active state when all the html elements aree loaded.

Example 3:

Accesing viewchild in Childcomponent:-

Create a component: **ng g c second**

code for second.componen.ts:-

```
export class SecondComponent implements OnInit {
  constructor() { }
name:string="Hello Bhargav";
  ngOnInit() {
  }
}
```

Second.component.html:-

```
<p>{{name}}</p>
```

app.component.html:-

```
<!--The content below is only a placeholder and can be replaced.-->
<div>
  <button type="button" #btn>Click me</button>
</div>
<app-second></app-second>
```

Note:- Here AppComponent is parentComponent
<app-second> will work like child component

Output:

Click me

Hello Bhargav

Example 4:

code for app.component.html

```
<div>
  <button type="button" (click)="ChangeName()" #btn>Click me</button>
</div>
<app-second></app-second>
<router-outlet></router-outlet>
```

code for app.component.ts

```
import { Component, ViewChild, ElementRef} from '@angular/core';
import {SecondComponent} from './second/second.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'viewchilddemo1';
  @ViewChild("btn",{static:true})myvariable:ElementRef;
  @ViewChild(SecondComponent)mycomponent:SecondComponent;
  ngAfterViewInit()
  {
   console.log(this.myvariable.nativeElement);
    this.myvariable.nativeElement.style.color="red";

    debugger;
  }
  ChangeName(){
    this.mycomponent.name="Hello Vijitha";
  }
}
```

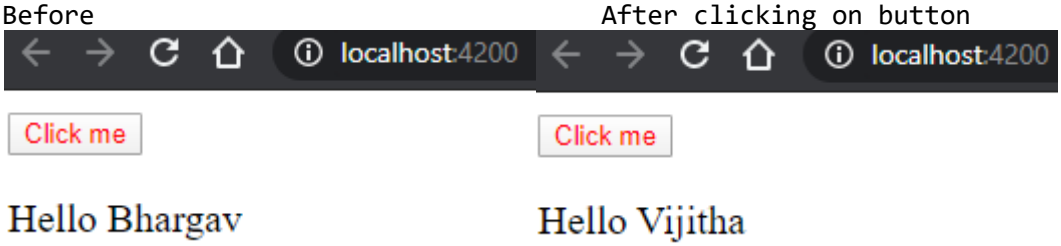
Second.component.html:-

```
<p>{{name}}</p>
```

Second.component.ts:-

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

@Component({
  selector: 'app-second',
  templateUrl: './second.component.html',
  styleUrls: ['./second.component.css']
})
export class SecondComponent implements OnInit {

  constructor() { }
  name:string="Hello Bhargav";
  ngOnInit(): void {  }
}
```

Output: Before After clicking on button



Example 5:

code for app.component.html

```
<div>
  <button type="button" (click)="ChangeName()" #btn>Click me</button>
</div>
<app-second></app-second>
<router-outlet></router-outlet>
```

code for app.component.ts

```
import { Component, ViewChild, ElementRef} from '@angular/core';
import {SecondComponent} from './second/second.component';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
```

```
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  title = 'viewchilddemo1';
  @ViewChild("btn",{static:true})myvariable:ElementRef;
  @ViewChild(SecondComponent)mycomponent:SecondComponent;
  ngAfterViewInit()
  {
   console.log(this.myvariable.nativeElement);
    this.myvariable.nativeElement.style.color="red";
    debugger;
  }
  ChangeName(){
    this.mycomponent.name="Hello Vijitha";
    this.mycomponent.Test();
  }
}
```

code for second.component.html

```
<p>{{name}}</p>
```

code for second.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-second',
  templateUrl: './second.component.html',
  styleUrls: ['./second.component.css']
})
export class SecondComponent implements OnInit {
  constructor() { }
  name:string="Hello Bhargav";
  ngOnInit(): void {
  }
Test()
{
  alert("i am from second component");
}
}
```

Output: