

**Q1. (single inheritance) implements inheritance concepts. Make a base class vehicle, make two derived classes at least (e.g car, bicycle etc) override the method of base class in derived classes.**

```
// Base class
class Vehicle {
    price = 50000;
    color = "blue";
    mileage = 0;

    drive() {
        console.log("The vehicle is driving...");
    }

    brake() {
        console.log("The vehicle is braking...");
    }
}

// Derived class 1: Motorbike
class Motorbike extends Vehicle {
    helmet = true;

    // Overriding drive() method
    drive() {
        console.log("The motorbike is zooming on the road!");
    }

    wheelie() {
        console.log("The motorbike is doing a wheelie!");
    }
}

// Derived class 2: Car
class Car extends Vehicle {
    airbags = 6;

    // Overriding drive() method
    drive() {
        console.log("The car is cruising smoothly!");
    }
}
```

```

    }

    openSunroof() {
        console.log("The sunroof is open!");
    }
}

// Creating instances
const myMotorbike = new Motorbike();
myMotorbike.drive();          // Output: The motorbike is zooming on the
road!
myMotorbike.wheelie();        // Output: The motorbike is doing a wheelie!
console.log(myMotorbike.color); // Output: blue

const myCar = new Car();
myCar.drive();                // Output: The car is cruising smoothly!
myCar.openSunroof();          // Output: The sunroof is open!
console.log(myCar.color);      // Output: blue

```

OUTPUT:

```

PS C:\Users\91993\OneDrive\Desktop\webx practise> node inheritance.js
The motorbike is zooming on the road!
The motorbike is doing a wheelie!
blue
The car is cruising smoothly!
The sunroof is open!
blue

```

// Both **Motorbike** and **Car** classes override the **drive()** method to give their **own custom message** instead of using the default one from **Vehicle**. This is called **method overriding**, where the child class provides a **specific version** of the parent's method. the **drive()** method is also **customized** (overridden) in each class to match its own behavior.

## Q2. multi-level inheritance question

```
// Base class
class Vehicle {
  price = 50000;
  color = "blue";
  mileage = 0;

  drive() {
    console.log("The vehicle is driving...");
  }

  brake() {
    console.log("The vehicle is braking...");
  }
}

// Derived class 1 (inherits from Vehicle)
class Car extends Vehicle {
  airbags = 6;

  drive() {
    console.log("The car is cruising smoothly!");
  }

  openSunroof() {
    console.log("The sunroof is open!");
  }
}

// Derived class 2 (inherits from Car) - Multi-Level Inheritance
class SportsCar extends Car {
  turbo = true;

  drive() {
    console.log("The sports car is speeding with turbo boost!");
  }

  activateTurbo() {
    console.log("Turbo mode activated!");
  }
}
```

```

    }
}

// Creating an instance
const mySportsCar = new SportsCar();
mySportsCar.drive();           // Output: The sports car is speeding with
turbo boost!
mySportsCar.openSunroof();     // Output: The sunroof is open!
mySportsCar.brake();           // Output: The vehicle is braking...
console.log(mySportsCar.color); // Output: blue

```

### ✓ Explanation (easy 2-line way for viva):

- Here, **SportsCar** inherits from **Car**, and **Car** inherits from **Vehicle**. So this is called **multi-level inheritance**.
- **SportsCar** gets features of **both Car** and **Vehicle**, like **brake()**, **openSunroof()**, and also adds its own turbo feature.

```

● PS C:\Users\91993\OneDrive\Desktop\webx practise> tsc multilevel.ts
● PS C:\Users\91993\OneDrive\Desktop\webx practise> node multilevel.js
The sports car is speeding with turbo boost!
The sunroof is open!
The vehicle is braking...
blue

```

**Q3. Create a new interface Shape. With getArea method. Create 3 class Rectangle, Circle, Triangle and implement the Shape**

(without constructor)

Here, all 3 classes (**Rectangle**, **Circle**, **Triangle**) implement the **Shape interface** and provide their **own version** of **getArea()** method.

```

● PS C:\Users\91993\OneDrive\Desktop\webx practise> tsc interface.ts
● PS C:\Users\91993\OneDrive\Desktop\webx practise> node interface.js
Rectangle Area: 50
Circle Area: 153.86
Triangle Area: 16

```

```
2 interface Shape {
3     getArea(): number;
4 }
5
6 // Rectangle class implementing Shape
7 class Rectangle implements Shape {
8     width: number = 10;
9     height: number = 5;
10
11     getArea(): number {
12         return this.width * this.height;
13     }
14 }
15
16 // Circle class implementing Shape
17 class Circle implements Shape {
18     radius: number = 7;
19
20     getArea(): number {
21         return 3.14 * this.radius * this.radius;
22     }
23 }
24
25 // Triangle class implementing Shape
26 class Triangle implements Shape {
27     base: number = 8;
28     height: number = 4;
29
30     getArea(): number {
31         return 0.5 * this.base * this.height;
32     }
33 }
34
35 // Creating objects and calling getArea
36 const rect = new Rectangle();
37 console.log("Rectangle Area:", rect.getArea());
38
39 const circle = new Circle();
40 console.log("Circle Area:", circle.getArea());
41
42 const triangle = new Triangle();
43 console.log("Triangle Area:", triangle.getArea());
```

(with constructor)

```
1  interface Shape {
2      getArea(): number;
3  }
4
5  // Rectangle class implementing Shape
6  class Rectangle implements Shape {
7      width: number;
8      height: number;
9
10     constructor(width: number, height: number) {
11         this.width = width;
12         this.height = height;
13     }
14
15     getArea(): number {
16         return this.width * this.height;
17     }
18 }
19 // Circle class implementing Shape
20 class Circle implements Shape {
21     radius: number;
22
23     constructor(radius: number) {
24         this.radius = radius;
25     }
26
27     getArea(): number {
28         return 3.14 * this.radius * this.radius;
29     }
30 }
31 // Triangle class implementing Shape
32 class Triangle implements Shape {
33     base: number;
34     height: number;
35
36     constructor(base: number, height: number) {
37         this.base = base;
38         this.height = height;
39     }
40
41     getArea(): number {
42         return 0.5 * this.base * this.height;
43     }
44 }
```

```

43     }
44 }
45
46 // Creating objects and calling getArea
47 const rect = new Rectangle(10, 5);
48 console.log("Rectangle Area:", rect.getArea()); // 50
49
50 const circle = new Circle(7);
51 console.log("Circle Area:", circle.getArea()); // 153.86
52
53 const triangle = new Triangle(8, 4);
54 console.log("Triangle Area:", triangle.getArea()); // 16
55

```

#### Q4.Implementation of library system using typescript

Create module of book transaction users, Use import and export statement , Error handling & Create modules like book owner and price, use import and export Modules in ts(common question).

myExport1.ts:

```

export interface myExport1 {
    title: string;
    author: string;
    owner: string;
    price: number;
}

export const book1: myExport1 = {
    title: "The Great Gatsby",
    author: "F. Scott Fitzgerald",
    owner: "Library A",
    price: 300
};

```

myExport2.ts:

```

export interface myExport2 {
    name: string;
    borrowedBooks: string[];
}

export const user1: myExport2 = {
    name: "Mihir",

```

```
    borrowedBooks: []  
};
```

myExport3.ts:

```
import { myExport1 } from "../myExport1";  
import { myExport2 } from "../myExport2";  
  
export function borrowBook(book: myExport1, user: myExport2): string {  
    try {  
        if (user.borrowedBooks.includes(book.title)) {  
            throw new Error("Book already borrowed!");  
        }  
        user.borrowedBooks.push(book.title);  
        return `${user.name} borrowed "${book.title}" for ₹${book.price}`;  
    } catch (error) {  
        return `Error: ${(error as Error).message}`;  
    }  
}
```

index.ts:

```
import { book1 } from "../myExport1";  
import { user1 } from "../myExport2";  
import { borrowBook } from "../myExport3";  
const result = borrowBook(book1, user1);  
console.log(result);
```

**Q5.create Class called Bank account using ts and method deposit and withdrawal, personal details and cannot withdraw if balance is less than 100 in typescript**

```
class BankAccount {  
    private owner: string;  
    private accNo: number;  
    private balance: number;  
  
    constructor(owner: string, accNo: number, balance: number = 0) {  
        this.owner = owner;  
        this.accNo = accNo;  
        this.balance = balance;  
    }  
}
```



```
// Deposit money
public deposit(amount: number): void {
    if (amount > 0) {
        this.balance += amount;
        console.log(`₹${amount} added. New balance: ₹${this.balance}`);
    } else {
        console.log("Enter valid amount to deposit.");
    }
}

// Withdraw money
public withdraw(amount: number): void {
    if (amount <= 0) {
        console.log("Enter valid amount to withdraw.");
    } else if (this.balance - amount < 100) {
        console.log("Cannot withdraw. Must keep at least ₹100 in account.");
    } else {
        this.balance -= amount;
        console.log(`₹${amount} withdrawn. Balance now: ₹${this.balance}`);
    }
}

// Show account details
public showDetails(): void {
    console.log(`Name: ${this.owner}`);
    console.log(`Account No: ${this.accNo}`);
    console.log(`Balance: ₹${this.balance}`);
}

// Example use
const myAccount = new BankAccount("Mihir", 1112223333, 500);
myAccount.showDetails();
myAccount.deposit(1000);
myAccount.withdraw(1400); // Will fail (balance would go below 100)
myAccount.withdraw(300); // Will succeed
myAccount.showDetails();
```

```
PS C:\Users\91993\OneDrive\Desktop\webx practise> tsc bank.ts
PS C:\Users\91993\OneDrive\Desktop\webx practise> node bank.js
Name: Mihir
Account No: 111222333
Balance: ₹500
₹1000 added. New balance: ₹1500
₹1400 withdrawn. Balance now: ₹100
Cannot withdraw. Must keep at least ₹100 in account.
Name: Mihir
Account No: 111222333
Balance: ₹100
```

### ✓ Method 1: deposit()

typescript

```
public deposit(amount: number): void {
```

- `public`: This means the method can be called from **outside** the class (normal).
- `amount: number`: It expects a number as input (TypeScript typing).
- `void`: This means the function **returns nothing** (just does console logs or updates).

typescript

```
if (amount > 0) {
  this.balance += amount;
  console.log(`₹${amount} added. New balance: ₹${this.balance}`);
}
```

✓ If you enter a valid amount (greater than 0):

- It **adds** that amount to your balance.
- Prints the new balance.



```
else {  
  console.log("Enter valid amount to deposit.");  
}
```

✗ If amount is 0 or negative, it shows an error message.

## ✓ Method 2: `withdraw()`

typescript

```
public withdraw(amount: number): void {
```

Same as above — public, expects a number, returns nothing.

typescript

```
if (amount <= 0) {  
  console.log("Enter valid amount to withdraw.");  
}
```

✗ If user enters zero or negative, it says invalid amount.

```
else if (this.balance - amount < 100) {  
  console.log("Cannot withdraw. Must keep at least ₹100 in account.");  
}
```

✗ This checks the **bank rule**:

- After withdrawing, your balance must be  $\geq$  ₹100.
- If not, it shows error and doesn't allow withdrawal.

typescript

```
else {  
  this.balance -= amount;  
  console.log(`₹${amount} withdrawn. Balance now: ₹${this.balance}`);  
}
```

✓ If everything is okay:

- It subtracts ( `-=` ) the amount from balance.
- Shows success message with new balance.

Q6. Build a utility App in TypeScript to manage employee salary calculations for a company. The company has a policy of giving performance-based bonuses. Your task is to implement these calculations using arrow functions wherever applicable. Define an array of employee objects. Each object should have: name (string), baseSalary (number), performanceRating (5 or 10). Based on performance rating, make a function to compute the final salary by applying the bonus to the base salary.

```
// Step 1: Define Employee Type
type Employee = {
  name: string;
  salary: number;
  rating: 5 | 10;
};

// Step 2: List of Employees
const employees: Employee[] = [
  { name: "Mihir", salary: 30000, rating: 10 },
  { name: "Asha", salary: 28000, rating: 5 },
  { name: "Ravi", salary: 32000, rating: 10 }
];

// Step 3: Function to calculate final salary
const getFinalSalary = (emp: Employee): number => {
  let bonus = emp.rating === 10 ? 0.2 : 0.1; // 20% if rating 10, else 10%
  return emp.salary + (emp.salary * bonus);
};

// Step 4: Print salaries
employees.forEach(emp => {
  console.log(`${emp.name}'s final salary is ₹${getFinalSalary(emp)}`);
});
```

```
● PS C:\Users\91993\OneDrive\Desktop\webx practise> tsc bank.ts
● PS C:\Users\91993\OneDrive\Desktop\webx practise> node bank.js
Mihir's final salary is ₹36000
Asha's final salary is ₹30800
Ravi's final salary is ₹38400
```

## ✓ Line by line meaning:

Line 1:

typescript

```
const getFinalSalary = (emp: Employee): number => {
```


- Creates a function named `getFinalSalary`
- Takes `emp` → which is an **employee**
- It will return a **number** (the final salary)

Line 2:

typescript

```
let bonus = emp.rating === 10 ? 0.2 : 0.1;
```

- If the employee's **rating** is **10** → give **20% bonus** (`0.2`)
- Else (if rating is 5) → give **10% bonus** (`0.1`)

 Example:


- rating = 10 → bonus = 0.2 (means 20%)
- rating = 5 → bonus = 0.1 (means 10%)

Line 3:

```
typescript

return emp.salary + (emp.salary * bonus);
```

- Final salary = base salary + (base salary × bonus)

 Example:

- salary = ₹30000, bonus = 0.2 →

$$30000 + (30000 \times 0.2) = 30000 + 6000 = ₹36000$$

✓ **Printing part:**

```
typescript

employees.forEach(emp => {
  console.log(`${emp.name}'s final salary is ₹${getFinalSalary(emp)}`);
});
```

👉 **Meaning:**

- `employees.forEach()` → Loops through every employee
- `emp` is the current employee in the loop
- It **prints** that employee's name and their **final salary** (using `getFinalSalary(emp)`)

## Q7 . CALCULATOR in ts

Save file as calci.html

```
<!DOCTYPE html>

<html>
<head>
  <title>Simple Calculator</title>
</head>
<body>

  <h2>Simple Calculator</h2>

  <input type="number" id="num1" placeholder="Enter number 1">
  <input type="number" id="num2" placeholder="Enter number 2">

  <br><br>

  <button onclick="calculate('add') ">Add</button>
```

```
<button onclick="calculate('sub')">Subtract</button>
<button onclick="calculate('mul')">Multiply</button>
<button onclick="calculate('div')">Divide</button>

<p id="result">Result: </p>

<script>
  function calculate(operation) {
    const n1 = parseFloat(document.getElementById("num1").value);
    const n2 = parseFloat(document.getElementById("num2").value);
    let res = 0;

    if (operation === "add") res = n1 + n2;
    else if (operation === "sub") res = n1 - n2;
    else if (operation === "mul") res = n1 * n2;
    else if (operation === "div") res = n2 !== 0 ? n1 / n2 : NaN;

    let message = "Result: " + res;
    if (isNaN(res)) {
      message = "Cannot divide by zero!";
    }

    document.getElementById("result").textContent = message;
  }
</script>

</body>
</html>
```

---

## Simple Calculator

Result: 5

## Dom manipulation explanation-

✓ 1) `document.getElementById("result").textContent = message;`

Let's split this up:

- `document` : This means the whole web page.
- `getElementById("result")` :
  - It looks for the HTML element with `id="result"`.
  - In your code, it's this:

```
html

<p id="result">Result: </p>
```

- `.textContent = message` :
  - This changes the text inside that `<p>` tag.
  - So, if `message` is `"Result: 10"`, it will change your HTML like this:

```
html

<p id="result">Result: 10</p>
```

👉 So this line basically says:

"Find the element with id 'result' and change its text to show the answer."

✓ 2) `parseFloat(document.getElementById("num2").value)`

Let's split this too:

- `document.getElementById("num2")` :
    - Finds the input box with `id="num2"`
    - In your code:
- ```
html                                     Copy

<input type="number" id="num2" placeholder="Enter number 2">
```
- `.value` :
    - Gets what the user typed in that box.
    - Example: if you type `5`, `.value` will be `"5"` (a string, like text).
  - `parseFloat(...)` :
    - This converts the text `"5"` into the real number `5`.
    - So you can do math with it (like addition, division, etc.).

👉 So this line basically says:

"Get the number typed in the second box and convert it to a number I can calculate with."

## Q8.typescript with html-temperature converter f to Celsius Temp.html:



```

<!DOCTYPE html>
<html>
<head>
  <meta charset="UTF-8">
  <title>Temp Converter</title>
</head>
<body>
  <h1>F to C Converter</h1>

  <input id="f" placeholder="Enter °F">
  <button onclick="convert()">Convert</button>

  <p id="r"></p>

  <script>
    function convert() {
      let f = document.getElementById("f").value;
      let c = (f - 32) * 5 / 9;
      document.getElementById("r").textContent = c + "°C";
    }
  </script>
</body>
</html>

```

\*TS file ko HTML mai hi add kiya hai inside script  
 For f to c the formula is-  $\text{let } f = (c * 9 / 5) + 32;$

# F to C Converter



48.888888888888886°C

## Q9. Student Registration Form:

Studentform.html

```
<!DOCTYPE html>
<html>
<head>
  <title>Student Registration</title>
</head>
<body>
  <h1>Student Registration</h1>

  Name: <input id="name"><br><br>
  Age: <input id="age"><br><br>
  Course: <input id="course"><br><br>

  <button onclick="register()">Register</button>

  <p id="msg"></p>

  <script src="studentform.js"></script> <!-- Link to JS -->
</body>
</html>
```

[studentform.ts](#)

```
function register() {
  let name = (document.getElementById("name") as any).value;
  let age = (document.getElementById("age") as any).value;
  let course = (document.getElementById("course") as any).value;

  let msg = document.getElementById("msg");
  if (msg) {
    msg.textContent = "Registered: " + name + ", Age: " + age + ", Course: " + course;
  }
}
```

# Student Registration

Name:

Age:

Course:

Registered: neha, Age: 20, Course: IT

## AngularJS:

Q.Angular js mei web app create karna tha

Koi bhi AngularJS ka code jo neeche hai  
daal do -eg. Shopping Cart

Q.create a task based SPA(task manager ) by using angular directives and services

```
<!DOCTYPE html>

<html lang="en" ng-app = "myApp">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Document</title>
  <script
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.3/angular.min.js"
></script>
</head>
<body ng-controller = 'myCtrl'>
  <h1>Task Manager:</h1>
  <input type="text" ng-model="newTask" placeholder="Enter your Task"/>
  <button ng-click ='addTask()'>Add Task</button>
  <ul>
    <li ng-repeat="t in tasks track by $index">
      {{ t }} <button ng-click="removeTask($index)">Remove</button>
    </li>
  </ul>
  <p>{{ msg }}</p>
  <script>
angular.module('myApp', [])
.controller('myCtrl', function($scope) {
  $scope.tasks = [];
  $scope.newTask = '';
  $scope.msg = '';

  $scope.addTask = function() {
    if ($scope.newTask) {
      $scope.tasks.push($scope.newTask);
    }
  }
})

```

```

        $scope.msg = 'Task Added!';
        $scope.newTask = '';
    }
};

$scope.removeTask = function(index) {
    $scope.tasks.splice(index, 1);
    $scope.msg = 'Task Removed!';
};
});
</script>
</body>
</html>

```

## Task Manager:

- new

Task Added!

Q.create single page application of online shopping cart using angular js where the cart value should update when user clicks on the product

```

<!DOCTYPE html>
<html lang="en" ng-app = 'myApp'>
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Document</title>
    <script src =
"https://ajax.googleapis.com/ajax/libs/angularjs/1.8.3/angular.min.js"></s
cript>
</head>
<body ng-controller = "myCtrl">
    <h1>CART PLACE</h1>

```

```
<p><strong>Total Items:</strong> {{ cart.length }} | <strong>Total  
Price in INR:</strong> {{ total }}</p>  
<ul>  
  <li ng-repeat="item in products">  
    {{ item.name }} - {{ item.price }}  
    <button ng-click="addItem(item)">Add To Cart</button>  
  </li>  
</ul>  
<h2>Cart:</h2>  
<ul>  
  <li ng-repeat="c in cart">  
    {{ c.name }} - {{ c.price }}  
  </li>  
</ul>  
<script>  
  angular.module('myApp', [])  
    .controller('myCtrl', function($scope){  
      $scope.products = [  
        {name: 'Apple', price: 100},  
        {name: 'Cherry', price: 50},  
        {name: 'Banana', price: 20},  
        {name: 'Grapes', price: 30},  
        {name: 'Mango', price: 40}  
      ]  
      $scope.cart = []  
      $scope.total = 0  
      $scope.addItem = function(item){  
        $scope.cart.push(item)  
        $scope.total += item.price  
      }  
    })  
</script>  
</body>  
</html>
```

## CART PLACE

Total Items: 2 | Total Price in INR: 120

- Apple - 100
- Cherry - 50
- Banana - 20
- Grapes - 30
- Mango - 40

### Cart:

- Apple - 100
- Banana - 20

2Q.angular js filters-showing product + angular js filters- Search box implement filters.html

```
<!DOCTYPE html>

<html ng-app="app">

<head>

<script

src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"

></script>

</head>

<body ng-controller="ctrl">

<input type="text" ng-model="search" placeholder="Search product...">

<ul>

  <li ng-repeat="p in products | filter:customFilter | orderBy:'price' |

limitTo:3">

    {{ p.name }} - ₹{{ p.price }}

  </li>

</ul>

<script>

angular.module('app', [])

.controller('ctrl', function($scope) {

  $scope.products = [

    { name: "Laptop", price: 45000 },

    { name: "Mouse", price: 500 },

    { name: "Keyboard", price: 800 },

    { name: "Monitor", price: 7000 },

    { name: "Printer", price: 6000 }
```

```
];  
$scope.customFilter = function(p) {  
    if (p.price <= 1000) return false;  
    if (!$scope.search) return true;  
    return p.name.toLowerCase().includes($scope.search.toLowerCase());  
};  
});  
</script>  
</body>  
</html>
```

- Printer - ₹6000
- Monitor - ₹7000
- Laptop - ₹45000



```
<html ng-app="app">
```

- Starts an AngularJS app called "app".

```
html
```

```
<body ng-controller="ctrl">
```

- Uses the controller named "ctrl" to handle all data and logic for this page.

## ✓ Input Search Box

```
html
```

```
<input type="text" ng-model="search" placeholder="Search product...">
```

- `ng-model="search"` creates a **live two-way binding** with `$scope.search`.
- As the user types in this box, the `search` variable updates in real time.

## ✓ Product List with Filters

```
html
```

[Copy](#)

```
<ul>
  <li ng-repeat="p in products | filter:customFilter | orderBy:'price' | limitTo:3">
    {{ p.name }} - ₹{{ p.price }}
  </li>
</ul>
```

Let's understand this pipeline:

### ng-repeat

```
html
```

[Copy](#)

```
ng-repeat="p in products"
```

- Loops through each **product** object from `$scope.products`.

## Filter

html

```
| filter:customFilter
```

- Uses a **custom filter function** called `customFilter`.
- Filters out products based on search text and price condition.
  - Must match the search term (case-insensitive)
  - And price must be greater than 1000 (`p.price > 1000`).

## OrderBy

html

```
| orderBy:'price'
```

- Sorts the filtered products in **ascending order** by price (₹ lowest to highest).

## LimitTo

html

```
| limitTo:3
```

- Shows **only the first 3** products after filtering and sorting.

## Controller Logic

javascript

```
angular.module('app', []).controller('ctrl', function($scope) {
```

- Defines **Angular module** and **controller**.

```
$scope.customFilter = function(p) {
  if (p.price <= 1000) return false;
  if (!$scope.search) return true;
  return p.name.toLowerCase().includes($scope.search.toLowerCase());
};
```

#### ✓ Easy to read:

- Price check first → hide if ₹1000 or less
- If search is empty → show
- Else → check if product name matches search

**Q.registration form and 6 angular services**

**Q.Any 4 angular js services**

**Q. Create a registration form using AngularJS with the following fields Name , Email , Password**

**Use ng-model to bind all input fields.**

**Use ng-submit to handle form submission.**

**The Password must be exactly 6 digits using ng-pattern.**

or

**Form Validation with AngularJS**

**Design a registration form using AngularJS built-in directives that includes real-time validation. Input fields for Name, Email, and Password using ng-model. Use ng-required, ng-minlength, and ng-pattern for validation. Display error messages conditionally using ng-show or ng-if. Disable the submit button using ng-disabled if the form is invalid.\*\*\***

registration.html

```
<!DOCTYPE html>
```

```
<html ng-app="myApp">
```

```
<head>
```

```
  <title>Registration Form (No Routing)</title>
```

```
  <script
```

```
src="https://ajax.googleapis.com/ajax/libs/angularjs/1.8.2/angular.min.js"
```

```
></script>
```

```
  <style>
```

```
    .error { color: red; font-size: 12px; }
```

```
  </style>
```

```
</head>
```

```
<body ng-controller="FormCtrl">
```

```

<h2>Register</h2>

<form name="regForm" ng-submit="submitForm()" novalidate>

  <p>
    Name:
    <input type="text" name="name" ng-model="user.name"
ng-required="true">
    <span class="error" ng-show="regForm.name.$touched &&
regForm.name.$invalid">Required</span>
  </p>

  <p>
    Email:
    <input type="email" name="email" ng-model="user.email"
ng-required="true">
    <span class="error" ng-show="regForm.email.$touched &&
regForm.email.$invalid">Valid Email Required</span>
  </p>

  <p>
    Password:
    <input type="password" name="password" ng-model="user.password"
ng-required="true" ng-pattern="/^\d{6}$/">
    <span class="error" ng-show="regForm.password.$touched &&
regForm.password.$error.required">Required</span>
    <span class="error" ng-show="regForm.password.$touched &&
regForm.password.$error.pattern">6 digits only</span>
  </p>

  <button type="submit" ng-disabled="regForm.$invalid">Submit</button>

</form>

<!-- Show submitted data -->
<div ng-if="submitted">
  <h3>Submitted Data:</h3>
  <p>Name: {{user.name}}</p>
  <p>Email: {{user.email}}</p>
  <p>Password: {{user.password}}</p>

```

```
</div>

<script>
  var app = angular.module('myApp', []);

  app.controller('FormCtrl', function($scope, $timeout, $log, $window,
$filter, $http) {
    $scope.user = {};
    $scope.submitted = false;

    $scope.submitForm = function() {
      if ($scope.regForm.$valid) {

        // 1) Log data
        $log.info("Form Data", $scope.user);

        // 2) Format Name uppercase
        $scope.user.name = $filter('uppercase')($scope.user.name);

        // 3) Show alert
        $window.alert("Registration Successful!");

        // 4) Use timeout to simulate delay
        $timeout(function() {
          $scope.submitted = true;
        }, 500);

        // 5) Example use of $http (fake call)
        $http.get('https://jsonplaceholder.typicode.com/posts/1')
          .then(function(response) {
            $log.info("HTTP success", response.data);
          }, function(error) {
            $log.error("HTTP error", error);
          });
      }
    };
  });
</script>

</body>
```

</html>

## Register

Name:

Email:

Password:

### Submitted Data:

Name: ASD  
Email: asd@hjk  
Password: 123456

← → ↺  C:/Users/91993/OneDrive/Desktop/webx%20practise1/registration.html

## Register

Name:

Email:

Password:

This page says  
Registration Successful!

## Register

Name:

Email:

Password:  6 digits only

## Register

Name:

Email:  Valid Email Required

Password:

**\*\*Agar koi ek code karke jana hai toh  
.angular js filters-showing product  
Toh ye wala karke jao**

---

- 1. Web Analytics:** It is the measurement, collection, analysis, and reporting of web data to understand and optimize website usage.
- 2. Semantic Web:** It is an extension of the current web where data is structured and linked for better machine understanding.
- 3. Tools for Web Analytics:** Google Analytics, Matomo, Adobe Analytics, Hotjar.
- 4. Web Analytics:** Tracking user interaction & behavior on websites.
- 5. Web App:** An application accessed via web browser over the internet.
- 6. RIA (Rich Internet Application):** Web apps with desktop-like interactive features (e.g., Google Maps).
- 7. Web 1.0:** Static content, read-only web.
- 8. Web 2.0:** Interactive, user-generated content (social media, blogs).
- 9. Web 3.0:** Semantic, AI-driven, decentralized web (blockchain, smart apps).
- 10. Code Logic:** It refers to the step-by-step programming flow to solve a problem.
- 11. What is Mongoose?:** An ODM library for Node.js to interact with MongoDB easily.
- 12. Command to retrieve data from Mongo:** `db.collection.find()`.
- 13. MongoDB find query outputs?:** Returns all matched documents; for 20 elements, it fetches 20.

- 14.What is MongoDB?: A NoSQL, document-oriented database storing data in JSON-like format.**
- 15.Why MongoDB over SQL?: Schema-less, flexible, scalable, handles unstructured data well.**
- 16.What is TypeScript?: A superset of JavaScript adding static types.**
- 17.Types of inheritance in TypeScript: Single, Multi-level, Hierarchical.**
- 18.Diff between JS and TS: TS has types, interfaces, OOP features; JS is dynamically typed.**
- 19.Node to Mongo connection: Using MongoDB driver or Mongoose to connect via URI.**
- 20.What is Flask?: A lightweight Python web framework for building web applications.**
- 21.Difference Django vs Flask: Django is full-stack & batteries-included; Flask is minimal & flexible.**
- 22.Routing: Mapping URLs to specific functions or controllers in web apps.**
- 23.HTTP Methods: GET, POST, PUT, DELETE, PATCH, OPTIONS, HEAD.**
- 24.RIA Integration: Using AJAX, JavaScript, Flash to create dynamic web experiences.**
- 25.Real-life RIA example: Google Maps, Gmail, Trello.**



- 26.What is AJAX?: Asynchronous JavaScript and XML, for sending/receiving data without page reload.**
- 27.AJAX Full Form: Asynchronous JavaScript And XML.**
- 28.What is Asynchronous?: Operations that run independently without blocking the main flow.**
- 29.What is Angular?: A TypeScript-based web framework for building SPA (Single Page Applications).**
- 30.Framework of AJAX/How it operates: Uses XMLHttpRequest/fetch API to send/receive data without reloading page.**
- 31.AngularJS Usage: For creating dynamic, single-page, data-driven web apps.**
- 32.When to use AngularJS: For complex client-side applications with dynamic content updates.**
- 33.Angular Directives: Special HTML attributes to extend functionality (ngModel, ngFor, ngIf).**
- 34.Angular Services: Singleton objects for shared data & logic (e.g., HttpClientService).**
- 35.Angular Filters: Used to format data (e.g., date, currency, custom filters in templates).**
- 36.ng-directives: Built-in AngularJS attributes for DOM manipulation.**
- 37.ng-model: Binds input field to application data.**

**38.ng-bind: Binds application data to HTML view.**

**39.What is XML?: A markup language to store & transport structured data.**

**40.Diff between XML & HTML: XML is for data storage & transport (custom tags), HTML is for web content & structure.**

**Ajax:**

**server.js**

```
const express = require('express');  
const app = express();  
const cors = require('cors');
```

```
app.use(cors());  
app.use(express.json());
```

// Dummy Product Catalog

```
const products = [  
  { id: 1, name: 'iPhone 14' },  
  { id: 2, name: 'Samsung Galaxy S23' },  
  { id: 3, name: 'MacBook Pro' },  
  { id: 4, name: 'Dell XPS 13' },  
  { id: 5, name: 'Sony Headphones' }  
];
```

// Search API

```
app.get('/search', (req, res) => {  
  const q = req.query.q?.toLowerCase() || "";  
  const result = products.filter(p => p.name.toLowerCase().includes(q));  
  res.json(result);  
});
```

```
app.listen(5000, () => console.log('Server running on port 5000'));
```

## **Index.html**

```
<!DOCTYPE html>
<html>
<head>
  <title>AJAX Product Search</title>
  <style>
    input { padding: 10px; width: 300px; }
    .result { margin-top: 10px; }
  </style>
</head>
<body>
  <h2>Product Search</h2>
  <input type="text" id="search" placeholder="Search products...">
  <div class="result" id="result"></div>

  <script>
    const searchInput = document.getElementById('search');
    searchInput.addEventListener('input', async () => {
      const query = searchInput.value;
      const res = await fetch(`http://localhost:5000/search?q=${query}`);
      const products = await res.json();

      document.getElementById('result').innerHTML = products.map(p =>
        `<p>${p.name}</p>`).join("") || '<p>No products found</p>';
    });
  </script>
</body>
</html>
```

**node [server.js](#)**

**mkdir myproject**

**cd myproject**

**npm init -y**

**npm install express cors mongoose**

**npm install mongodb**

**npm install mongoose**

**npm install cors**

**npm install express**