TypeScript

TypeScript is a syntactic superset of JavaScript which adds static typing.It means that TypeScript adds syntax on top of JavaScript, allowing developers to add types.

TypeScript is a typed superset, meaning that it adds rules about how different kinds of values can be used.

TypeScript being a "Syntactic Superset" means that it shares the same base syntax as JavaScript, but adds something to it.

TypeScript is also a programming language that preserves the runtime behavior of JavaScript. For example, dividing by zero in JavaScript produces Infinity instead of throwing a runtime exception. As a principle, TypeScript never changes the runtime behavior of JavaScript code.This means that if you move code from JavaScript to TypeScript, it is guaranteed to run the same way, even if TypeScript thinks that the code has type errors.Keeping the same runtime behavior as JavaScript is a foundational promise of TypeScript because it means you can easily transition between the two languages without worrying about subtle differences that might make your program stop working.

TypeScript doesn’t provide any additional runtime libraries. Your programs will use the same standard library (or external libraries) as JavaScript programs, so there’s no additional TypeScript-specific framework to learn.

TypeScript doesn’t consider any JavaScript code to be an error because of its syntax. This means you can take any working JavaScript code and put it in a TypeScript file without worrying about exactly how it is written.

Why should I use TypeScript?

JavaScript is a loosely typed language. It can be difficult to understand what types of data are being passed around in JavaScript.In JavaScript, function parameters and variables don't have any information! So developers need to look at documentation, or guess based on the implementation.

TypeScript allows specifying the types of data being passed around within the code, and has the ability to report errors when the types don't match. TypeScript uses compile time type checking. Which means it checks if the specified types match before running the code.

TypeScript being converted into JavaScript means it runs anywhere that JavaScript runs!

Installing the Compiler:

TypeScript has an official compiler which can be installed through npm.

```
npm install typescript --save-dev
```

The compiler is installed in the node_modules directory and can be run with:

```
npx tsc.
```

Which should give you an output similar to:

```
Version 4.5.5
tsc: The TypeScript Compiler - Version 4.5.5
```

Configuring the compiler:

By default the TypeScript compiler will print a help message when run in an empty project. The compiler can be configured using a tsconfig.json file.

You can have TypeScript create tsconfig.json with the recommended settings with:

```
npx tsc --init
```

Which should give you an output similar to:

Created a new tsconfig.json with:

```
TS
  target: es2016
  module: commonjs
  strict: true
  esModuleInterop: true
  skipLibCheck: true
  forceConsistentCasingInFileNames: true
```

You can learn more at https://aka.ms/tsconfig.json

You can open the file in an editor to add those options. This will configure the TypeScript compiler to transpile TypeScript files located in the src/ directory of your project, into JavaScript files in the build/ directory.

This is one way to quickly get started with TypeScript. **There are many other options available** such as create a angular-app, create-react-app template, a node starter project, and a webpack plugin etc.

TypeScript Types:

Typescript has both simple types(primitive types) and special types.

There are three main primitives in JavaScript and TypeScript.

boolean - true or false values

number - whole numbers and floating point values

string - text values

There are also 2 less common primitives used in later versions of Javascript and TypeScript. Those are bigint and symbol.

bigint – whole numbers and floating point values, but allows larger negative and positive numbers than the number type.

symbol - used to create a globally unique identifier.

Type Assignment:

When creating a variable, there are two main ways TypeScript assigns a type:

Explicit--writing with the type

Example:

```
firstName: string = "Evalitic";
```

Implicit--TypeScript will "guess" the type, based on the assigned value

Example:

```
firstName = "Evalitic";
```

TypeScript will throw an error if data types do not match. JavaScript will not throw an error for mismatched types.

Example:

```
firstName: string = "Evalitic";
firstName = 33;--->throw an error.
```

We can stop implicit assignment. This behavior can be disabled by enabling noImplicitAny as an option in a TypeScript's project tsconfig.json. That is a JSON config file for customizing how some of TypeScript behaves.

TypeScript Special Types

TypeScript has special types that may not refer to any specific type of data.

Type: any

any is a type that disables type checking and effectively allows all types to be used.

Example:

v: any = true;

v = "string"; // no error as it can be "any" type

Math.round(v); // no error as it can be "any" type

Type: unknown

unknown is a similar, but safer alternative to any.

TypeScript will prevent unknown types from being used

unknown is best used when you don't know the type of data being typed. To add a type later, you'll need to cast it.

Casting is when we use the "as" keyword to say property or variable is of the casted type.

Type: never

never effectively throws an error whenever it is defined.

never is rarely used, especially by itself, its primary use is in advanced generics.

Type: undefined & null

undefined and null are types that refer to the JavaScript primitives undefined and null respectively.

y: undefined = undefined;

z: null = null;

These types don't have much use unless strictNullChecks is enabled in the tsconfig.json file.

TypeScript Arrays

An array is a special variable, which can hold more than one value.

An array can hold many values under a single name, and you can access the values by referring to an index number.

Syntax:

```
array_name = [item1, item2, ...];
```

Example

```
cars = ["BMW", "AUDI", "BENZ"];
```

Spaces and line breaks are not important. A declaration can span multiple lines as follows:

```
cars = [
  "Saab",
  "Volvo",
  "BMW"
];
```

You can also create an array, and then provide the elements as follows:

```
cars = [];
cars[0]= "BMW";
cars[1]= "Volvo";
cars[2]= "BENTLY";
```

we can also creates an Array, and assigns values to it as follows:

```
cars = new Array("Saab", "Volvo", "BMW");
```

Accessing Array Elements:

You access an array element by referring to the index number.

Example:

cars = ["TATA", "SUZIKI", "KIA"];

car = cars[0];

Changing an Array Element:

We can change the array elements by using its index number.

Example:

mobiles = ["vivo", "oppo", "moto"];

cars[0] = "samsung";

Converting an Array to a String:

The JavaScript method toString() converts an array to a string of (comma separated) array values.

fruits = ["Banana", "Orange", "Apple", "Mango"];

x: String = fruits.toString();

Access the Full Array:

the full array can be accessed by referring to the array name.

fruits = ["Banana", "Orange", "Apple", "Mango"];

console.log(fruits);

const keyword:

In 2015, JavaScript introduced an important new keyword const.

It has become a common practice to declare arrays using const.

An array declared with const cannot be reassigned.

Example:

const cars = ["Saab", "Volvo", "BMW"];

cars = ["Toyota", "Volvo", "Audi"];   // ERROR

The keyword const is a little misleading.It does NOT define a constant array. It defines a constant reference to an array. we can still change the elements of a constant array.

You can change the elements of a constant array.

example:

```
// You can create a constant array:
const cars = ["Saab", "Volvo", "BMW"];
// You can change an element:
cars[0] = "Toyota";
// You can add an element:
cars.push("Audi");
```

TypeScript Tuples:

A tuple is a typed array with a pre-defined length and types for each index.

Tuples are great because they allow each element in the array to be a known type of value.

To define a tuple, specify the type of each element in the array.

Example 1:

```
// define our tuple
let ourTuple: [number, boolean, string];
// initialize correctly
ourTuple = [5, false, 'Coding God was here'];
```

Example 2:

```
// define our tuple
let ourTuple: [number, boolean, string];
// initialized incorrectly which throws an error
ourTuple = [false, 'Coding God was mistaken', 5];
```

Even though we have a boolean, string, and number the order matters in our tuple and will throw an error.

Readonly Tuple:

A good practice is to make your tuple readonly. Tuples only have strongly defined types for the initial values.

Example:

```
// define our readonly tuple
const ourReadonlyTuple: readonly [number, boolean, string] = [5, true, 'The Real Coding God'];

ourReadonlyTuple.push('Coding God took a day off'); // throws error as it is readonly.
```

To learn more about access modifiers like readonly we can see in 'Type script classes'.

TypeScript Object Types

Objects are variables too. But objects can contain many values.

The values are written as name:value pairs (name and value separated by a colon).

Object definition & declaration:

```
const car: { type: string, model: string, year: number } = {
  type: "Toyota",
  model: "Corolla",
  year: 2009
};
```

Object direct declaration without definition:

```
const car = {type:"Fiat", model:"500", color:"white"}; //object definition
```

It is a common practice to declare objects with the const keyword.

Spaces and line breaks are not important. An object definition can span multiple lines.

Example:

```
const person = {
  firstName: "John",
  lastName: "Doe",
  age: 50,
  eyeColor: "blue"
};
```

Accessing Object Properties:

You can access object properties in two ways.

```
objectName.propertyName
        or
objectName["propertyName"]
```

Example:

```
person.lastName;
or
person["lastName"];
```

Binding in angular:

In Angular, binding refers to the process of establishing a connection between the component's data and the view (HTML template). It allows you to synchronize the data between the component and the view, so any changes made in the component are reflected in the view, and vice versa.

There are several types of bindings in Angular:

Interpolation ({{ }}): This is the simplest form of binding where you can embed expressions in double curly braces directly in the HTML template. Angular evaluates the expression and replaces the interpolation with the result.

Example:

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

Property Binding ([ ]): Property binding allows you to set the value of an HTML element property to a component's property. It binds data from the component to the HTML element property.

```
<img [src]="imageUrl">
```

Event Binding (( )): Event binding allows you to listen to events raised by the HTML elements and execute some logic in response to those events in the component.

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

Two-Way Binding ([( )]): Two-way binding is a combination of property binding and event binding. It allows data to flow both from the component to the view and from the view to the component. It's typically used with form elements.

```
<input [(ngModel)]="username">
```

These binding mechanisms are essential for creating dynamic and interactive Angular applications by facilitating the communication between the component and its corresponding view. They help in keeping the UI in sync with the underlying data and responding to user interactions effectively.

Examples on Interpolation:

Example1: Displaying Component Property using interpolation

html code:

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

TS code:

```
export class AppComponent {
  name = 'eva_app_1';
}
```

Example2: Do Arithmetic Operations using interpolation

html code:

```
<p>2 + 2 equals {{ 2 + 2 }}</p>
```

Example3: String Concatenation using interpolation

html code:

```
<p>{{ "Hello, " + name }}</p>
```

TS code:

```
export class AppComponent {
  name = 'eva_app_1';
}
```

Example4: Accessing Object Properties using interpolation

html code:

```
<p>User: {{ user.firstName }} {{ user.lastName }}</p>
```

TS code:

```
export class AppComponent {
  user = { firstName: 'John', lastName: 'Doe' };
}
```

Example5: Calling Component Methods using interpolation

html code:

```
<p>{{ getGreeting() }}</p>
```

TS code:

```
// In the component
getGreeting() {
    return "Hello, " + this.name;
}
```

Example6: Interpolation Using Component Methods with Parameters

html code:

```
<p>{{ greet('Alice') }}</p>
```

TS code:

```
// In the component
greet(name: string) {
    return "Hello, " + name;
}
```

Example7: Interpolation Accessing Nested Object Properties

html code:

```
<p>{{ user.name }} lives in {{ user.address.city }}, {{ user.address.country }}</p>
```

TS code:

```
// In the component
user = {
   name: 'John',
   address: {
      city: 'New York',
      country: 'USA'
   }
};
```

Example8: Interpolation Using Ternary Operator for Conditional Display

html code:

```
<p>{{ isLoggedIn ? 'Welcome back' : 'Please log in' }}</p>
```

TS code:

```
// In the component
isLoggedIn = true;
```

Example9: Displaying Dynamic URLs using interpolation

html code:

```
<a href="/users/{{ userId }}">User Profile</a>
```

TS code:

```
// In the component
userId = 123;
```

Example10: Interpolation Using Math Operations

html code:

```
<p>{{ num1 }} + {{ num2 }} equals {{ num1 + num2 }}</p>
```

TS code:

```
// In the component
num1 = 10;
num2 = 5;
```

Example11: Displaying Dynamic Images using interpolation

html code:

```
<img src="/assets/{{ imageName }}">
```

TS code:

```
// In the component
imageName = 'logo.png';
```

Example12: Dynamic Styling using interpolation

html code:

```
<div style="color: {{ textColor }}; font-size: {{ fontSize }}px;">Styled Content</div>
```

TS code:

```
// In the component
textColor = 'red';
fontSize = 16;
```

Examples on Property Binding:

Example1: Binding to a Property

html code:

```
<img [src]="imageUrl">
```

TS code:

```
// In the component
imageUrl = "/assets/pic1.jpg";
```

Example2: Conditional Property Binding

html code:

```
<button [disabled]="isDisabled">Click me</button>
```

TS code:

```
// In the component
isDisabled = true;
```

Example3: Setting CSS Classes Dynamically using property binding

css code:

```
/* Define styles for the 'error' class */
.error {
  color: red;
  font-weight: bold;
}
```

TS code:

```
// app.component.ts
hasError = true;
```

HTML code:

```
<!-- app.component.html -->
<div [class.error]="hasError">Error message</div>
```

Example4: Binding Multiple Classes using property binding

css code:

```
/* Define styles for the 'error' class */
.error {
  color: red;
  font-weight: bold;
}
/* Define styles for the 'warning' class */
.warning {
  color: orange;
}
```

HTML code:

```
<div [class.error]="hasError" [class.warning]="hasWarning">Message</div>
```

TS code:

```
// app.component.ts
hasError = true;
hasWarning = false;
```

Example5: Dynamic Styling using property binding

HTML code:

```
<div [style.background-color]="isHighlighted ? 'yellow' : 'transparent'"
[style.font-size.px]="fontSize">
  Dynamic Styling Example
</div>
```

TS code:

```
  isHighlighted = true;
  fontSize = 20;
```

Example6: Binding to DOM Properties

HTML code:

```
<input [value]="username">
```

TS code:

```
username="kiran";
```

Example7: Binding to Object Properties

HTML code:

```
<h2>User Details</h2>
<p>Name: <span [innerText]="user.name"></span></p>
<p>Age: <span [innerText]="user.age"></span></p>
<p>Email: <span [innerText]="user.email"></span></p>
```

TS code:

```
export class AppComponent {
        user = {
          name: 'John Doe',
          age: 30,
          email: 'john@example.com'
        };
}
```

Example8: another example for binding to object properties

HTML code:

```
<a [href]="user.profileUrl">{{ user.name }}</a>
```

TS code:

```
export class AppComponent {
  users = { name: 'John Doe', profileUrl: 'https://example.com/john' };
}
```

Examples on Event Binding:

Example1: create sample event binding

HTML code:

```
<button (click)="sayHello()">Say Hello</button>
```

TS code:

```
sayHello() {
    alert('Hello, world!');
  }
```

Example2: create click event for sum of two numbers

HTML code:

```
<button (click)="sum(5,9)">SUM</button>
<h1>{{result}}</h1>
```

TS code:

```
export class AppComponent {
  result = 0;
  sum(num1:number, num2:number){
    this.result = num1 + num2;
  }
}
```

Example3: create mouse over and mouse leave event bindings

HTML code:

```
<img src="/assets/rabbit.jpg" [width]= w [height]= h (mouseover)="zoom()"
(mouseleave)="normal()" />
```

TS code:

```
export class Task5Component {
  w = 200;
  h = 200;

  zoom(){
    this.w = 400;
    this.h = 400;
  }

  normal(){
    this.w = 200;
    this.h = 200;
  }
}
```

Example4: creat input event to bind input values

HTML code:

```
<input type="text" (input)="hi($event)">
<h1>{{result}}</h1>
```

TS code:

```
export class AppComponent {
  result = "";
  hi(event: any){
    this.result=event.target.value;
  }
}
```

Examples for Two-way Binding:

Example1: Sample two way binding

HTML code:

```
<input type="text" [(ngModel)]="name">
<p>Hello, {{ name }}!</p>
```

TS code:

```
export class AppComponent {
  name: string = '';
}
```

Example2: Sum of two numbers using two way binding

HTML code:

```
<!-- app.component.html -->
<div>
        <label for="num1">Enter first number:</label>
        <input type="number" id="num1" [(ngModel)]="num1">
  </div>
  <div>
        <label for="num2">Enter second number:</label>
        <input type="number" id="num2" [(ngModel)]="num2">
  </div>
  <button (click)="calculateSum()">Calculate Sum</button>
       <div *ngIf="sum !== null">
        <p>The sum of {{ num1 }} and {{ num2 }} is: {{ sum }}</p>
  </div>
```

TS code:

```
num1: number=0;
  num2: number=0;
  sum: number | null = null;
  calculateSum() {
    if (this.num1 !== undefined && this.num2 !== undefined) {
      this.sum = this.num1 + this.num2;
    }
  }
```

Example3: Bind select option values using two way binding

HTML code:

```
<select [(ngModel)]="selectedOption">
   <option value="option1">Option 1</option>
   <option value="option2">Option 2</option>
   <option value="option3">Option 3</option>
 </select>
 <p>Selected option: {{ selectedOption }}</p>
```

TS code:

```
export class AppComponent {
  selectedOption: string = 'option1';
}
```

Example4: Bind radio buttons in two way

HTML code:

```
<label>
   <input type="radio" [(ngModel)]="selectedColor" value="red"> Red
 </label>
 <label>
   <input type="radio" [(ngModel)]="selectedColor" value="green"> Green
 </label>
 <label>
   <input type="radio" [(ngModel)]="selectedColor" value="blue"> Blue
 </label>
 <p>Selected color: {{ selectedColor }}</p>
```

TS code:

```
selectedColor: string = 'red';
```

Directives:

In Angular, directives are a type of component that allows you to add behavior to elements in the DOM (Document Object Model). They can be classified into three types:

Structural Directives:

These directives change the DOM layout by adding, removing, or manipulating elements based on certain conditions. Structural directives are prefixed with an asterisk (*) in the HTML template. Structural directives help in conditionally rendering elements, iterating over lists, and handling other structural changes in the DOM.

Angular provides several built-in structural directives for common use cases:

*ngIf: Conditionally adds or removes the host element and its children based on a condition.

*ngFor: Iterates over a collection and generates multiple instances of the host element for each item in the collection.

*ngSwitch: Conditionally renders different elements based on a provided value.

Examples on Structural Directives:

Example1: Conditional rendering

HTML code:

```
<div *ngIf="isAdmin || (isModerator && hasPermission)">
  You have administrative privileges.
 </div>
 <div *ngIf="isLoggedIn">
  Welcome, {{ username }}!
 </div>
```

TS code:

```
export class AppComponent {
  isAdmin = false;
  isModerator = true;
  hasPermission = true;
  isLoggedIn = true;
  username = "Jhon"
}
```

Example2: Conditional expression rendering

HTML code:

```
<div *ngIf="data.length > 0">
   There is data to display.
  </div>
```

TS code:

```
export class AppComponent {
  data = [1,2,4];
}
```

Example3: else conditional rendering

HTML code:

```
<div *ngIf="isLoggedIn; else notLoggedIn">
   Welcome, {{ username }}!
  </div>
  <ng-template #notLoggedIn>
    <div>
      Please log in to access the content.
    </div>
  </ng-template>
```

TS code:

```
isLoggedIn = false;
username = "Smith";
```

Explanation:

Here, if isLoggedIn is true, then the first div will be rendered, otherwise, the content inside the ng-template with the #notLoggedIn template reference id will be rendered.

What is Template Reference Variable?

In Angular, a template variable, also known as a template reference variable, is a variable declared in an Angular template (HTML) that provides a reference to a DOM element, directive, or component within the template. Template variables are prefixed with the # symbol and can be used to reference elements, access their properties or invoke methods, and share data between the template and the component class. You can use template variables with event bindings or property bindings to interact with DOM elements programmatically.

Template variables can be used to pass data from the template to the component class. This is commonly done by passing the template variable as an argument to event handlers or methods in the component class.

Template variables are local to the template in which they are declared. They cannot be accessed outside of the template in which they are defined. This ensures encapsulation and prevents naming conflicts.

What is ng-template?

ng-template is an Angular directive used to declare a template that can be rendered later by Angular. It is a non-visible element that Angular uses for template rendering, data manipulation, and structural directives such as *ngIf, *ngFor, and *ngSwitch.

ng-template is declared as a regular HTML element in Angular templates. It does not render anything visible in the DOM.

ng-template is often used with the ngTemplateOutlet directive to render the content of the template at a specific location in the DOM. This allows for dynamic rendering of templates based on conditions or user interactions.

Example 4: Write a code to find biggest of two numbers using ngIf in angular

HTML code:

```
<div>
  <h2>Find the Biggest Number</h2>
  <div>
    <label>Number 1: </label>
    <input type="number" [(ngModel)]="number1">
  </div>
  <div>
    <label>Number 2: </label>
    <input type="number" [(ngModel)]="number2">
  </div>
  <button (click)="findBiggestNumber()">Find Biggest Number</button>
  <div *ngIf="biggestNumber !== null">
    The biggest number is: {{ biggestNumber }}
  </div>
 </div>
```

TS code:

```
export class AppComponent {
  number1: number =0;
  number2: number = 0;
  biggestNumber: number = 0;

  findBiggestNumber() {
    if (this.number1 > this.number2) {
      this.biggestNumber = this.number1;
    } else{
      this.biggestNumber = this.number2;
    }
  }
}
```

Example5: Use ngIf, then and else for login and logout conditions.

HTML Code:

```
<div>
  <h2>Login Example</h2>

  <button *ngIf="!isLoggedIn" (click)="login()">Log In</button>
  <button *ngIf="isLoggedIn" (click)="logout()">Log Out</button>

  <ng-template #loggedInTemplate>
    <p>Welcome, you are logged in!</p>
  </ng-template>

  <ng-template #loggedOutTemplate>
    <p>Please log in to continue.</p>
  </ng-template>
  <div *ngIf="isLoggedIn; then loggedInTemplate else
loggedOutTemplate"></div>
 </div>
```

TS Code:

```
export class AppComponent {
 isLoggedIn: boolean = false;

 login() {
  // Simulating a login action
  this.isLoggedIn = true;
 }

 logout() {
  // Simulating a logout action
  this.isLoggedIn = false;
 }
}
```

Example6 : write a code to demonstrate template variable in angular.

HTML code:

```
<div>
  <label for="num1">Enter first number:</label>
  <input type="number" id="num1" #input1>
 </div>
 <div>
  <label for="num2">Enter second number:</label>
  <input type="number" id="num2" #input2>
 </div>
 <button (click)="calculateSum(input1.valueAsNumber,
input2.valueAsNumber)">Calculate Sum</button>
 <div *ngIf="sum !== null">
  <p>The sum is: {{ sum }}</p>
 </div>
```

TS code:

```
export class AppComponent {
  sum: number | null = null;

  calculateSum(num1: number, num2: number) {
    this.sum = num1 + num2;
  }
}
```

Example 7: Write a code to get which style classes are used for element and get inner text of element.

HTML code:

```
<h1 class="bgc fgc" #x>Welcome to Angular</h1>


{{"classes which are used for Heading"+x.classList}}<br>
{{ "Inner content of Heading"+x.innerText}}
```

Example8: Using ngFor Iterating Over a Range of Numbers

HTML code:

```
<h1 *ngFor="let i of [0, 1, 2, 3, 4]">{{ i }}</h1>
```

Example9: Using ngFor display all array values with their index

HTML code:

```
<h1 *ngFor="let item of items; let i = index"> {{ i + 1 }}. {{ item }} </h1>
```

TS code:

```
export class AppComponent {
  items=["laptop", "mobile", "pendrive", "hard disk"]
}
```

Example10: using ngFor display all object values.

HTML code:

```
<h1 *ngFor="let employee of employees"> {{ employee.id }}: {{ employee.name }}: {{employee.city}} </h1>
```

TS code:

```
export class AppComponent {
  employees=[{"id":111, "name":"ram", "city":"Ayodya"},{"id":112, "name":"robert", "city":"Mumbai"},{"id":113, "name":"raheem", "city":"Haryana"},{"id":114, "name":"krishna", "city":"dwaraka"}]
}
```

Example11: Set options for select using ngFor.

HTML code:

```
<select [(ngModel)]="selectedOption">
      <option *ngFor="let option of options" [ngValue]="option">{{ option.name }}</option>
  </select>
  <p>Selected option: {{ selectedOption | json }}</p>
```

TS code:

```
export class AppComponent {
  options = [
    { id: 1, name: 'Option 1' },
    { id: 2, name: 'Option 2' },
    { id: 3, name: 'Option 3' }
  ];
  selectedOption = this.options[0];
}
```

Example12: Set options for select using ngFor.

HTML code:

```
        <div *ngFor="let fruit of fruits">
          <label>
            <input type="checkbox" [(ngModel)]="fruit.selected"> {{ fruit.name }}
          </label>
         </div>
         <p>Selected fruits: {{ selectedFruits }}</p>
```

TS code:

```
        fruits = [
          { name: 'Apple', selected: false },
          { name: 'Banana', selected: false },
          { name: 'Orange', selected: false }
         ];
         get selectedFruits(): string[] {
           return this.fruits.filter(fruit => fruit.selected).map(fruit => fruit.name);
         }
```

Explanation:

In this example, ngModel is used with checkboxes and an array of objects representing fruits. The selected property of each fruit object is bound to the checkbox state using [(ngModel)]. When the user checks or unchecks a checkbox, the selected property of the corresponding fruit object is updated accordingly. The selectedFruits property returns an array of names of the selected fruits, which is displayed in the template.

Example13: select choice using ngSwitch

HTML code:

```
<div [ngSwitch]="choice">
    <p *ngSwitchCase="'A'">You selected option A</p>
    <p *ngSwitchCase="'B'">You selected option B</p>
    <p *ngSwitchCase="'C'">You selected option C</p>
    <p *ngSwitchDefault>No option selected</p>
  </div>
```

TS code:

```
export class AppComponent {
  choice: string = 'B';
}
```

Example14: another example for ngSwitch

HTML code:

```
<div [ngSwitch]="dayOfWeek">
  <p *ngSwitchCase="'Monday'">It's Monday!</p>
  <p *ngSwitchCase="'Tuesday'">It's Tuesday!</p>
  <p *ngSwitchCase="'Wednesday'">It's Wednesday!</p>
  <p *ngSwitchCase="'Thursday'">It's Thursday!</p>
  <p *ngSwitchCase="'Friday'">It's Friday!</p>
  <p *ngSwitchCase="'Saturday'">It's Saturday!</p>
  <p *ngSwitchCase="'Sunday'">It's Sunday!</p>
  <p *ngSwitchDefault>Invalid day of the week</p>
</div>
```

TS code:

```
export class EventbindingdemoComponent {
  dayOfWeek: string = 'Monday';
}
```

Example15: arithmetic operations using ngSwitch

HTML code:

```
<div [ngSwitch]="operation">
  <p *ngSwitchCase="'add'">Result of addition: {{ operand1 + operand2
}}</p>
  <p *ngSwitchCase="'subtract'">Result of subtraction: {{ operand1 -
operand2 }}</p>
  <p *ngSwitchCase="'multiply'">Result of multiplication: {{ operand1 *
operand2 }}</p>
  <p *ngSwitchCase="'divide'">Result of division: {{ operand1 / operand2
}}</p>
```

```
    <p *ngSwitchDefault>Invalid operation</p>
  </div>
```

TS code:

```
export class EventbindingdemoComponent {
  operation: string = 'add';
  operand1: number = 10;
  operand2: number = 5;
}
```

Example16: arithmetic operations using inputs ngSwitch

HTML code:

```
<input type="number" [(ngModel)]="num1">
<input type="number" [(ngModel)]="num2">
<input type="text" [(ngModel)]="op">

<div [ngSwitch]="op">
    <p *ngSwitchCase="'+'">Result of addition: {{ num1 + num2 }}</p>
    <p *ngSwitchCase="'-'">Result of subtraction: {{ num1 - num2 }}</p>
    <p *ngSwitchCase="'*'">Result of multiplication: {{ num1 * num2 }}</p>
    <p *ngSwitchCase="'/'">Result of division: {{ num1 / num2 }}</p>
    <p *ngSwitchCase="''"></p>
    <p *ngSwitchDefault>Invalid operation</p>
  </div>
```

TS code:

```
export class AppComponent {
  op:string="";
  num1:number = 0;
  num2:number = 0;
}
```

Attribute Directives: These directives change the appearance or behavior of an element by applying CSS styles, adding event listeners, or modifying element properties. Attribute directives are applied to elements as attributes and modify the behavior or appearance of those elements.

Examples of attribute directives in Angular include ngStyle, ngClass, and ngModel.

The syntax for applying attribute directives is

[directiveName] or [directiveName]="expression".

For example, [ngClass], [ngStyle], [ngModel], etc.

Attribute directives can accept input values to customize their behavior. Input values are typically provided as attributes on the same element where the directive is applied.

For example, [ngClass]="{ 'active': isActive }".

Example1: Conditional Class Based on a Boolean Expression

CSS code:

```
.active {
  background-color: green;
 }

 .disabled {
  opacity: 0.5;
 }
```

HTML code:

```
<h1 [ngClass]="{'active': isActive, 'disabled': isDisabled}">
 welcome to angular
</h1>
```

TS code:

```
export class AppComponent {
 isActive = false;
 isDisabled = true;
}
```

Example2: Class Based on Component Property

CSS code:

```
.text-danger {
    color: red;
  }
```

HTML code:

```
<h1 [ngClass]="customClass">
  welcome to angular
</h1>
```

TS code:

```
export class AppComponent {
  customClass: string = 'text-danger';
}
```

Example3: Class Based on Expression

CSS code:

```
.active {
    background-color: green;
  }
  .inactive {
    background-color: red;
  }
```

HTML code:

```
<h1 [ngClass]="isActive ? 'active' : 'inactive'">
  welcome to angular
</h1>
```

TS code:

```
export class EventbindingdemoComponent {
  isActive: boolean = true;
}
```

Example4: Class Based on Object(same as Example 1 but we call as object also)

CSS Code:

```
.bg-primary {
   background-color: blue;
 }
 .text-danger {
   color: red;
 }
```

HTML Code:

```
<h1 [ngClass]="{'bg-primary': isPrimary, 'text-danger': isError}">
 welcome to angular
</h1>
```

TS code:

```
export class AppComponent {
 isPrimary: boolean = true;
 isError: boolean = false;
}
```

Example5: Multiple Classes with Array

CSS code:

```
.text-success {
   color: green;
 }


 .font-weight-bold {
   font-weight: bold;
 }
```

HTML code:

```
<h1 [ngClass]="['text-success', 'font-weight-bold']">
 welcome to angular
</h1>
```

Example 6: Conditional Style Based on a Boolean Expression

HTML code:

```
<h1 [ngStyle]="{ 'color': isActive ? 'green' : 'red', 'font-size': isLarge ?
'24px' : '16px' }">
  welcome to angular
</h1>
```

TS code:

```
export class AppComponent {
  isActive: boolean = true;
  isLarge: boolean = false;
}
```

Example 7: Style Based on Component Property

HTML code:

```
<h1 [ngStyle]="customStyles">
  welcome to angular
</h1>
```

TS code:

```
export class AppComponent {
  customStyles = {
    'color': 'blue',
    'font-size': '20px',
    'background-color': 'yellow'
  };
}
```

Example 8: Dynamic Styles Based on Expressions

HTML code:

```
<h1 [ngStyle]="{ 'background-color': isPrimary ? 'blue' : 'gray', 'border':
(isError && isWarning) ? '2px solid red' : 'none' }">
  welcome to angular
</h1>
```

TS code:

```
export class EventbindingdemoComponent {
  isPrimary: boolean = true;
  isError: boolean = false;
  isWarning: boolean = true;
}
```

Example 9: Dynamic Styles with Calculations

HTML code:

```
<h1 [ngStyle]="{ 'width.px': containerWidth, 'height.px': containerHeight }">
  welcome to angular
</h1>
```

TS code:

```
export class AppComponent {
  containerWidth: number = 200;
  containerHeight: number = 100;
}
```

Component Directives:

These are the most common type of directives in Angular. They are components with templates and logic. Component directives in Angular are essentially components that encapsulate both the visual aspect (HTML template) and the behavior (component class) into a single entity. Component directives enable you to create reusable UI components, encapsulate logic, and maintain the state of the application. This encapsulation helps in creating modular, reusable, and maintainable components.

Component directives can communicate with other components by using input properties and output events. Input properties allow you to pass data into a component, while output events enable a component to emit events that can be listened to by parent components.

Component directives have access to Angular's lifecycle hooks, which allow you to execute logic at specific points in the component's lifecycle. These hooks include ngOnInit, ngOnChanges, ngAfterViewInit, ngOnDestroy, etc., and they provide opportunities to perform initialization, cleanup, and other tasks.

Component directives can leverage Angular's dependency injection system to inject services, dependencies, and other components into their constructor. This enables you to access external resources and services within your component.

Component directives are often used in Angular's routing system to define the views associated with different routes. Each route can be associated with a component directive, allowing you to navigate between different views in your application.

Overall, component directives play a central role in Angular applications, allowing you to create modular, reusable, and maintainable UI components with encapsulated logic and behavior. They are the building blocks of Angular applications and are essential for creating dynamic and interactive user interfaces.

Example for Component directives:

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

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrl: './app.component.css'
})
```

Pipes in Angular:

In Angular, pipes are a way to transform data in templates before displaying it to the user. Pipes allow you to format data such as strings, numbers, dates, and arrays in a way that is easy to display in the template.

Angular provides a set of built-in pipes that cover common transformations such as formatting dates, numbers, currencies, and uppercase/lowercase conversions.

Examples of built-in pipes include lowercase, uppercase, date, decimal, percent, slice etc.

Pipes are used in templates with the | operator followed by the pipe name and any parameters. For example, {{ value | uppercase }} will transform the value to uppercase before displaying it in the template.

Some pipes accept parameters to customize their behavior. Parameters are passed to the pipe using colon-separated values. For example, {{ value | currency:'USD':true:'1.2-2' }} will format the value as a currency in US dollars with two decimal places.

You can also create custom pipes to perform custom transformations on data. Custom pipes are created using the @Pipe decorator and implementing the PipeTransform interface. They can accept parameters and perform any logic you need to transform the data.

You can chain multiple pipes together in a template to perform complex transformations. The output of one pipe becomes the input of the next pipe in the chain.

By default, pipes in Angular are pure, meaning they are stateless and do not change unless the input data changes. However, you can create impure pipes if you need them to be re-evaluated on every change detection cycle. Impure pipes can have performance implications, so they should be used judiciously.

Angular provides the AsyncPipe for handling asynchronous data streams such as promises or observables. The AsyncPipe automatically subscribes to the data source, retrieves the emitted values, and then unwraps the value from the observable or promise.

Example1: builtin pipes in angular

```
<p>USA Price: {{ price | currency }}</p>
<p>Indian Price: {{ price | currency : 'INR' }}</p>
<p>Percentage {{ percentage | percent }}</p>
<p>Today is {{ today | date }}</p>
<p>Full date of Today is {{ today | date: 'fullDate' }}</p>
<p>{{ today | date:'MMM dd, yyyy' }}</p>
<p>slice 0 to 5 {{ 'Hello World' | slice: 0:5 }}</p>
<p>slice -5 {{ 'Hello World' | slice: -5 }}</p>
<p>{{ object | json }}</p>
<p *ngFor="let item of object | keyvalue">{{ item.key }}: {{ item.value }}</p>
<p>{{  0.7589 | percent:'1.2-2' }}</p>
<p>{{ 123.456 | currency:'USD':'symbol':'1.2-2' }}</p>
```

Example2: Create custom pipe that should calculate cube of number(triple).

Step1: create pipe in angular using command ng g p triple.

Step2: Pipe will generate with the name of triple change code as follows.

```
@Pipe({
  name: 'triple'
})
export class TriplePipe implements PipeTransform {
  transform(value: number): number {
    return value * value * value;
  }
}
```

Step3: apply triple pipe in HTML template as follows.

```
<h1>{{ 5 | tripple }}</h1>
```

Example: create a custom pipe that truncate any string with specified value.

Pipe code:

```
import { Pipe, PipeTransform } from '@angular/core';

@Pipe({
  name: 'truncate'
})
export class TruncatePipe implements PipeTransform {

  transform(value: string, maxLength: number = 50): string {
    if (!value) return '';

    if (value.length <= maxLength) return value;

    return value.substring(0, maxLength) + '...';
  }
}
```

HTML code:

```
<h1>{{"abcdefghijklmnopqrstuvwxyz" | truncate : 21}}</h1>
```

Component Communication in Angular:

Component communication in Angular refers to the exchange of data between components. There are several ways to achieve component communication in Angular.

Input Binding(Parent to Child): This is the simplest form of communication, where data flows from a parent component to a child component. The parent component passes data to the child component through input properties. Input properties are declared in the child component using the @Input() decorator. Can become complex in deeply nested component hierarchies.

Output Binding with EventEmitter(Child to Parent): This allows child components to emit events to parent components. The child component can raise events using EventEmitter, and the parent component can listen to these events and respond accordingly. Child components declare output properties using EventEmitter and the @Output() decorator. They emit events using emit() method. Parent components bind to these events in the template using event binding syntax. Can become cumbersome in deeply nested component trees.

ViewChild and ContentChild(Child to Parent): These decorators allow parent components to access child components' properties or elements directly in the parent template. ViewChild is used to access child components, while ContentChild is used to access elements projected into a component using ng-content. May not be suitable for more complex communication needs. Can lead to tightly coupled parent-child relationships.

Communication By Template Variable(Child to Parent):

Component communication using template variables in Angular allows parent components to access child components or DOM elements directly within their templates. This method is useful for simple interactions between components or for accessing DOM elements within a template.

Services: Services are a way to share data and functionality across components in Angular. Components can inject services and use them to communicate with each other. Services act as a central point for managing shared data and communication between components. Services are singleton objects that are instantiated once and shared across components in an Angular application. They are used to encapsulate and share data and functionality across components. Components can inject services in their constructors and use them to communicate with each other. Services act as intermediaries for cross-component communication. Facilitates loosely coupled communication between components. Suitable for sharing data and functionality across multiple components. Requires extra setup for dependency injection.

Example1: communicate using input binding in angular.

Suppose we have a parent component ParentComponent and a child component ChildComponent. We want to pass a message from the parent component to the child component using input binding.

ChildComponent:

Let's start by creating the child component. We'll define an input property called message in the child component, which will receive the message from the parent component.

child HTML code:

```
<p>{{ message }}</p>
```

child TS code:

```
import { Component, Input } from '@angular/core';
@Component({
  selector: 'app-childcomponent',
  templateUrl: './childcomponent.component.html',
  styleUrl: './childcomponent.component.css'
})
export class ChildcomponentComponent {
  @Input() message: string="";
}
```

ParentComponent:

Next, we'll create the parent component. In the parent component's template, we'll use the child component and bind the message property to a variable containing the message we want to pass.

Parent HTML code:

```
<app-childcomponent [message]="parentMessage"></app-childcomponent>
```

parent TS code:

```
export class ParentComponent {
  parentMessage: string = 'This message used and displayed by child class';
}
```

Run Parent component to see the message which is from child component.

Explanation:

In the parent component's template, we use the child component

< app-childcomponent > and bind the message input property to the parentMessage variable using square brackets [message]="parentMessage". This is how input binding is done in Angular.

Now, when the parent component is rendered, it will pass the parentMessage to the child component, and the child component will display the message received from the parent.

Example2: Another example for communication using input binding in angular.

ChildComponent:

child HTML code:

```
<h1 *ngFor="let item of items">{{ item.name }} - {{ item.age }}</h1>
```

child TS code:

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

@Component({
  selector: 'app-childcomponent',
  templateUrl: './childcomponent.component.html',
  styleUrl: './childcomponent.component.css'
})
export class ChildcomponentComponent {

  @Input() items: { name: string, age: number }[]=[];

}
```

ParentComponent:

Next, we'll create the parent component. In the parent component's template, we'll use the child component and bind the message property to a variable containing the message we want to pass.

Parent HTML code:

```
<app-childcomponent [items]="parentItems"></app-childcomponent>
```

parent TS code:

```
export class ParentComponent {
  parentItems: { name: string, age: number }[] = [
    { name: 'Alice', age: 30 },
    { name: 'Bob', age: 35 },
    { name: 'Charlie', age: 40 }
  ];
}
```

Run Parent component to see the message which is from child component.

Example3: communicate child to parent using @output

Child HTML:

```
<button (click)="onButtonClick()">Click Me</button>
```

Child TS:

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

@Component({
  selector: 'app-childcomponent',
  templateUrl: './childcomponent.component.html',
  styleUrl: './childcomponent.component.css'
})
export class ChildcomponentComponent {
  @Output() buttonClicked = new EventEmitter<void>();

  onButtonClick() {
    this.buttonClicked.emit();
  }
}
```

Parent HTML:

```
<app-childcomponent (buttonClicked)="onButtonClicked()">
</app-childcomponent>
    <p *ngIf="buttonWasClicked">Button was clicked!</p>
```

Parent TS:

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

@Component({
  selector: 'app-parentcomponent',
  templateUrl: './parentcomponent.component.html',
  styleUrl: './parentcomponent.component.css'
})
export class ParentcomponentComponent {
  buttonWasClicked: boolean = false;

  onButtonClicked() {
    this.buttonWasClicked = true;
  }
}
```

Run Parent component to see the message which is from child component.

Example4: another example for communication using @output

Child HTML:

```
<button (click)="emitData()">Send Data</button>
```

Child TS:

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

@Component({
  selector: 'app-childcomponent',
  templateUrl: './childcomponent.component.html',
  styleUrl: './childcomponent.component.css'
})
export class ChildcomponentComponent {
  @Output() dataChanged = new EventEmitter<string>();
```

```
            emitData() {
              this.dataChanged.emit('Data from child');
            }
          }
```

Parent HTML:

```
      <app-childcomponent (dataChanged)="onDataChanged($event)">
    </app-childcomponent>
      <p>{{ modifiedData }}</p>
```

Parent TS:

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

@Component({
  selector: 'app-parentcomponent',
  templateUrl: './parentcomponent.component.html',
  styleUrl: './parentcomponent.component.css'
})
export class ParentcomponentComponent {
  modifiedData: string="";

  onDataChanged(data: string) {
    this.modifiedData = 'Modified: ' + data.toUpperCase();
  }
}
```

Example5: Communication with child class object using @ViewChild

Child Component:

HTML code:

```
<input type="text" [(ngModel)]="inputValue">
  <button (click)="clearInput()">Clear Input</button>
```

TS code:

```
import { Component } from '@angular/core';
@Component({
  selector: 'app-childcomponent',
  templateUrl: './childcomponent.component.html',
  styleUrl: './childcomponent.component.css'
})
export class ChildcomponentComponent {
  inputValue: string = '';
  clearInput() {
    this.inputValue = '';
  }
}
```

Parent Component:

HTML code:

```
<app-childcomponent></app-childcomponent>
    <button (click)="clearChildInput()">Clear Child Input</button>
```

TS code:

```
import { Component, ViewChild } from '@angular/core';
import { ChildcomponentComponent } from
'../childcomponent/childcomponent.component';

@Component({
  selector: 'app-parentcomponent',
  templateUrl: './parentcomponent.component.html',
  styleUrl: './parentcomponent.component.css'
})
export class ParentcomponentComponent {
  @ViewChild(ChildcomponentComponent) childComponent:
ChildcomponentComponent = new ChildcomponentComponent;

  clearChildInput() {
    this.childComponent.clearInput();
  }
}
```

Example6: Communication using template variable

Child Component:
HTML code:

```
<p>Child Component</p>
```

TS code:

```
export class ChildcomponentComponent {
  childMessage: string = 'Hello from child';
}
```

Parent Component:
HTML code:

```
<app-childcomponent #childRef></app-childcomponent>
<p>{{ childRef.childMessage }}</p>
```

TS code: Not required

Example7: Communication using Service

Step1: create service using ng g s service1

Step2: import that service1 into appModule and include it in providers.

Step3: import that service1 in which component required.

Step4: Create object for that service.

Step5: use that object to get required properties or methods of that service.

Service1 code:

```
export class Service1Service {
  cars = ["SUZUKI", "KIA", "TOYOTO", "NISSAN", "FORD"]
  constructor() { }
}
```

Component:

HTML code:

```
export class CompoComponent {
   ts = new Service1Service();
}
```

TS code:

```
<h1 *ngFor="let i of ts.cars">{{i}}</h1>
```

Example8: Create service and components and add some users to service.

Service:

```
export class Service2Service {

  private users: string[] = ['Alice', 'Bob', 'Charlie'];

  constructor() { }

  getUsers(): string[] {

    return this.users;

  }

  addUser(user: string): void {

    this.users.push(user);

  }
}
```

Component:

TS code:

```
export class CompoComponent {

    ts = new Service2Service();

    users:string[]= this.ts.getUsers();

     createUser() {

      const newUser = prompt('Enter a new user name:');

      if (newUser) {

        this.ts.addUser(newUser);

        this.users = this.ts.getUsers(); // Refresh user list

      }
    }
}
```

HTML code:

```
<h2>User List</h2>
    <ul>
      <li *ngFor="let user of users">{{ user }}</li>
    </ul>
    <button (click)="createUser()">Add User</button>
```