1. **Create a component using terminal, and use it in the root component and display it.**

```
// app.component.ts
import { Component } from '@angular/core';

@Component({
 selector: 'app-root',
 template: `
  <h1>Welcome to My App!</h1>
  <app-my-new-component></app-my-new-component>
 `
})
export class AppComponent {}
```

2. **Write a program to show usage of union types in typescript.**

```
// Define a union type for a variable that can be either a string or a number
type ID = number | string;

// Create a function that takes an ID and returns a message
function getIdMessage(id: ID): string {
   // Check the type of the id parameter
   if (typeof id === "string") {
      return `The ID is a string: ${id}`;
   } else {
      return `The ID is a number: ${id}`;
   }
}

// Example usage of the function with different types
const id1: ID = 12345; // number
const id2: ID = "user_67890"; // string

console.log(getIdMessage(id1)); // Output: The ID is a number: 12345
console.log(getIdMessage(id2)); // Output: The ID is a string: user_67890
```

3. **Make Parent child relation through component and when data is entered in parent component show it in the child component.**

Step 1: Create the Parent Component

// ParentComponent.js

```
import React, { useState } from 'react';
import ChildComponent from './ChildComponent';

const ParentComponent = () => {
  const [inputData, setInputData] = useState('');

  const handleInputChange = (event) => {
    setInputData(event.target.value);
  };

  return (
    <div>
      <h1>Parent Component</h1>
      <input
        type="text"
        value={inputData}
        onChange={handleInputChange}
        placeholder="Enter something..."
      />
      <ChildComponent data={inputData} />
    </div>
  );
};

export default ParentComponent;
```

Step 2: Create the Child Component

```
// ChildComponent.js
import React from 'react';

const ChildComponent = ({ data }) => {
  return (
    <div>
      <h2>Child Component</h2>
      <p>Data from Parent: {data}</p>
    </div>
  );
};
```

```
export default ChildComponent;
```

Step 3: Use the Parent Component in Your App

```
// App.js
import React from 'react';
import ParentComponent from './ParentComponent';

const App = () => {
  return (
    <div>
      <ParentComponent />
    </div>
  );
};

export default App;
```

4. **Show difference type and interface through example.**
   Example of Type.

```
// Define a type alias for a simple object
type Person = {
  name: string;
  age: number;
};

// You can also define a union type
type ID = string | number;

// Example of a function using the type
function greet(person: Person): string {
    return `Hello, my name is ${person.name} and I am ${person.age} years
    old.`;
}

// Using the type
const user: Person = {
  name: "Alice",
  age: 30
};
```

```
console.log(greet(user)); // Output: Hello, my name is Alice and I am 30 years old.
```

Example of Interface

```
// Define an interface for a simple object
interface Person {
  name: string;
  age: number;
}

// Extending interfaces
interface Employee extends Person {
  employeeId: number;
}

// Example of a function using the interface
function greet(employee: Employee): string {
    return `Hello, my name is ${employee.name}, I am ${employee.age} years old, and my employee ID is ${employee.employeeId}.`;
}

// Using the interface
const staff: Employee = {
  name: "Bob",
  age: 40,
  employeeId: 12345
};

console.log(greet(staff)); // Output: Hello, my name is Bob, I am 40 years old, and my employee ID is 12345.
```

5. **Use decorator on a class to show data in component.**

```
def display_data_decorator(func):
  def wrapper(self, *args, **kwargs):
    # Call the original method
    result = func(self, *args, **kwargs)

    # Display the data in a component-like style
    print(f"Component Data: {self.data}")
    return result
```

```
    return wrapper

class Component:
    def __init__(self, data):
        self.data = data

    @display_data_decorator
    def update_data(self, new_data):
        self.data = new_data
        print(f"Data updated to: {self.data}")

    @display_data_decorator
    def reset_data(self):
        self.data = None
        print("Data reset to None")

# Usage
component = Component("Initial Data")
component.update_data("Updated Data")
component.reset_data()
```

6. **Show data in html using interpolation through the component ts file.**

Interpolation in Angular

Component (example.component.ts)

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html'
})
export class ExampleComponent {
  title = 'Angular Interpolation Example';
  name = 'John Doe';
  age = 30;
  occupation = 'Software Developer';
}
```

Template (example.component.html)

```
<div>
  <h1>{{ title }}</h1>
  <p>Name: {{ name }}</p>
  <p>Age: {{ age }}</p>
  <p>Occupation: {{ occupation }}</p>
  <p>Full Info: {{ name }} is {{ age }} years old and works as {{ occupation }}</p>
</div>
```

7. **Make Parent child relation and get data in parent component.**

Parent-Child Relationship in Angular

Parent Component (parent.component.ts)

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

@Component({
  selector: 'app-parent',
  template: `
    <app-child (childEvent)="getParentData($event)"></app-child>
    <p>Received from Child: {{ childData }}</p>
  `
})
export class ParentComponent {
  childData = '';

  getParentData(data) {
    this.childData = data;
  }
}
```

Child Component (child.component.ts)

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

```
@Component({
  selector: 'app-child',
  template: `
    <button (click)="sendData()">Send Data to Parent</button>
  `
})
export class ChildComponent {
  @Output() childEvent = new EventEmitter();

  sendData() {
    this.childEvent.emit('Data from Child');
  }
}
```

8. **Write a Angular program to using ngif.**

Angular ngIf Directive

The ngIf directive is used to conditionally include or exclude a part of the HTML template.

Example Program

Component (example.component.ts)

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html'
})
export class ExampleComponent {
  isAdmin = true;
  isLoggedin = false;
}
```

Template (example.component.html)

```
<div>
```

```
 <h1>ngIf Example</h1>

 <!-- Simple ngIf -->
 <p *ngIf="isAdmin">You are an admin</p>

 <!-- ngIf with else block -->
 <div *ngIf="isLoggedin; else logout">
  You are logged in
 </div>
 <ng-template #logout>
  Please login
 </ng-template>

 <!-- ngIf with then and else blocks (Angular 4+) -->
 <div [ngIf]="isAdmin">
  <ng-template ngIfThen>
   You are an admin
  </ng-template>
  <ng-template ngIfElse>
   You are not an admin
  </ng-template>
 </div>
</div>
```

9. **what is the use of $event and make a simple angular program.**

   $event in Angular

   $event is a special variable in Angular that represents the event object passed to an event handler.

   Use of $event:

   - Access event properties (e.g., target, value, keyCode)
   - Prevent default event behavior (e.g., form submission)
   - Get event data (e.g., mouse coordinates, keyboard input)

   Simple Angular Program:

   Component (example.component.ts)

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

@Component({
  selector: 'app-example',
  templateUrl: './example.component.html'
})
export class ExampleComponent {
  eventName = '';
  eventValue = '';
}
```

Template (example.component.html)

```
<div>
  <h1>$event Example</h1>

  <!-- Click event -->
  <button (click)="handleClick($event)">Click me</button>

  <!-- Input event -->
  <input (input)="handleInput($event)" type="text" placeholder="Type
something">

  <p>Event Name: {{ eventName }}</p>
  <p>Event Value: {{ eventValue }}</p>
</div>
```

Component Methods

```
handleClick(event: MouseEvent) {
  this.eventName = 'Click';
  console.log('Mouse coordinates:', event.clientX, event.clientY);
}

handleInput(event: KeyboardEvent) {
  this.eventName = 'Input';
  this.eventValue = event.target.value;
```

```
  console.log('Key pressed:', event.key);
}
```

10. **Explain the Booting process of Angular.**

Angular Boot Process

The Angular boot process is the sequence of events that occurs when an Angular application is loaded. Here's an overview of the boot process:

Step 1: Index.html Loading

- The browser loads the index.html file, which serves as the entry point for the Angular application.
- The index.html file contains a <app-root> selector, which is the root component of the Angular application.

Step 2: Main.ts Loading

- The index.html file loads the main.ts file, which is the entry point for the Angular application.
- main.ts imports the AppModule and bootstraps it using the platformBrowserDynamic() function.

Step 3: AppModule Loading

- The AppModule is loaded, which imports other modules and components.
- The AppModule is decorated with the @NgModule decorator, which defines the module's metadata.

Step 4: Component Creation

- The AppModule creates the root component (AppComponent) and its child components.
- Components are instantiated and their templates are rendered.

Step 5: Dependency Injection

- Angular's dependency injection system resolves dependencies for components and services.

- Services are instantiated and provided to components.

Step 6: Change Detection

- Angular's change detection system checks for changes in component bindings.
- Changes are detected and updates are applied to the DOM.

Step 7: Rendering

- The Angular application is rendered to the DOM.
- The user interacts with the application.

Key Files and Functions

- index.html: Entry point for the Angular application.
- main.ts: Entry point for the Angular application; bootstraps the AppModule.
- app.module.ts: Defines the AppModule and its dependencies.
- app.component.ts: Defines the root component (AppComponent).
- platformBrowserDynamic(): Bootstraps the AppModule in the browser.

Best Practices

- Use the @NgModule decorator to define module metadata.
- Use dependency injection to provide services to components.
- Use change detection to optimize application performance.

Angular CLI Commands

- ng new: Creates a new Angular project.
- ng serve: Builds and serves the Angular application.
- ng build: Builds the Angular application for production.

By understanding the Angular boot process, developers can better optimize their applications for performance and scalability.