Interfaces:

Interface is a syntactical contract that defines the structure of an object. It specifies the properties and methods that an object should have, allowing for type-checking

why use interface?

1.Code Readability: They improve code clarity by defining clear contracts for what an object should look like, making it easier for developers to understand and maintain the code.

2.Modularity: Interfaces promote modular design by allowing different parts of the code to communicate through defined structures, facilitating easier updates and changes.

3.Multiple Implementations: A class can implement multiple interfaces, allowing for flexible designs and promoting code reuse.

4.Type Safety: Interfaces provide a way to enforce type checking, reducing runtime errors by ensuring that objects conform to expected structures.

Implements

Purpose: The `implements` keyword is used by a class to signify that it is adhering to a specific interface. This means the class must define all the properties and methods specified in the interface.

Example:

```
interface Animal {

name: string;

sound(): void;

}

class Dog implements Animal {

constructor(public name: string) {}

sound() {

console.log("Woof!");

}}
```

```
const dog = new Dog("Buddy");

dog.sound(); // Output: Woof!
```

Extends

Purpose: The `extends` keyword is used to create a new interface or class based on an existing one. When an interface extends another interface, it inherits all the properties and methods of the parent interface and can also define additional ones.

Example:

```
interface Animal {

name: string;

sound(): void;

}

interface Pet extends Animal {

owner: string;

}

class Cat implements Pet {

constructor(public name: string, public owner: string) {}



sound() {

console.log("Meow!");

}

}

const cat = new Cat("Whiskers", "Alice");

cat.sound(); // Output: Meow!

console.log(cat.owner); // Output: Alice
```

Conclusion:

Use `implements` when defining classes that need to adhere to specific contracts (interfaces).

Use `extends` when you want to create a new interface or class that builds upon the structure of an existing one, promoting reuse and hierarchy.

Interfaces example:

```
interface Person {
    name: string;
    age: number;
    greet(): void;
}
class Employee implements Person {
    constructor(public name: string, public age: number) {
    }
    greet() {
        console.log(`Hello, my name is ${this.name}`);
    }
}
const employee = new Employee("nithu",90);
employee.greet();
```

Interface using objects:

```
interface employee_details{
    empName: string;
    empId: number
```

```
empSalary: number;
}
let employee: employee_details={
    empName:"priyu",
    empId:2234,
    empSalary: 50000
};
console.log(employee);
```

Interface using functions

```
interface convertCase{
    (input: string, isUpper: boolean): string;
}
let format: convertCase;

format= function(input: string, isUpper: boolean): string{
    return isUpper? input.toUpperCase(): input.toLowerCase();
}
console.log(format("nithu",true));
```

Interface:

```
interface person{
    name: string;
    age: number;
    display(): void;
}
class employee implements person {
```

```
    name: string;
    age: number;
    constructor(name: string, age: number,  public id:number) {     //manual assignment
        this.name = name;
        this.age = age;
       //here ,we used public keyword for printing extra property without declare the property in
interface
    }
    /*
     constructor(public name: string, public age: number) {
        // Properties are automatically declared and initialized
    }
    */
    display() {
        console.log(`the name is ${this.name}`);
    }
}
const Employee = new employee("nithu", 90,2234);
Employee.display();
```

Extending Interface:

```
interface student_details{
    name: string;
    age: number;
}
interface student_educationalDetails extends student_details{
```

```
    class: string;
    rollNumber: string;
}
class student implements student_educationalDetails{
    name="nithya";
    age=20;
    class="10thstd";
    rollNumber: "21ucs009";
}
const Student = new student();
console.log(Student.class);
```

ReadOnly & Optional using interfaces:

```
interface Person {
    readonly id: number;  // This property is readonly, meaning it cannot be changed after being assigned
    name: string;         // Regular property, can be modified
    age?: number;         // This property is optional, meaning it may or may not be provided
}
class Employee implements Person {
    readonly id: number;
    name: string;
    age?: number; //optional
    constructor(id: number, name: string, age?: number) {
        this.id = id;
        this.name = name;
        this.age = age;
```

```
    }

  display(): void {
        console.log(`ID: ${this.id}, Name: ${this.name}, Age: ${this.age ?? 'not provided'} `);
    }}
const emp1 = new Employee(1, 'nithu', 30);
emp1.display();
const emp2 = new Employee(2, 'pandi');
emp2.display();
```

Real Time Example: E-Commerce Payment System

```
interface PaymentMethod {
    processPayment(amount: number): boolean;
    getTransactionId(): string;
}
class CreditCardPayment implements PaymentMethod {
    constructor(private cardNumber: string) {}
    processPayment(amount: number): boolean {
        console.log(`Processed credit card payment of $${amount}`);
        return true;
    }
    getTransactionId(): string {
        return "CC12345";
    }}
class PayPalPayment implements PaymentMethod {
```

```
    constructor(private email: string) {}
     processPayment(amount: number): boolean {
          console.log(`Processed PayPal payment of $${amount}`);
        return true;
    }
getTransactionId(): string {
        return "PP67890";
    }
}
function completePayment(paymentMethod: PaymentMethod, amount: number) {
    if (paymentMethod.processPayment(amount)) {
        console.log(`Payment successful! Transaction ID: ${paymentMethod.getTransactionId()}`);
    } else {
        console.log("Payment failed.");
    }
}
const creditCard = new CreditCardPayment("1234-5678-9012-3456");
const paypal = new PayPalPayment("user@example.com");
completePayment(creditCard, 100);
completePayment(paypal, 50);
```

Output:

```
Processed credit card payment of $100
Payment successful! Transaction ID: CC12345
Processed PayPal payment of $50
Payment successful! Transaction ID: PP6789
```

Abstract class

- Define an abstract class in Typescript using the `abstract` keyword.
- Abstract classes are mainly for inheritance where other classes may derive from them.
- We cannot create an instance of an abstract class.
- An abstract class typically includes one or more abstract methods or property declarations.
- The class which extends the abstract class must define all the abstract methods.

Why we use abstract class over interfaces?

Feature	Abstract Class	Interface
Implementation	Can contain implementation of methods (concrete methods).	Cannot contain any method implementation (only declarations).
Instantiation	Cannot be instantiated directly (like interfaces), but can contain constructors.	Cannot be instantiated directly and does not have constructors.
Member Variables	Can have member variables (fields) with default values or concrete implementations.	Cannot have fields, only the structure for an object (properties and methods).
Inheritance	Supports single inheritance (a class can only extend one abstract class).	Supports multiple inheritance (a class can implement multiple interfaces).
Access Modifiers	Can use `public`, `private`, `protected` for fields and methods.	All members are implicitly `public` (no access modifiers).
Usage	Use when you need to provide both method signatures and some shared implementation details.	Use when you only need to define a contract for classes without implementation.

example:

```
abstract class Animal {
    constructor(public name: string) {}
    abstract sound(): void; // Abstract method
    eat() {
        console.log(`${this.name} is eating.`);
    }}
class Dog extends Animal {
    sound() {
        console.log("Woof!");
    }}
class Cat extends Animal {
    sound() {
        console.log("Meow!");
}}
const dog = new Dog("Buddy");
const cat = new Cat("Whiskers");
dog.eat();
dog.sound();
cat.eat();
cat.sound();
```

Real time example: netflix

```
abstract class Content {
    constructor(public title: string, public releaseYear: number) {}

    abstract play(): void; // Abstract method

    info() {
        console.log(`${this.title} (${this.releaseYear})`);
    }
}
class Movie extends Content {
    play() {
        console.log(`Playing movie: ${this.title}`);
```

```
    }
}

class TVShow extends Content {
    play() {
        console.log(`Playing TV show: ${this.title}`);
    }
}

class Documentary extends Content {
    play() {
        console.log(`Playing documentary: ${this.title}`);
    }
}

const movie = new Movie("Inception", 2010);
const tvShow = new TVShow("Breaking Bad", 2008);
const documentary = new Documentary("Planet Earth", 2006);

movie.info();          // Output: Inception (2010)
tvShow.info();        // Output: Breaking Bad (2008)
documentary.info();   // Output: Planet Earth (2006)

movie.play();         // Output: Playing movie: Inception
tvShow.play();       // Output: Playing TV show: Breaking Bad
documentary.play(); // Output: Playing documentary: Planet Earth
```

Inheritance:

What is Inheritance?

Inheritance is a fundamental concept in object-oriented programming (OOP) that allows one class (the child or subclass) to inherit properties and methods from another class (the parent or superclass). This promotes code reusability and establishes a hierarchical relationship between classes.

Uses of Inheritance

1. Code Reusability: Inherit common functionality from a parent class, reducing redundancy.
2. Polymorphism: Enables methods to be overridden in child classes, allowing for dynamic method resolution.
3. Organization: Helps organize code into a clear hierarchy, making it easier to manage and understand.
4. Extensibility: Allows for easier updates and extensions of code without modifying existing classes.

When to Use Inheritance

- When you have multiple classes that share common behaviors and properties.
- When you want to create a new class that extends the functionality of an existing class.
- When you want to implement polymorphic behavior.

Example:

```
// Parent class
class Animal {
    constructor(public name: string) {}
    speak(): void {
        console.log(`${this.name} makes a noise.`);
    }
}
class Dog extends Animal {
    speak(): void {
        console.log(`${this.name} barks.`);
    }
}
class Cat extends Animal {
    speak(): void {
        console.log(`${this.name} meows.`);
    }
}

const dog = new Dog("Rex");
dog.speak(); // Output: Rex barks.

const cat = new Cat("Whiskers");
cat.speak(); // Output: Whiskers meows.
```

Real Time example: E-Commerce system:

```
// Base class for all products
class Product {
    constructor(public name: string, public price: number, public
description: string) {}

    displayInfo(): void {
        console.log(`Name: ${this.name}`);
        console.log(`Price: $${this.price}`);
        console.log(`Description: ${this.description}`);
    }
}

// Subclass for Electronics
class Electronics extends Product {
    constructor(name: string, price: number, description: string, public
warranty: number) {
        super(name, price, description);
    }

    displayInfo(): void {
        super.displayInfo();
        console.log(`Warranty: ${this.warranty} years`);
    }
}

// Subclass for Clothing
class Clothing extends Product {
    constructor(name: string, price: number, description: string, public
size: string) {
        super(name, price, description);
    }

    displayInfo(): void {
        super.displayInfo();
        console.log(`Size: ${this.size}`);
    }
}
```

```
// Subclass for Groceries
class Grocery extends Product {
    constructor(name: string, price: number, description: string, public
expiryDate: string) {
        super(name, price, description);
    }

    displayInfo(): void {
        super.displayInfo();
        console.log(`Expiry Date: ${this.expiryDate}`);
    }
}

// Usage
const laptop = new Electronics("Laptop", 999.99, "High-performance
laptop", 2);
laptop.displayInfo();

const tShirt = new Clothing("T-Shirt", 19.99, "100% cotton t-shirt", "M");
tShirt.displayInfo();

const apple = new Grocery("Apple", 0.99, "Fresh apple", "2024-10-20");
apple.displayInfo();
```

Classes:

Class

A **class** is a blueprint for creating objects. It encapsulates data (properties) and behavior (methods) related to a particular entity. Classes are a fundamental part of object-oriented programming in TypeScript.

Access Modifiers

Access modifiers control the visibility and accessibility of class members (properties and methods). TypeScript provides three access modifiers:

- **public**: Members are accessible from anywhere.
- **private**: Members are accessible only within the class they are declared in.

- **protected**: Members are accessible within the class they are declared in and in subclasses.

Getter

A **getter** is a method that allows you to access the value of a private or protected property. It is defined using the `get` keyword. Getters provide a way to read property values without directly accessing the properties.

Setter

A **setter** is a method that allows you to modify the value of a private or protected property. It is defined using the `set` keyword. Setters provide validation or transformation before setting a property.

Readonly

A **readonly** property can only be assigned during declaration or in the constructor. Once assigned, it cannot be changed. This is useful for defining constants or properties that should not be modified after initialization.

```
class Movie {
    public title: string;
    private _duration: number; // Duration in minutes

    constructor(title: string, duration: number) {
        this.title = title;
        this._duration = duration;
    }

    get duration(): number {
        return this._duration;
    }

    set duration(newDuration: number) {
        this._duration = Math.max(newDuration, 0); // Ensures duration is
non-negative
    }
}
```

```
class Show {
    public movie: Movie;
    public availableSeats: number;

    constructor(movie: Movie, availableSeats: number) {
        this.movie = movie;
        this.availableSeats = availableSeats;
    }

    public bookTicket(seats: number): boolean {
        if (seats > 0 && seats <= this.availableSeats) {
            this.availableSeats -= seats;
            console.log(`Successfully booked ${seats} seats for
${this.movie.title}.`);
            return true;
        } else {
            console.error('Not enough available seats.');
            return false;
        }
    }
}


// Usage
const movie = new Movie('Inception', 148);
const show = new Show(movie, 10);

console.log(`Movie: ${movie.title}, Duration: ${movie.duration} minutes`);
show.bookTicket(3); // Output: Successfully booked 3 seats for Inception.
console.log(`Available seats: ${show.availableSeats}`); // Output:
Available seats: 7

show.bookTicket(15); // Output: Successfully booked 7 seats for Inception.
console.log(`Available seats: ${show.availableSeats}`); // Output:
Available seats: 0
```

Output:

Movie: Inception, Duration: 148 minutes
Successfully booked 3 seats for Inception.
Available seats: 7
Available seats: 7
Not enough available seats.
Available seats: 7

Function:

1. Functions

A function is a reusable block of code that performs a specific task.

****Example:****

```
function greet(name: string): string {
    return `Hello, ${name}!`;
}

console.log(greet("Alice")); // Output: Hello, Alice!
```

2. Function Types

Function types specify the types of parameters and return values for functions.

Example:

```
type GreetFunction = (name: string) => string;

const greet: GreetFunction = (name) => {
    return `Hello, ${name}!`;
};
```

```
console.log(greet("Bob")); // Output: Hello, Bob!
```

3. Optional Parameters

Optional parameters are parameters that are not required when calling a function. They are defined using a question mark (`?`).

Example

```
function greetOptional(name: string, age?: number): string {
    if (age) {
        return `Hello, ${name}! You are ${age} years old.`;
    }
    return `Hello, ${name}!`;
}

console.log(greetOptional("Alice")); // Output: Hello, Alice!
console.log(greetOptional("Bob", 30)); // Output: Hello, Bob! You are 30 years old.
```

4. Default Parameters

Default parameters allow you to specify a default value for a parameter if no value is provided.

****Example:****

```
function greetWithDefault(name: string, greeting: string = "Hello"): string {
    return `${greeting}, ${name}!`;
}

console.log(greetWithDefault("Alice")); // Output: Hello, Alice!
console.log(greetWithDefault("Bob", "Hi")); // Output: Hi, Bob!
```

5. Rest Parameters

Rest parameters allow you to pass an arbitrary number of arguments to a function as an array.

Example:

```
function sum(...numbers: number[]): number {
    return numbers.reduce((acc, curr) => acc + curr, 0);
}

console.log(sum(1, 2, 3, 4)); // Output: 10
console.log(sum(5, 10)); // Output: 15
```

6. Function Overloading

Function overloading allows you to define multiple signatures for a function, enabling it to accept different types or numbers of parameters.

Example:

```
function combine(input: number, input2: number): number;
function combine(input: string, input2: string): string;
function combine(input: any, input2: any): any {
    return input + input2;
}

console.log(combine(1, 2)); // Output: 3
console.log(combine("Hello, ", "world!")); // Output: Hello, world!
```

Real time example: User profile management

```
// Define the UserProfile interface
interface UserProfile {
    name: string;
    age?: number; // Optional
    email: string;
}

// Function Type for creating user profiles
```

```typescript
type CreateProfileFunction = (profile: UserProfile) => string;

// Function to create a user profile
const createUserProfile: CreateProfileFunction = (profile) => {
    return `User Profile: ${profile.name}, Age: ${profile.age ?? "N/A"},
Email: ${profile.email}`;
};

// Function to create a user profile with an optional address
function createUserProfileWithAddress(profile: UserProfile, address?:
string): string {
    return `User Profile: ${profile.name}, Age: ${profile.age ?? "N/A"},
Email: ${profile.email}, Address: ${address ?? "N/A"}`;
}

// Function to create a user profile with a default age
function createUserProfileWithDefaultAge(profile: UserProfile, defaultAge:
number = 18): string {
    const age = profile.age !== undefined ? profile.age : defaultAge;
    return `User Profile: ${profile.name}, Age: ${age}, Email:
${profile.email}`;
}

// Function to add multiple emails to a user profile
function addEmailsToProfile(profile: UserProfile, ...emails: string[]):
string {
    const allEmails = emails.join(", ");
    return `User Profile: ${profile.name}, Age: ${profile.age ?? "N/A"},
Emails: ${allEmails}`;
}

// Function overloading to update user information
function updateProfile(name: string, age: number): string;
function updateProfile(email: string): string;
function updateProfile(input: any, age?: number): string {
    if (typeof input === "string" && typeof age === "number") {
        return `Updated: Name - ${input}, Age - ${age}`;
    } else if (typeof input === "string") {
        return `Updated: Email - ${input}`;
    }
```

```
    return "Invalid input";
}

// Usage Examples
console.log(createUserProfile({ name: "Alice", email: "alice@example.com"
}));
console.log(createUserProfileWithAddress({ name: "Bob", email:
"bob@example.com" }, "123 Main St"));
console.log(createUserProfileWithDefaultAge({ name: "Charlie", email:
"charlie@example.com" }));
console.log(addEmailsToProfile({ name: "Dana", age: 25, email:
"dana@example.com" }, "dana1@example.com", "dana2@example.com"));
console.log(updateProfile("Eve", 30));
console.log(updateProfile("eve@example.com"));
```

BASICS TS:

```
1. Type Annotation

Type annotations explicitly specify the type of a variable. This helps
with type checking and makes the code more readable.


let name: string = "Alice"; // 'name' is explicitly annotated as a string

2. Number
The `number` type represents both integer and floating-point numbers.


let age: number = 30; // 'age' is a number


3. String
The `string` type is used for textual data.


let greeting: string = "Hello, world!"; // 'greeting' is a string
```

4. Boolean
The `boolean` type represents a true or false value.

```
let isActive: boolean = true; // 'isActive' is a boolean
```

5. Object Type
An object type can have properties with specified types. This allows for structured data.

```
let person: { name: string; age: number } = {
    name: "Alice",
    age: 30
}; // 'person' is an object with specific properties
```

6. Array
The `array` type can hold multiple values of the same type.

```
let scores: number[] = [85, 90, 78]; // 'scores' is an array of numbers
```

7. Tuple
A tuple is a fixed-size array with specified types for each element.

```
let coordinates: [number, number] = [10, 20]; // 'coordinates' is a tuple of two numbers
```

8. Enum
Enums are a way to define named constants, making code easier to read and maintain.

```
enum Color {
```

```
    Red,
    Green,
    Blue
}
let favoriteColor: Color = Color.Green; // 'favoriteColor' uses the Color
enum


9. Any Type
The `any` type allows a variable to hold values of any type, providing
flexibility.


let randomValue: any = "Could be anything";
randomValue = 42; // 'randomValue' can change type freely

10. Void Type
The `void` type indicates that a function does not return a value.

function logMessage(message: string): void {
    console.log(message);
} // This function does not return anything


11. Never Type

The `never` type represents values that never occur, typically used for
functions that throw errors.


function throwError(message: string): never {
    throw new Error(message);
} // This function never completes normally


12. Union Types
Union types allow a variable to hold multiple types.


let id: string | number = "123"; // 'id' can be a string or a number
```

```
id = 456; // Now it's a number


13. Type Aliases
Type aliases create a new name for a type, making it easier to use complex
types.


type StringOrNumber = string | number;
let value: StringOrNumber = "Hello"; // 'value' can be a string or a
number

14. String Literal Types
String literal types restrict a variable to specific string values.


type Direction = "left" | "right" | "up" | "down";
let move: Direction = "left"; // 'move' can only be one of the specified
strings


15. Type Inference
Type inference allows TypeScript to automatically deduce the type of a
variable based on its initial value.


let inferredString = "This is inferred"; // Type is automatically inferred
as string
```