**Date: 24/06/2025--- 25/06/2025  
Training section topics: TYPESCRIPT, Functions in JavaScript, Scope, ES6(2015)Features, Modules, Namespace , object oriented in typescript, Advanced features typescript**

**1. TYPSCRIPT**

TypeScript is a **superset of JavaScript** designed to add static typing and other features that enhance JavaScript for larger, more complex projects.

.How to Start Using TypeScript

 **Install TypeScript Compiler**:

npm install -g typescript

 **Write Code in .ts Files**: Create a TypeScript file (e.g., example.ts).

 **Compile to JavaScript**:

tsc example.ts

This produces a example.js file that can run in any JavaScript environment.

### Advantages of TypeScript

1. **Better Code Quality**: Type checking catches many bugs before runtime.
2. **Improved Collaboration**: Explicit types make the code easier for teams to understand.
3. **Easier Refactoring**: Tooling helps prevent errors during code changes.
4. **Optional Types**: Developers can opt-in to static typing as needed.

When creating a variable, there are two main ways in typescript

1. Explicit
2. Implicit

**Explicit**->

Let firstname : string=”Anagha”;

Console.log(typeof firstname); // output will be string

**Implicit**-> typescript will guess the type based on the assigned value;

Let firstname =”Anagha”;

Console.log(typeof firstname); // output will be string

(In JavaScript specifies values only in explicit

Implicit-> global without let,cont,or var

Eg: age =25;)

# TypeScript Special Types

## Type: any

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

The example below does not use any and will throw an error:

### Example without any

let u = true;  
u = "string"; // Error: Type 'string' is not assignable to type 'boolean'.  
Math.round(u); // Error: Argument of type 'boolean' is not assignable to parameter of type 'number'.

Example with any

let 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, as shown in the below example:

let w: unknown = 1;  
w = "string"; // no error  
w = {  
  runANonExistentMethod: () => {  
    console.log("I think therefore I am");  
  }  
} as { runANonExistentMethod: () => void}  
// How can we avoid the error for the code commented out below when we don't know the type?  
// w.runANonExistentMethod(); // Error: Object is of type 'unknown'.  
if(typeof w === 'object' && w !== null) {  
  (w as { runANonExistentMethod: Function }).runANonExistentMethod();  
}

## Type: never

Never effectively throws an error whenever it is defined.

let x: never = true; // Error: Type 'boolean' is not assignable to type 'never'.

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

# TypeScript Tuples

A **tuple** is a typed [array](https://www.w3schools.com/js/js_arrays.asp) 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.

Fixed-length arrays with specific types for each element.

Eg: let ourTuple: [number, boolean, string];

ourTuple = [5, false, 'Coding God was here']

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

# TypeScript Enums

An **enum** is a special "class" that represents a group of constants (unchangeable variables).

enum StatusCodes {  
  NotFound = 404,  
  Success = 200,  
  Accepted = 202,  
  BadRequest = 400  
}

console.log(StatusCodes.NotFound); // output will be 404  
console.log(StatusCodes.Success); // output will be 200

### ****Access Modifiers****

* Control visibility of class members using public, private, or protected.
* Example:

class Person {

private name: string;

constructor(name: string) {

this.name = name;

}

}

let person = new Person("Alice");

console.log(person.name); //

### ****Type Guards****

* Narrow types at runtime using conditionals.
* Example:

typescript

CopyEdit

function print(value: number | string) {

if (typeof value === "string") {

console.log(value.toUpperCase());

}

}

### ****Union and Intersection Types****

* Combine types to create flexible structures.
* Example:

typescript

CopyEdit

let value: string | number = "hello";

### readonly ****Modifier****

* Prevent variables or properties from being reassigned.
* Example:

let readonlyArray: readonly number[] = [1, 2, 3];

readonlyArray[0] = 10; // Error: Cannot assign to '0' because it is a read-only property.

### ****Module System****

* TypeScript supports ES Modules and CommonJS, improving code modularity and scalability.
* Example:

export function greet() {

console.log("Hello");

}

import { greet } from "./module";

greet();

### ****Example: JavaScript vs TypeScript****

**JavaScript**:

function add(a, b) {

return a + b;

}

console.log(add(5, '10')); // Output: '510' (string concatenation)

**TypeScript**:

function add(a: number, b: number): number {

return a + b;

}

// console.log(add(5, '10')); // Error: Argument of type 'string' is not assignable to parameter of type 'number'.

console.log(add(5, 10)); // Output: 15

TypeScript catches type mismatches during development, whereas JavaScript would only reveal the issue at runtime.

**Differences Between TypeScript and JavaScript**

| **Aspect** | **JavaScript** | **TypeScript** |
| --- | --- | --- |
| **Typing** | Dynamically typed: Types are inferred at runtime. | Statically typed: Types must be explicitly defined or inferred at compile time. |
| **Compilation** | Interpreted by browsers or Node.js directly. | Compiles (transpiles) to JavaScript, which is then executed. |
| **Error Detection** | Errors are detected only at runtime. | Many errors can be caught during development (at compile time). |
| **Code Scalability** | Lacks features to manage complex projects, which can lead to maintenance issues in large codebases. | Provides features like interfaces, enums, and type annotations, making it suitable for large-scale projects. |
| **Tooling Support** | IDE support is available but less robust. | Strong IDE support, including autocompletion, refactoring, and real-time type checking. |
| **Features** | Supports basic JavaScript features. | Adds features like interfaces, generics, access modifiers, and decorators. |
| **Community** | Older and more widely used; native to browsers. | Growing popularity but requires additional setup. |
|  |  |  |

**Summary of Benefits**

| **Feature** | **TypeScript** | **JavaScript** |
| --- | --- | --- |
| **Type Safety** | Yes | No |
| **Interfaces** | Yes | No |
| **Generics** | Yes | No |
| **Enum Types** | Yes | No |
| **Compile-Time Checking** | Yes | No |
| **Access Modifiers** | Yes | No |
| **Tooling and IDE Support** | Strong | Moderate |

TypeScript's features make it especially valuable for large, complex, or team-based projects.

**2.Functions in javascript**

A **function** is a reusable block of code designed to perform a specific task. It can take inputs (parameters), process them, and return a result. There are several ways to define functions in JavaScript:

A **function** is a set of instructions grouped together to perform a specific task. Functions are building blocks of JavaScript programs and are reusable.

### ****Why Use Functions?****

1. **Reusability**: Write once, use multiple times.
2. **Modularity**: Break code into smaller, manageable pieces.
3. **Maintainability**: Easier to debug and maintain.
4. **Encapsulation**: Hide details and expose only necessary parts.

### ****Types of Functions****

#### ****1. Function Declaration****

* Declares a function with a name.
* Can be called before it's defined (hoisting applies).

function add(a, b) {

return a + b;

}

console.log(add(2, 3)); // Output: 5

#### ****2. Function Expression****

* Defines a function and assigns it to a variable.
* Not hoisted, so it must be defined before use.

const multiply = function(a, b) {

return a \* b;

};

console.log(multiply(4, 5)); // Output: 20

#### ****3. Arrow Functions****

* Introduced in ES6.
* Shorter syntax, especially useful for one-liners.
* Does not bind its own this, arguments, or super.

const divide = (a, b) => a / b;

console.log(divide(10, 2)); // Output: 5

#### ****4. Anonymous Functions****

* Functions without names, often passed as arguments.

setTimeout(function() {

console.log("Executed after 1 second");

}, 1000);

#### ****5. Immediately Invoked Function Expression (IIFE)****

* Executes immediately upon definition.
* Commonly used to create private scopes.

(function() {

let privateVar = "I am private";

console.log(privateVar);

})();

### ****Function Parameters and Arguments****

* **Parameters**: Variables listed in the function's definition.
* **Arguments**: Values passed when invoking a function.

javascript

CopyEdit

function greet(name, age) {

return `Hello ${name}, you are ${age} years old.`;

}

console.log(greet("Anagha", 25)); // Output: Hello Anagha, you are 25 years old.

#### Default Parameters (ES6)

javascript

CopyEdit

function greet(name = "Guest") {

return `Hello, ${name}`;

}

console.log(greet()); // Output: Hello, Guest

#### Rest Parameters (ES6)

* Captures multiple arguments into an array.
* function sum(...numbers) {

return numbers.reduce((total, num) => total + num, 0);

}

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

**3.Scope in javascript**

In JavaScript, scope defines the accessibility and visibility of variables, functions, and objects within different parts of your code. It determines where these elements can be accessed and modified.

Scope determines the accessibility (visibility) of variables.

JavaScript variables have 3 types of scope:

* Block scope
* Function scope
* Global scope

Before ES6 (2015), JavaScript variables had only **Global Scope** and **Function Scope**.

ES6 introduced two important new JavaScript keywords: let and const.

These two keywords provide **Block Scope** in JavaScript.

**Global Scope:**

* Variables declared outside of any function or block reside in the global scope.
* They are accessible from anywhere in the entire program, including within functions and blocks.
* Using too many global variables can lead to potential naming conflicts and make code harder to maintain
* Variables declared outside any function or block are global.
* Accessible everywhere in the code.

var globalVar = "I am global";

function showGlobal() {

console.log(globalVar); // Accessible

}

showGlobal();

console.log(globalVar); // Accessible

**Function Scope (Local Scope):**

Variables declared inside a function (using var, let, or const) have function scope.

They are only accessible within that specific function and its nested functions.

* + Variables with the same name can be used in different functions without causing conflicts.

function funcScope() {

if (true) {

var funcScopedVar = "I am function-scoped";

}

console.log(funcScopedVar); // Accessible

}

funcScope();

* **Block Scope:**
  + Introduced with let and const in ES6 (ECMAScript 2015).
  + Variables declared with let or const inside a block (e.g., if statements, for loops, while loops, or any curly braces {}) are only accessible within that block.
  + This helps in creating more contained and predictable code, preventing unintended variable access outside their intended scope.

Variables declared with let or const inside a block are confined to that block.

if (true) {

let blockLet = "Block scoped with let";

const blockConst = "Block scoped with const";

console.log(blockLet); // Accessible

console.log(blockConst); // Accessible

}

// console.log(blockLet); // Error: blockLet is not defined

// console.log(blockConst); // Error: blockConst is not defined

# 4. Javascript 2015 (ES6)

ECMAScript 2015 (commonly known as ES6) was a major update to JavaScript, introducing many new features and syntax improvements. It laid the foundation for modern JavaScript.

### ****1. Block Scope with**** let ****and**** const

* let and const introduced block-level scoping, replacing var for many use cases.
* **let**: Mutable variables with block scope.
* **const**: Immutable variables (constant references).

javascript

CopyEdit

let x = 10;

const y = 20;

if (true) {

let z = 30;

console.log(z); // 30

}

### ****2. Arrow Functions****

* A concise syntax for defining functions. Automatically binds this to the enclosing context
* . Arrow functions allows a short syntax for writing function expressions.
* You don't need the function keyword, the return keyword, and the **curly brackets**.
* You can only omit the return keyword and the curly brackets if the function is a single statement.

const add = (a, b) => a + b;

console.log(add(2, 3)); // 5

## 3.Object Destructuring

Extract values from arrays or objects into variables.

const [a, b] = [1, 2];

console.log(a, b); // 1, 2

// Object Destructuring

const { name, age } = { name: "Alice", age: 25 };

console.log(name, age); // Alice, 25

## 4.Array Destructuring

Destructuring assignment makes it easy to assign array values and object properties to variables.

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

// Destructuring

let [fruit1, fruit2] = fruits;

// Display Primitive Data

document.getElementById ("demo").innerHTML = fruit1 + ", " + fruit2; //Banana, Orange

## 5.The For/Of Loop

The JavaScript for/of statement loops through the values of iterable objects.

for/of lets you loop over data structures that are iterable such as Arrays, Strings, Maps, NodeLists, and more.

const arr = [10, 20, 30];

for (const num of arr) {

console.log(num);

}

## 6.JavaScript Maps

A Map holds key-value pairs where the keys can be any datatype.

A Map remembers the original insertion order of the keys.

You can create a Map by passing an Array to the new Map() constructor:

const map = new Map();

map.set("name", "Alice");

console.log(map.get("name"));

## 7.JavaScript Sets

A JavaScript Set is a collection of unique values.

Each value can only occur once in a Set.

The values can be of any type, primitive values or objects.

const set = new Set([1, 2, 2, 3]);

console.log(set); // Set(3) {1, 2, 3}

## 8.Array entries()

Create an Array Iterator, and then iterate over the key/value pairs:

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

const f = fruits.entries();

for (let x of f) {

document.getElementById("demo").innerHTML += x + "<br>";

}

The entries() method returns an Array Iterator object with key/value pairs:

[0, "Banana"]  
[1, "Orange"]  
[2, "Apple"]  
[3, "Mango"]

The entries() method does not change the original array.

## 9.Array.from()

The Array.from() method returns an Array object from any object with a length property or any iterable object.

let text = "ABCDEFG";

const myArr = Array.from(text);

document.getElementById("demo").innerHTML = myArr; // output will be A,B,C,D,E,F,G

## 10.Array keys ()

The keys() method returns an Array Iterator object with the keys of an array.

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

const keys = fruits.keys();

let text = "";

for (let x of keys) {

text += x + "<br>";

}

document.getElementById("demo").innerHTML = text;

// output will be

Return an Array Iterator object with the keys of the array:

0  
1  
2  
3

## 11. Array find ()

The find () method returns the value of the first array element that passes a test function.

This example finds (returns the value of) the first element that is larger than 18:

const numbers = [4, 9, 16, 25, 29];

let first = numbers.find(myFunction);

document.getElementById("demo").innerHTML = "First number over 18 is " + first;

function myFunction(value, index, array) {

return value > 18;} // output is 25

## 12.Array findIndex()

The findIndex() method returns the index of the first array element that passes a test function.

This example finds the index of the first element that is larger than 18:

const numbers = [4, 9, 16, 25, 29];

document.getElementById("demo").innerHTML = "First number over 18 has index " + numbers.findIndex(myFunction);

function myFunction(value, index, array) {

return value > 18;

} // output will be 3

**13.Array.every ()**

 Tests whether **all elements** in an array pass the provided function (predicate).

 Returns true if **every element** passes the test; otherwise, returns false.

const ages = [32, 33, 16, 40];

// Function to Run for every Element

function checkAge(age) {

return age > 18;

} // false

# 14.Array some()

 Tests whether **at least one element** in an array passes the provided function (predicate).

 Returns true if **any element** passes the test; otherwise, returns false.

<script>

const ages = [3, 10, 18, 20];

document.getElementById("demo").innerHTML = ages.some(checkAdult);

function checkAdult(age) {

return age > 18;

} // True

# 15.Array forEach()

The forEach method in JavaScript is used to iterate over elements in an array and execute a provided function once for each array element.

let text = "";

const fruits = ["apple", "orange", "cherry"];

fruits.forEach(myFunction);

document.getElementById("demo").innerHTML = text;

function myFunction(item, index) {

text += index + ": " + item + "<br>";

} //

0: apple  
1: orange  
2: cherry

# 5.Modules in typescript

Modules are files or code blocks that can export or import members like classes, functions, objects, or variables. Modules in TypeScript follow the **ECMAScript module** standard.  
In TypeScript, a module is a separate file that contains code and declarations that can be imported and used in other files. A module can be a single class, function, interface, variable, or a combination of these entities. Modules are defined using the export keyword.

### Features of Modules:

* Use export and import keywords.
* Each file is a module by default in TypeScript.
* Encourages better code reuse and separation of concerns.

### Example of Modules

#### File: mathUtils.ts

export function add(a: number, b: number): number {

return a + b;

}

export function subtract(a: number, b: number): number {

return a - b;

}

#### File: app.ts

import { add, subtract } from './mathUtils';

console.log(add(10, 5)); // Output: 15

console.log(subtract(10, 5)); // Output: 5

#### Steps to Run:

1. Compile both files:

tsc mathUtils.ts app.ts /// compilation

1. Run the app.js file using Node.js:

node app.js

output will be // 15 5

# 6.Namespace in typescript

A **namespace** in TypeScript is a way to logically group related code together. Think of it as a container for your classes, functions, or interfaces to avoid name conflicts.

* **When to use:** In projects where you’re not using modern module systems (import/export), like when including TypeScript directly in an HTML file with <script> tags.

Namespaces are a way to group related code together under a single namespace. They are typically used in projects that don't rely on a module system, such as simple web applications using a <script> tag.

**Features of Namespaces:**

* Use the namespace keyword.
* Does not use export and import like modules but can use the export keyword inside a namespace to make members accessible outside.
* More commonly used in legacy systems.

**Why Use a Namespace?**

Imagine you have two functions or classes with the same name. Without namespaces, you'd have conflicts. With namespaces, you can organize your code so that each function/class is scoped within its own namespace.

### Example of Namespaces

#### Single File Namespace

namespace MathUtils {

export function add(a: number, b: number): number {

return a + b;

}

export function subtract(a: number, b: number): number {

return a - b;

}

}

console.log(MathUtils.add(10, 5)); // Output: 15

console.log(MathUtils.subtract(10, 5)); // Output: 5

#### Multi-file Namespace

##### File: mathUtils.ts

namespace MathUtils {

export function add(a: number, b: number): number {

return a + b;

}

}

##### File: app.ts

/// <reference path="mathUtils.ts" />

console.log(MathUtils.add(10, 5)); // Output: 15

#### Steps to Run:

1. Compile with --outFile to generate a single JavaScript file:

tsc --outFile app.js app.ts

1. Run the app.js file:

node app.js

### Example 1: Simple Namespace

namespace Calculator {

export function add(a: number, b: number): number {

return a + b;

}

export function subtract(a: number, b: number): number {

return a - b;

}

}

// Access functions in the namespace

console.log(Calculator.add(5, 3)); // Output: 8

console.log(Calculator.subtract(5, 3)); // Output: 2

**Explanation**:

* The Calculator namespace groups the add and subtract functions.
* The export keyword makes these functions accessible outside the namespace.

### Example 2: Namespace with Classes

namespace Shapes {

export class Circle {

constructor(public radius: number) {}

area(): number {

return Math.PI \* this.radius \* this.radius;

}

}

export class Rectangle {

constructor(public width: number, public height: number) {}

area(): number {

return this.width \* this.height;

}

}

}

// Using the namespace

const circle = new Shapes.Circle(5);

console.log(`Circle area: ${circle.area()}`); // Output: Circle area: 78.53981633974483

const rectangle = new Shapes.Rectangle(10, 5);

console.log(`Rectangle area: ${rectangle.area()}`); // Output: Rectangle area: 50

**Explanation**:

* The Shapes namespace organizes classes related to shapes (Circle and Rectangle).
* You can access these classes using the Shapes namespace.

### Example 3: Multiple Files with Namespaces

#### File: mathUtils.ts

namespace MathUtils {

export function multiply(a: number, b: number): number {

return a \* b;

}

}

#### File: app.ts

/// <reference path="mathUtils.ts" />

console.log(MathUtils.multiply(4, 5)); // Output: 20

#### Steps to Compile and Run:

1. Compile both files together:

tsc --outFile app.js app.ts

1. Run the output file:

node app.js

**Explanation**:

* The /// <reference path="..."> directive connects the files.
* Both files' content is combined into a single JavaScript file.

## ****Modules vs. Namespaces****

| **Feature** | **Modules** | **Namespaces** |
| --- | --- | --- |
| **Definition** | File-based code organization. | Logical grouping of code. |
| **Syntax** | Uses export and import. | Uses namespace. |
| **Scoping** | Scopes to the file. | Scoped within the namespace. |
| **Use Case** | Modern projects using Node.js or ES6 modules. | Older or legacy projects. |
| **Compilation** | Supports import/export. | Requires --outFile flag. |

**7. Object-Oriented TypeScript**

**Key Concepts in Object-Oriented TypeScript**

1. **Classes**
2. **Interfaces**
3. **Inheritance**
4. **Access Modifiers**
5. **Abstract Classes**
6. **Static Members**
7. **Encapsulation**
8. **Polymorphism**

### ****1. Classes****

A class is a blueprint for creating objects, providing initial values for state (fields) and implementations of behavior (methods).

#### Example:

class Person {

name: string;

age: number;

constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

greet(): string {

return `Hi, my name is ${this.name}, and I am ${this.age} years old.`;

}

}

const person = new Person("John", 25);

console.log(person.greet()); // Output: Hi, my name is John, and I am 25 years old.

### ****2. Interfaces****

Interfaces define the structure of an object, ensuring that objects adhere to a specific shape.

#### Example:

interface Animal {

name: string;

sound(): string;

}

class Dog implements Animal {

name: string;

constructor(name: string) {

this.name = name;

}

sound(): string {

return "Woof!";

}

}

const dog = new Dog("Buddy");

console.log(`${dog.name} says ${dog.sound()}`); // Output: Buddy says Woof!

### ****3. Inheritance****

Classes can extend other classes to inherit properties and methods.

#### Example:

class Vehicle {

move(): string {

return "The vehicle is moving.";

}

}

class Car extends Vehicle {

move(): string {

return "The car is driving.";

}

}

const car = new Car();

console.log(car.move()); // Output: The car is driving.

### ****4. Access Modifiers****

Access modifiers control the visibility of class members.

* **public**: Accessible everywhere (default).
* **private**: Accessible only within the class.
* **protected**: Accessible within the class and its subclasses.

#### Example:

class BankAccount {

public accountNumber: number;

private balance: number;

constructor(accountNumber: number, balance: number) {

this.accountNumber = accountNumber;

this.balance = balance;

}

deposit(amount: number): void {

this.balance += amount;

}

getBalance(): number {

return this.balance;

}

}

const account = new BankAccount(12345, 1000);

account.deposit(500);

console.log(account.getBalance()); // Output: 1500

// console.log(account.balance); // Error: Property 'balance' is private

### ****5. Abstract Classes****

Abstract classes define a base class that cannot be instantiated and must be extended.

#### Example:

abstract class Animal {

abstract makeSound(): string;

move(): string {

return "The animal is moving.";

}

}

class Cat extends Animal {

makeSound(): string {

return "Meow!";

}

}

const cat = new Cat();

console.log(cat.makeSound()); // Output: Meow!

console.log(cat.move()); // Output: The animal is moving.

### ****6. Static Members****

Static members belong to the class, not an instance.

#### Example:

class Calculator {

static add(a: number, b: number): number {

return a + b;

}

}

console.log(Calculator.add(5, 10)); // Output: 15

### ****7. Encapsulation****

Encapsulation restricts access to some of an object's components to maintain integrity.

#### Example:

class Rectangle {

private width: number;

private height: number;

constructor(width: number, height: number) {

this.width = width;

this.height = height;

}

getArea(): number {

return this.width \* this.height;

}

}

const rectangle = new Rectangle(5, 10);

console.log(rectangle.getArea()); // Output: 50

### ****8. Polymorphism****

Polymorphism allows methods to behave differently based on the object calling them.

#### Example:

class Shape {

area(): number {

return 0;

}

}

class Circle extends Shape {

radius: number;

constructor(radius: number) {

super();

this.radius = radius;

}

area(): number {

return Math.PI \* this.radius \* this.radius;

}

}

class Square extends Shape {

side: number;

constructor(side: number) {

super();

this.side = side;

}

area(): number {

return this.side \* this.side;

}

}

const shapes: Shape[] = [new Circle(5), new Square(4)];

shapes.forEach(shape => console.log(shape.area()));

// Output:

// 78.53981633974483 (Circle area)

// 16 (Square area)

**Summary of Object-Oriented Features in TypeScript**

| **Feature** | **Description** | **Example** |
| --- | --- | --- |
| **Class** | Blueprint for creating objects. | class MyClass {...} |
| **Interface** | Defines a contract for objects. | interface MyInterface {...} |
| **Inheritance** | Enables a class to inherit from another class. | class Child extends Parent {...} |
| **Access Modifiers** | Restricts visibility of class members. | public, private, protected |
| **Abstract Classes** | Base class for other classes. Can't be instantiated. | abstract class |
| **Static Members** | Belong to the class, not instances. | static myMethod() |
| **Encapsulation** | Restricts access to class properties. | Use private |
| **Polymorphism** | Allows different behaviors through a common interface. | Overriding methods |

**8.Advanced typescript features**

### ****1. Union and Intersection Types****

#### ****Union Types****

A union type allows a value to be one of several types. It's useful when a variable could hold different types of values.

**Example:**

function printId(id: string | number): void {

if (typeof id === "string") {

console.log(`ID as string: ${id.toUpperCase()}`);

} else {

console.log(`ID as number: ${id.toFixed(2)}`);

}

}

printId(123.45); // Output: ID as number: 123.45

printId("abc123"); // Output: ID as string: ABC123

#### ****Intersection Types****

An intersection type combines multiple types into one. It represents a type that satisfies all the combined types.

**Example:**

interface User {

name: string;

}

interface Admin {

permissions: string[];

}

type AdminUser = User & Admin;

const admin: AdminUser = { name: "Alice", permissions: ["read", "write"] };

console.log(admin);

### ****2. Type Aliases****

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

**Example:**

type Point = { x: number; y: number };

const p1: Point = { x: 10, y: 20 };

type ID = string | number;

const userId: ID = 123; // Valid

### ****3. Literal Types****

Literal types let you specify exact values a variable can have, restricting it to predefined options.

**Example:**

type Direction = "up" | "down" | "left" | "right";

function move(direction: Direction): void {

console.log(`Moving ${direction}`);

}

move("up"); // Valid

// move("forward"); // Error: Argument not assignable to 'Direction'

### ****4. Mapped Types****

Mapped types transform existing types into new ones. You can apply transformations to all properties of a type.

**Example:**

type Readonly<T> = {

readonly [K in keyof T]: T[K];

};

interface User {

name: string;

age: number;

}

const readonlyUser: Readonly<User> = { name: "John", age: 30 };

// readonlyUser.age = 31; // Error: Cannot assign to 'age' because it is a read-only property

### ****5. Conditional Types****

Conditional types enable type logic, where the type is determined based on a condition.

**Example:**

type IsString<T> = T extends string ? "Yes" : "No";

type Test1 = IsString<string>; // "Yes"

type Test2 = IsString<number>; // "No"

### ****6. Utility Types****

TypeScript provides built-in utility types to simplify type transformations.

#### ****Partial****

Makes all properties in a type optional.

**Example:**

interface User {

name: string;

age: number;

}

const updateUser: Partial<User> = { name: "John" }; // Only 'name' is updated

#### ****Pick****

Creates a type by picking specific properties from another type.

**Example:**

type NameOnly = Pick<User, "name">;

const userName: NameOnly = { name: "Alice" };

#### ****Omit****

Creates a type by omitting specific properties from another type.

**Example:**

type WithoutAge = Omit<User, "age">;

const userWithoutAge: WithoutAge = { name: "Alice" };

### ****7. Template Literal Types****

Template literal types allow constructing string-based types dynamically.

**Example:**

type HttpMethod = "GET" | "POST" | "PUT" | "DELETE";

type ApiEndpoint = `/api/${HttpMethod}`;

const endpoint: ApiEndpoint = "/api/GET"; // Valid

// const invalidEndpoint: ApiEndpoint = "/api/SEARCH"; // Error

### ****8. Indexed Access Types****

Access the type of a specific property in an object.

**Example:**

interface User {

name: string;

age: number;

}

type NameType = User["name"]; // string

### ****9. Keyof Operator****

The keyof operator retrieves the keys of a type as a union of string literals.

**Example:**

type UserKeys = keyof User; // "name" | "age"

function getValue<T, K extends keyof T>(obj: T, key: K): T[K] {

return obj[key];

}

const user: User = { name: "Alice", age: 30 };

console.log(getValue(user, "name")); // Output: Alice

### ****10. Advanced Generics****

Generics allow creating reusable and flexible type-safe code.

#### ****Generic Functions****

**Example:**

function identity<T>(value: T): T {

return value;

}

console.log(identity<string>("Hello")); // Output: Hello

console.log(identity<number>(42)); // Output: 42

#### ****Generic Constraints****

Restrict the kinds of types that can be used with generics.

**Example:**

function logLength<T extends { length: number }>(item: T): void {

console.log(item.length);

}

logLength("Hello"); // Output: 5

logLength([1, 2, 3]); // Output: 3

// logLength(42); // Error: Type 'number' does not satisfy the constraint '{ length: number }'.

### ****11. Discriminated Unions****

A discriminated union is a union type that includes a common property (discriminator) for type narrowing.

**Example:**

interface Circle {

kind: "circle";

radius: number;

}

interface Square {

kind: "square";

side: number;

}

type Shape = Circle | Square;

function area(shape: Shape): number {

if (shape.kind === "circle") {

return Math.PI \* shape.radius \* shape.radius;

} else {

return shape.side \* shape.side;

}

}