**1. TypeScript Basics**

* Introduction to TypeScript
* Installing and Setting Up TypeScript (tsc, tsconfig.json)
* Type Annotations (string, number, boolean, any, unknown, void, never)
* Type Inference
* Type Narrowing
* Union and Intersection Types
* Type Aliases (type)
* Interfaces (interface)
* Enums (enum)
* Tuples
* Type Guards (typeof, instanceof, in, custom type guards`)
* Type Assertion (as, <type>value)

**2. Functions and Objects**

* Function Signatures
* Optional and Default Parameters
* Rest Parameters
* Parameter Destructuring
* Function Overloading
* Variadic Functions
* Arrow Functions & Lambda Expressions
* Getters and Setters
* Method Overriding

**3. Object-Oriented Programming (OOP) in TypeScript**

* Classes (class, constructor)
* Access Modifiers (public, private, protected)
* Static Properties and Methods (static)
* Abstract Classes (abstract class)
* Interfaces vs Abstract Classes
* Inheritance (extends, super)
* Multiple Inheritance (Workarounds)
* Method Overloading
* Encapsulation

**4. Advanced TypeScript Features**

* Keyof and Index Types (keyof T)
* Mapped Types (Partial, Readonly, Pick, Omit, Record)
* Conditional Types (T extends U ? X : Y)
* Template Literal Types
* Infer (infer T)
* Deeply Nested Objects

**5. Generics & Utility Types**

* Generic Functions
* Generic Interfaces & Classes
* Constrained Generics (extends)
* Variadic Generics
* Utility Types (Partial, Readonly, Record, Omit, Pick, Extract, Exclude)

**6. Modules & Namespaces**

* Importing and Exporting Modules (import/export)
* Default and Named Exports
* Module Resolution
* Namespaces (namespace)
* Module Types
* Transpilation

**7. Asynchronous Programming**

* Promises (Promise<T>)
* Promise that Resolves to a String
* Async/Await (async function)
* Handling Errors in Async Functions
* Using Promise.all and Promise.race

**8. Design Patterns & Principles**

* SOLID Principles (Single Responsibility, Open-Closed, etc.)
* Factory Pattern
* Singleton Pattern
* Dependency Injection
* Mixins (extends & implements)
* Constrained Mixins

**9. Decorators & Meta-programming**

* Introduction to Decorators (@Decorator)
* Custom Decorators (Class, Method, Property, Parameter)
* Decorator Composition
* Metadata Reflection

**10. Advanced Concepts**

* Iterators and Generators (function\*)
* Symbol (Symbol())
* Duck Typing
* Immutable Arrays
* Intersection vs Union Types
* Variadic Functions

**1. TypeScript Basics**

* Introduction to TypeScript

TypeScript is a superset of JavaScript that adds static typing to the language. It helps developers catch errors at compile time rather than at runtime, making code more robust and maintainable. TypeScript compiles down to plain JavaScript, which can run in any environment that supports JavaScript.

* Installing and Setting Up TypeScript (tsc, tsconfig.json)

To use TypeScript, you need to install it. You can do this via npm:

npm install -g typescript

tsc: This is the TypeScript compiler. You can compile TypeScript files (.ts) into JavaScript files (.js) using the command tsc filename.ts.

tsconfig.json: This is a configuration file that specifies the compiler options and the files to be included in the compilation. You can create it by running tsc --init, which generates a default configuration file.

Example of tsconfig.json:

{

"compilerOptions": {

"target": "es6",

"module": "commonjs",

"strict": true,

"esModuleInterop": true

},

"include": ["src/\*\*/\*"]

}

* Type Annotations (string, number, boolean, any, unknown, void, never)

Type annotations allow you to specify the type of a variable, function parameter, or return value explicitly. Here are some common types:

* string: Represents text.
* number: Represents numeric values.
* boolean: Represents true/false values.
* any: Represents any type; it disables type checking.
* unknown: Similar to any, but safer; you must perform some type checking before using it.
* void: Indicates that a function does not return a value.
* never: Indicates a function that never returns (e.g., it throws an error or has an infinite loop).

Example:

let name: string = "Alice";

let age: number = 30;

let isActive: boolean = true;

let data: any = "Could be anything";

let unknownData: unknown = 42;

* Type Inference

Type inference is the ability of TypeScript to automatically deduce the type of a variable based on its value. You don’t always need to explicitly annotate types.

Example:

let message = "Hello, World!"; // TypeScript infers the type as string

* Type Narrowing

Type narrowing is the process of refining the type of a variable within a conditional block. TypeScript uses control flow analysis to determine the type of a variable based on the code path.

Example:

function printValue(value: string | number) {

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

console.log("String: " + value);

} else {

console.log("Number: " + value);

}

}

* Union and Intersection Types
  + Union Types: Allow a variable to hold multiple types. You can use the pipe (|) operator to define a union.
  + Intersection Types: Combine multiple types into one. You can use the ampersand (&) operator to define an intersection.

Example:// Union Type

function log(value: string | number) {

console.log(value);

}

// Intersection Type

interface Person {

name: string;

}

interface Employee {

employeeId: number;

}

type Worker = Person & Employee;

const worker: Worker = {

name: "Alice",

employeeId: 123

};

* Type Aliases (type)

Type aliases allow you to create a new name for a type. This can be useful for simplifying complex types or creating more readable code.

Example:

type StringOrNumber = string | number;

let value: StringOrNumber;

value = "Hello"; // valid

value = 42; // valid

* Interfaces (interface)

Interfaces define the structure of an object. They can be used to enforce that an object adheres to a specific shape. Interfaces can also extend other interfaces.

Example:

interface User {

name: string;

age: number;

}

const user: User = {

name: "Alice",

age: 30

};

* Enums (enum)

Enums are a way to define a set of named constants. They can be numeric or string-based. Enums improve code readability and maintainability.

Example:

enum Direction {

Up = 1,

Down,

Left,

Right

}

let move: Direction = Direction.Up; // 1

* Tuples

Tuples are a special type of array that allows you to store a fixed number of elements with different types. They are useful for representing a collection of related values.

Example:

let user: [string, number] = ["Alice", 30];

console.log(user[0]); // "Alice"

console.log(user[1]); // 30

* Type Guards (typeof, instanceof, in, custom type guards`)

Type guards are techniques used to narrow down the type of a variable within a conditional block. They can be implemented using typeof, instanceof, in, or custom type guards.

Example:

function isString(value: any): value is string {

return typeof value === "string";

}

function print(value: string | number) {

if (isString(value)) {

console.log("String: " + value);

} else {

console.log("Number: " + value);

}

}

* Type Assertion (as, <type>value)

Type assertion allows you to override TypeScript's inferred type. You can use the as keyword or the angle-bracket syntax to assert a specific type.

Example:

let someValue: any = "this is a string";

let strLength: number = (someValue as string).length; // using 'as'

let strLength2: number = (<string>someValue).length; // using angle-brac

**2. Functions and Objects**

* Function Signatures

A function signature in TypeScript defines the types of parameters a function accepts and the type of value it returns. It serves as a contract for what the function should look like.

Example:

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

return a + b;

}

In this example, the function add takes two parameters of type number and returns a number.

* Optional and Default Parameters
  + **Optional Parameters:** You can mark a parameter as optional by adding a ? after its name. This means the parameter may or may not be provided when the function is called.
  + **Default Parameters:** You can also provide a default value for a parameter. If the caller does not provide a value, the default will be used.

Example:

function greet(name: string, greeting: string = "Hello"): string {

return `${greeting}, ${name}!`;

}

greet("Alice"); // "Hello, Alice!"

greet("Bob", "Hi"); // "Hi, Bob!"

* Rest Parameters

Rest parameters allow you to represent an indefinite number of arguments as an array. This is useful when you don’t know how many arguments will be passed to the function.

function sum(...numbers: number[]): number {

return numbers.reduce((acc, curr) => acc + curr, 0);

}

sum(1, 2, 3, 4); // 10

* Parameter Destructuring

Parameter destructuring allows you to unpack values from arrays or properties from objects into distinct variables. This can make your function parameters more readable.

Example:

function displayUser ({ name, age }: { name: string; age: number }): void {

console.log(`Name: ${name}, Age: ${age}`);

}

displayUser ({ name: "Alice", age: 30 });

* Function Overloading

Function overloading allows you to define multiple signatures for a function, enabling it to handle different types or numbers of parameters. The implementation must be able to handle all defined signatures.

Example:

function combine(a: string, b: string): string;

function combine(a: number, b: number): number;

function combine(a: any, b: any): any {

return a + b;

}

combine("Hello, ", "World!"); // "Hello, World!"

combine(1, 2); // 3

* Variadic Functions

Variadic functions are functions that can take a variable number of arguments. In TypeScript, this is typically achieved using rest parameters.

Example:

function logMessages(...messages: string[]): void {

messages.forEach(msg => console.log(msg));

}

logMessages("Error", "Warning", "Info");

* Arrow Functions & Lambda Expressions

Arrow functions provide a concise syntax for writing function expressions. They also lexically bind the this value, which means they inherit this from the surrounding context.

Example:

* + const add = (a: number, b: number): number => a + b;
  + const numbers = [1, 2, 3];
  + const doubled = numbers.map(num => num \* 2); // [2, 4, 6]
* Getters and Setters

Getters and setters allow you to define methods that get or set the value of an object’s property. They provide a way to control access to properties.

Example:

class Person {

private \_name: string;

constructor(name: string) {

this.\_name = name;

}

get name(): string {

return this.\_name;

}

set name(newName: string) {

this.\_name = newName;

}

}

const person = new Person("Alice");

console.log(person.name); // "Alice"

person.name = "Bob";

console.log(person.name); // "Bob"

* Method Overriding

Method overriding occurs when a subclass provides a specific implementation of a method that is already defined in its superclass. This allows for polymorphism.

Example:

class Animal {

speak(): void {

console.log("Animal speaks");

}

}

class Dog extends Animal {

speak(): void {

console.log("Dog barks");

}

}

const myDog = new Dog();

myDog.speak(); // "Dog barks"

**3. Object-Oriented Programming (OOP) in TypeScript**

* Classes (class, constructor)

A class in TypeScript is a blueprint for creating objects. It encapsulates data and behavior related to that data. The constructor is a special method that is called when an instance of the class is created.

Example:

class Person {

name: string;

age: number;

constructor(name: string, age: number) {

this.name = name;

this.age = age;

}

greet() {

console.log(`Hello, my name is ${this.name} and I am ${this.age} years old.`);

}

}

const person = new Person("Alice", 30);

person.greet(); // "Hello, my name is Alice and I am 30 years old."

* Access Modifiers (public, private, protected)

Access modifiers control the visibility of class members (properties and methods):

* + public: Members are accessible from anywhere (default).
  + private: Members are only accessible within the class itself.
  + protected: Members are accessible within the class and its subclasses.

Example:

class Employee {

public name: string;

private salary: number;

protected department: string;

constructor(name: string, salary: number, department: string) {

this.name = name;

this.salary = salary;

this.department = department;

}

getSalary() {

return this.salary; // Accessible within the class

}

}

const emp = new Employee("Bob", 50000, "Engineering");

console.log(emp.name); // "Bob"

// console.log(emp.salary); // Error: Property 'salary' is private

* Static Properties and Methods (static)

Static properties and methods belong to the class itself rather than to instances of the class. They can be accessed without creating an instance of the class.

Example:

class MathUtils {

static PI: number = 3.14;

static areaOfCircle(radius: number): number {

return this.PI \* radius \* radius;

}

}

console.log(MathUtils.PI); // 3.14

console.log(MathUtils.areaOfCircle(5)); // 78.5

* Abstract Classes (abstract class)

Abstract classes cannot be instantiated directly and are meant to be subclassed. They can contain abstract methods (without implementation) that must be implemented by derived classes.

Example:

abstract class Animal {

abstract makeSound(): void; // Abstract method

move() {

console.log("Moving...");

}

}

class Dog extends Animal {

makeSound() {

console.log("Bark");

}

}

const dog = new Dog();

dog.makeSound(); // "Bark"

dog.move(); // "Moving..."

* Interfaces vs Abstract Classes
  + **Interfaces** define a contract for classes to implement. They can only contain method signatures and properties but no implementation.
  + **Abstract** classes can contain both implemented methods and abstract methods. They can also have constructors and state.

Example:

interface Shape {

area(): number;

}

abstract class ShapeBase {

abstract area(): number;

draw() {

console.log("Drawing shape");

}

}

class Circle extends ShapeBase implements Shape {

constructor(private radius: number) {

super();

}

area() {

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

}

}

* Inheritance (extends, super)

Inheritance allows a class to inherit properties and methods from another class. The extends keyword is used to create a subclass, and the super keyword is used to call the constructor of the parent class.

Example:

class Animal {

constructor(public name: string) {}

speak() {

console.log(`${this.name} makes a sound.`);

}

}

class Dog extends Animal {

speak() {

console.log(`${this.name} barks.`);

}

}

const dog = new Dog("Buddy");

dog.speak(); // "Buddy barks."

* Multiple Inheritance (Workarounds)

TypeScript does not support multiple inheritance directly (a class cannot extend more than one class). However, you can achieve similar behavior using interfaces or mixins.

Example using interfaces:

interface CanRun {

run(): void;

}

interface CanBark {

bark(): void;

}

class Dog implements CanRun, CanBark {

run() {

console.log("Dog is running");

}

bark() {

console.log("Dog barks");

}

}

const dog = new Dog();

dog.run(); // "Dog is running"

dog.bark(); // "Dog barks"

* Method Overloading

Method overloading allows you to define multiple signatures for a method, enabling it to handle different types or numbers of parameters. The implementation of the method can then handle the various cases.

Example:

class Calculator {

add(a: number, b: number): number;

add(a: string, b: string): string;

add(a: any, b: any): any {

return a + b;

}

}

const calc = new Calculator();

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

console.log(calc.add("Hello, ", "World!")); // "Hello, World!"

* Encapsulation

Encapsulation is the principle of bundling data (properties) and methods that operate on that data within a single unit (class). It restricts direct access to some of the object's components, which can prevent the accidental modification of data.

Example:

class BankAccount {

private balance: number = 0;

deposit(amount: number) {

if (amount > 0) {

this.balance += amount;

}

}

getBalance(): number {

return this.balance;

}

}

const account = new BankAccount();

account.deposit(100);

console.log(account.getBalance()); // 100

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

**4. Advanced TypeScript Features**

* Keyof and Index Types (keyof T)

The **keyof** operator creates a union type of the keys of an object type. It allows you to get the keys of a type as a string literal type.

Example:

interface Person {

name: string;

age: number;

}

type PersonKeys = keyof Person; // "name" | "age"

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

return obj[key];

}

const person: Person = { name: "Alice", age: 30 };

const name = getProperty(person, "name"); // "Alice"

* Mapped Types (Partial, Readonly, Pick, Omit, Record)

Mapped types allow you to create new types by transforming properties of an existing type. Common mapped types include:

* + Partial: Makes all properties optional.
  + Readonly: Makes all properties read-only.
  + Pick: Creates a type by picking a set of properties from another type.
  + Omit: Creates a type by omitting a set of properties from another type.
  + Record: Creates a type with specified keys and values.

Examples:

interface Person {

name: string;

age: number;

}

// Partial

type PartialPerson = Partial<Person>; // { name?: string; age?: number; }

// Readonly

type ReadonlyPerson = Readonly<Person>; // { readonly name: string; readonly age: number; }

// Pick

type PersonName = Pick<Person, "name">; // { name: string; }

// Omit

type PersonWithoutAge = Omit<Person, "age">; // { name: string; }

// Record

type StringMap = Record<string, number>; // { [key: string]: number }

* Conditional Types (T extends U ? X : Y)

Conditional types allow you to create types based on a condition. They take the form T extends U ? X : Y, meaning if T is assignable to U, the type will be X; otherwise, it will be Y.

Example:

* + type IsString<T> = T extends string ? "Yes" : "No";
  + type Test1 = IsString<string>; // "Yes"
  + type Test2 = IsString<number>; // "No"
* Template Literal Types

Template literal types allow you to create string literal types by combining string literals and unions. This feature is similar to template literals in JavaScript.

Example:

type EventName = "click" | "mouseover";

type EventHandler = `on${Capitalize<EventName>}`; // "onClick" | "onMouseover"

const handleClick: EventHandler = "onClick"; // valid

// const handleInvalid: EventHandler = "onHover"; // Error: Type '"onHover"' is not assignable

* Infer (infer T)

The infer keyword is used within conditional types to infer a type variable from a type. This is useful for extracting types from complex structures.

Example:

1. type ReturnType<T> = T extends (...args: any[]) => infer R ? R : never;
2. type Func = (a: number, b: string) => boolean;
3. type Result = ReturnType<Func>; // boolean

* Deeply Nested Objects

TypeScript allows you to define types for deeply nested objects. You can use interfaces or type aliases to create complex structures.

Example:

interface Address {

street: string;

city: string;

}

interface User {

name: string;

address: Address;

}

const user: User = {

name: "Alice",

address: {

street: "123 Main St",

city: "Wonderland"

}

};

// Accessing deeply nested properties

const city = user.address.city; // "Wonderland"

You can also create utility types to work with deeply nested objects, such as extracting types or making properties optional.

Example of a utility type to make all properties optional:

type DeepPartial<T> = {

[K in keyof T]?: T[K] extends object ? DeepPartial<T[K]> : T[K];

};

type UserPartial = DeepPartial<User>;

This DeepPartial type recursively makes all properties of a type optional, including nested properties.

**5. Generics & Utility Types**

* Generic Functions

Generic functions allow you to create functions that can work with any data type while maintaining type safety. You define a type parameter (usually denoted by T) that can be used within the function.

Example:

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

return arg;

}

const result1 = identity<string>("Hello"); // "Hello"

const result2 = identity<number>(42); // 42

* Generic Interfaces & Classes

You can also create interfaces and classes that use generics. This allows you to define types that can work with multiple data types.

Example of a Generic Interface:

interface Box<T> {

content: T;

}

const stringBox: Box<string> = { content: "Hello" };

const numberBox: Box<number> = { content: 42 };

Example of a Generic Class:

class GenericContainer<T> {

private items: T[] = [];

add(item: T) {

this.items.push(item);

}

getItems(): T[] {

return this.items;

}

}

const stringContainer = new GenericContainer<string>();

stringContainer.add("Hello");

console.log(stringContainer.getItems()); // ["Hello"]

* Constrained Generics (extends)

Constrained generics allow you to restrict the types that can be used as type parameters. You can use the extends keyword to specify a constraint.

Example:

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

console.log(arg.length);

}

logLength("Hello"); // 5

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

// logLength(42); // Error: Argument of type 'number' is not assignable

* Variadic Generics

Variadic generics allow you to define a generic type that can accept a variable number of type parameters. This feature is useful for creating types that can handle tuples of varying lengths.

Example:

type Tuple<T extends any[]> = {

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

};

function logTuple<T extends any[]>(...args: Tuple<T>): void {

console.log(args);

}

logTuple(1, "Hello", true); // [1, "Hello", true]

* Utility Types (Partial, Readonly, Record, Omit, Pick, Extract, Exclude)

TypeScript provides several built-in utility types that help manipulate types. Here are some commonly used utility types:

1. Partial: Makes all properties of a type optional.
2. Readonly: Makes all properties of a type read-only.
3. Record: Creates a type with specified keys and values.
4. Omit: Creates a type by omitting specified properties from another type.
5. Pick: Creates a type by picking specified properties from another type.
6. Extract: Extracts types from a union that are assignable to another type.
7. Exclude: Excludes types from a union that are assignable to another type.

Examples:

interface Person {

name: string;

age: number;

address: string;

}

// Partial

type PartialPerson = Partial<Person>; // { name?: string; age?: number; address?: string; }

// Readonly

type ReadonlyPerson = Readonly<Person>; // { readonly name: string; readonly age: number; readonly address: string; }

// Record

type StringMap = Record<string, number>; // { [key: string]: number }

// Omit

type PersonWithoutAddress = Omit<Person, "address">; // { name: string; age: number; }

// Pick

type PersonName = Pick<Person, "name">; // { name: string; }

// Extract

type Union = string | number | boolean;

type StringType = Extract<Union, string>; // string

// Exclude

type ExcludedType = Exclude<Union, string>; // number | boolean

**6. Modules & Namespaces**

* Importing and Exporting Modules (import/export)

In TypeScript, you can organize your code into modules, which are reusable pieces of code that can be imported and exported. This helps in maintaining a clean and modular codebase.

Example of Exporting:

// math.ts

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

return a + b;

}

export const PI = 3.14;

Example of Importing:

// main.ts

import { add, PI } from './math';

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

console.log(PI); // 3.14

* Default and Named Exports

1. Named Exports: You can export multiple values from a module using named exports. When importing, you must use the same names.
2. Default Exports: A module can have one default export. You can import it without using curly braces.

Example of Named and Default Exports:

// utils.ts

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

return a \* b;

}

const DEFAULT\_VALUE = 10;

export default DEFAULT\_VALUE; // Default export

Example of Importing Named and Default Exports:

// main.ts

import DEFAULT\_VALUE, { multiply } from './utils';

console.log(multiply(2, 3)); // 6

console.log(DEFAULT\_VALUE); // 10

* Module Resolution

Module resolution is the process by which TypeScript determines what an import refers to. TypeScript uses a set of rules to locate the files that are being imported. The two main strategies are:

* + Node Module Resolution: This is the default strategy, which mimics Node.js module resolution. It looks for modules in node\_modules and follows a specific path resolution strategy.
  + Classic Module Resolution: This is an older strategy that looks for files in the same directory or in parent directories.
  + You can configure module resolution in the tsconfig.json file using the moduleResolution option.
* Namespaces (namespace)

Namespaces are a way to group related code together. They help avoid naming collisions and can be used to organize code in a modular way. However, with the advent of ES6 modules, the use of namespaces has decreased.

Example of a Namespace:

namespace MathUtils {

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

return a + b;

}

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

return a \* b;

}

}

// Using the namespace

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

console.log(MathUtils.multiply(2, 3)); // 6

* Module Types
  + TypeScript supports different module types, which determine how modules are compiled and how they interact with each other. Common module types include:
  + CommonJS: Used in Node.js, where modules are loaded synchronously using require.
  + ES6 Modules: The standard module system in JavaScript, using import and export.
  + AMD (Asynchronous Module Definition): Used in browsers for asynchronous loading of modules.
  + UMD (Universal Module Definition): A pattern that works in both Node.js and browsers.
  + You can specify the module type in the tsconfig.json file using the module option.
* Transpilation

Transpilation is the process of converting TypeScript code into JavaScript code. The TypeScript compiler (tsc) takes TypeScript files and compiles them into JavaScript files that can run in any JavaScript environment.

You can configure the transpilation process in the tsconfig.json file, specifying options such as the target JavaScript version, module system, and more.

Example of a tsconfig.json file:

{

"compilerOptions": {

"target": "es6",

"module": "commonjs",

"outDir": "./dist",

"strict": true

},

"include": ["src/\*\*/\*"]

}

**7. Asynchronous Programming**

* Promises (Promise<T>)

A Promise is an object that represents the eventual completion (or failure) of an asynchronous operation and its resulting value. It can be in one of three states: pending, fulfilled, or rejected.

Example:

const fetchData = (): Promise<string> => {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = true; // Simulate success or failure

if (success) {

resolve("Data fetched successfully!");

} else {

reject("Error fetching data.");

}

}, 1000);

});

};

fetchData()

.then(result => console.log(result)) // "Data fetched successfully!"

.catch(error => console.error(error));

* Promise that Resolves to a String

You can create a Promise that resolves to a specific type, such as a string. This is useful for type safety in TypeScript.

Example:

const getStringPromise = (): Promise<string> => {

return new Promise((resolve) => {

setTimeout(() => {

resolve("Hello, World!");

}, 1000);

});

};

getStringPromise().then(result => {

console.log(result); // "Hello, World!"

});

* Async/Await (async function)

async and await are syntactic sugar built on top of Promises, making asynchronous code easier to read and write. An async function always returns a Promise, and you can use await to pause execution until the Promise is resolved.

Example:

const fetchDataAsync = async (): Promise<string> => {

return new Promise((resolve) => {

setTimeout(() => {

resolve("Data fetched successfully!");

}, 1000);

});

};

const main = async () => {

const result = await fetchDataAsync();

console.log(result); // "Data fetched successfully!"

};

main();

* Handling Errors in Async Functions

You can handle errors in async functions using try...catch blocks. This allows you to catch any errors that occur during the execution of the awaited Promise.

Example:

const fetchDataWithErrorHandling = async (): Promise<string> => {

return new Promise((resolve, reject) => {

setTimeout(() => {

const success = false; // Simulate failure

if (success) {

resolve("Data fetched successfully!");

} else {

reject("Error fetching data.");

}

}, 1000);

});

};

const mainWithErrorHandling = async () => {

try {

const result = await fetchDataWithErrorHandling();

console.log(result);

} catch (error) {

console.error(error); // "Error fetching data."

}

};

mainWithErrorHandling();

* Using Promise.all and Promise.race

**Promise.all:** Takes an array of Promises and returns a single Promise that resolves when all of the Promises in the array have resolved or rejects if any of the Promises reject.

Example:

const promise1 = Promise.resolve(1);

const promise2 = Promise.resolve(2);

const promise3 = new Promise<number>((resolve) => setTimeout(() => resolve(3), 1000));

Promise.all([promise1, promise2, promise3])

.then(results => {

console.log(results); // [1, 2, 3]

})

.catch(error => {

console.error(error);

});

**Promise.race**: Takes an array of Promises and returns a single Promise that resolves or rejects as soon as one of the Promises in the array resolves or rejects.

Example:

const promiseA = new Promise<string>((resolve) => setTimeout(() => resolve("A"), 1000));

const promiseB = new Promise<string>((resolve) => setTimeout(() => resolve("B"), 500));

Promise.race([promiseA, promiseB])

.then(result => {

console.log(result); // "B" (because it resolves first)

})

.catch(error => {

console.error(error);

});

**8. Design Patterns & Principles**

* SOLID Principles (Single Responsibility, Open-Closed, etc.)
* The SOLID principles are a set of five design principles intended to make software designs more understandable, flexible, and maintainable. They are:
* Single Responsibility Principle (SRP): A class should have only one reason to change, meaning it should have only one job or responsibility.

Example:

class User {

constructor(public name: string) {}

}

class UserRepository {

save(user: User) {

// Save user to the database

}

}

Open-Closed Principle (OCP): Software entities (classes, modules, functions, etc.) should be open for extension but closed for modification. This means you should be able to add new functionality without changing existing code.

Example:

interface Shape {

area(): number;

}

class Circle implements Shape {

constructor(private radius: number) {}

area() {

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

}

}

class Rectangle implements Shape {

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

area() {

return this.width \* this.height;

}

}

function calculateArea(shape: Shape) {

return shape.area();

}

**Liskov Substitution Principle (LSP):** Subtypes must be substitutable for their base types without altering the correctness of the program. This means derived classes should extend the base class without changing its behavior.

Example:

class Bird {

fly() {

console.log("Flying");

}

}

class Sparrow extends Bird {}

class Ostrich extends Bird {

fly() {

throw new Error("Ostriches can't fly");

}

}

**Interface Segregation Principle (ISP):** Clients should not be forced to depend on interfaces they do not use. This means you should create smaller, more specific interfaces rather than a large, general-purpose interface.

Example:

interface Printer {

print(): void;

}

interface Scanner {

scan(): void;

}

class MultiFunctionPrinter implements Printer, Scanner {

print() {

console.log("Printing");

}

scan() {

console.log("Scanning");

}

}

**Dependency Inversion Principle (DIP):** High-level modules should not depend on low-level modules. Both should depend on abstractions (e.g., interfaces). Abstractions should not depend on details; details should depend on abstractions.

Example:

interface Database {

save(data: any): void;

}

class MySQLDatabase implements Database {

save(data: any) {

// Save to MySQL

}

}

class UserService {

constructor(private db: Database) {}

saveUser (user: any) {

this.db.save(user);

}

}

* Factory Pattern

The Factory Pattern is a creational design pattern that provides an interface for creating objects in a superclass but allows subclasses to alter the type of objects that will be created.

Example:

interface Shape {

draw(): void;

}

class Circle implements Shape {

draw() {

console.log("Drawing a Circle");

}

}

class Square implements Shape {

draw() {

console.log("Drawing a Square");

}

}

class ShapeFactory {

static createShape(type: string): Shape {

if (type === "circle") {

return new Circle();

} else if (type === "square") {

return new Square();

}

throw new Error("Unknown shape type");

}

}

const shape1 = ShapeFactory.createShape("circle");

shape1.draw(); // "Drawing a Circle"

* Singleton Pattern

The Singleton Pattern ensures that a class has only one instance and provides a global point of access to it.

Example:

class Singleton {

private static instance: Singleton;

private constructor() {}

static getInstance(): Singleton {

if (!Singleton.instance) {

Singleton.instance = new Singleton();

}

return Singleton.instance;

}

}

// Usage

const singleton1 = Singleton.getInstance();

const singleton2 = Singleton.getInstance();

console.log(singleton1 === singleton2); // true

* Dependency Injection

Dependency Injection (DI) is a design pattern used to implement IoC (Inversion of Control), allowing a class to receive its dependencies from an external source rather than creating them itself.

Example:

class Logger {

log(message: string) {

console.log(message);

}

}

class UserService {

constructor(private logger: Logger) {}

createUser (name: string) {

this.logger.log(`User ${name} created.`);

}

}

// Usage

const logger = new Logger();

const userService = new UserService(logger);

userService.createUser ("Alice"); // "User Alice created."

* Mixins (extends & implements)

Mixins are a way to create classes that can inherit methods and properties from multiple sources. In TypeScript, you can use extends and implements to create mixins.

Example:

type Constructor<T = {}> = new (...args: any[]) => T;

function CanFly<T extends Constructor>(Base: T) {

return class extends Base {

fly() {

console.log("Flying");

}

};

}

class Animal {

eat() {

console.log("Eating");

}

}

class Bird extends CanFly(Animal) {}

const bird = new Bird();

bird.eat(); // "Eating"

bird.fly(); // "Flying"

* Constrained Mixins

Constrained mixins allow you to create mixins that can only be applied to classes that meet certain criteria, ensuring type safety.

Example:

interface CanRun {

run(): void;

}

function CanRunMixin<T extends Constructor>(Base: T) {

return class extends Base implements CanRun {

run() {

console.log("Running");

}

};

}

class Person {

walk() {

console.log("Walking");

}

}

class Athlete extends CanRunMixin(Person) {}

const athlete = new Athlete();

athlete.walk(); // "Walking"

athlete.run(); // "Running"

**9. Decorators & Meta-programming**

* Introduction to Decorators (@Decorator)

Decorators are a special kind of declaration that can be attached to a class, method, accessor, property, or parameter. They are a way to modify or enhance the behavior of the target they are applied to. Decorators are a stage 2 proposal for JavaScript and are available in TypeScript.

Example:

function log(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const originalMethod = descriptor.value;

descriptor.value = function (...args: any[]) {

console.log(`Calling ${propertyKey} with arguments: ${args}`);

return originalMethod.apply(this, args);

};

}

class Calculator {

@log

add(a: number, b: number) {

return a + b;

}

}

const calc = new Calculator();

calc.add(2, 3); // Logs: "Calling add with arguments: 2,3"

* Custom Decorators (Class, Method, Property, Parameter)

You can create custom decorators for different targets:

**Class Decorator:** A function that takes a class constructor and can modify or replace it.

Example:

function sealed(constructor: Function) {

Object.seal(constructor);

Object.seal(constructor.prototype);

}

@sealed

class Person {

constructor(public name: string) {}

}

**Method Decorator:** A function that takes the target object, the name of the method, and the property descriptor.

Example:

function logMethod(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

const originalMethod = descriptor.value;

descriptor.value = function (...args: any[]) {

console.log(`Method ${propertyKey} called with args: ${args}`);

return originalMethod.apply(this, args);

};

}

**Property Decorator**: A function that takes the target object and the name of the property.

Example:

function logProperty(target: any, propertyKey: string) {

let value: any;

const getter = () => value;

const setter = (newValue: any) => {

console.log(`Setting value ${newValue} to ${propertyKey}`);

value = newValue;

};

Object.defineProperty(target, propertyKey, { get: getter, set: setter });

}

class User {

@logProperty

name: string;

}

**Parameter Decorator**: A function that takes the target object, the name of the method, and the index of the parameter.

Example:

function logParameter(target: any, propertyKey: string, parameterIndex: number) {

console.log(`Parameter at index ${parameterIndex} in method ${propertyKey} has been decorated.`);

}

class UserService {

createUser (@logParameter name: string) {}

}

* Decorator Composition

You can compose multiple decorators together to apply multiple behaviors to a single target.

Example:

function firstDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

console.log("First decorator");

}

function secondDecorator(target: any, propertyKey: string, descriptor: PropertyDescriptor) {

console.log("Second decorator");

}

class Example {

@firstDecorator

@secondDecorator

method() {}

}

* Metadata Reflection

TypeScript provides a way to attach metadata to classes and their members using decorators. You can use the reflect-metadata library to enable this feature.

Example:

import "reflect-metadata";

function logType(target: any, propertyKey: string) {

const type = Reflect.getMetadata("design:type", target, propertyKey);

console.log(`Property ${propertyKey} type: ${type.name}`);

}

class Example {

@logType

name: string;

}

**10. Advanced Concepts**

* Iterators and Generators (function\*)

Iterators are objects that allow you to traverse through a collection (like an array) one element at a time. Generators are a special type of function that can be paused and resumed, allowing you to create iterators easily.

Example:

function\* numberGenerator() {

yield 1;

yield 2;

yield 3;

}

const iterator = numberGenerator();

console.log(iterator.next()); // { value: 1, done: false }

console.log(iterator.next()); // { value: 2, done: false }

console.log(iterator.next()); // { value: 3, done: false }

console.log(iterator .next()); // { value: undefined, done: true }

* Symbol (Symbol())

Symbols are a new primitive data type introduced in ES6. They are unique and immutable, making them useful for creating unique object keys.

Example:

const uniqueSymbol = Symbol("description");

const obj = {

[uniqueSymbol]: "value"

};

console.log(obj[uniqueSymbol]); // "value"

* Duck Typing

Duck typing is a concept in TypeScript where the type of an object is determined by its properties and methods rather than its explicit type. If an object has the required properties, it can be treated as that type.

Example:

interface Duck {

quack(): void;

}

function makeItQuack(duck: Duck) {

duck.quack();

}

const realDuck = {

quack() {

console.log("Quack!");

}

};

makeItQuack(realDuck); // "Quack!"

* Immutable Arrays

Immutable arrays are arrays that cannot be modified after they are created. This can be achieved using libraries like Immutable.js or by using spread operators to create new arrays.

Example:

const originalArray = [1, 2, 3];

const newArray = [...originalArray, 4]; // Creates a new array

console.log(originalArray); // [1, 2, 3]

console.log(newArray); // [1, 2, 3, 4]

* Intersection vs Union Types

Intersection types combine multiple types into one, while union types allow a variable to be one of several types.

Example of Intersection:

interface A {

a: number;

}

interface B {

b: string;

}

type AB = A & B;

const obj: AB = { a: 1, b: "hello" };

Example of Union:

type StringOrNumber = string | number;

let value: StringOrNumber;

value = "hello"; // valid

value = 42; // valid

* Variadic Functions

Variadic functions are functions that can take a variable number of arguments. In TypeScript, you can use the rest parameter syntax to define such functions.

Example:

function sum(...numbers: number[]): number {

return numbers.reduce((acc, curr) => acc + curr, 0);

}

console.log(sum(1, 2, 3)); // 6

console.log(sum(1, 2, 3, 4, 5)); // 15