**Exercise 1: Basic Abstract Class**

abstract class Shape {

constructor(protected color: string) {}

abstract calculateArea(): number;

getColor(): string {

return this.color;

}

}

class Circle extends Shape {

constructor(color: string, private radius: number) {

super(color);

}

calculateArea(): number {

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

}

}

class Rectangle extends Shape {

constructor(color: string, private width: number, private height: number) {

super(color);

}

calculateArea(): number {

return this.width \* this.height;

}

}

// Testing

const circle = new Circle("Red", 5);

console.log(circle.getColor(), circle.calculateArea());

const rectangle = new Rectangle("Blue", 10, 5);

console.log(rectangle.getColor(), rectangle.calculateArea());

**Exercise 2: Implementing an Interface**

interface Vehicle {

speed: number;

move(): void;

}

class Car implements Vehicle {

constructor(public speed: number) {}

move(): void {

console.log(`Car is moving at ${this.speed} km/h.`);

}

}

class Bicycle implements Vehicle {

constructor(public speed: number) {}

move(): void {

console.log(`Bicycle is moving at ${this.speed} km/h.`);

}

}

// Testing

const car = new Car(100);

car.move();

const bicycle = new Bicycle(20);

bicycle.move();

**Exercise 3: Combining Abstract Class and Interface**

interface Flyable {

fly(): void;

}

abstract class Bird {

constructor(public name: string) {}

eat(): void {

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

}

}

class Eagle extends Bird implements Flyable {

fly(): void {

console.log(`${this.name} is flying high.`);

}

}

class Sparrow extends Bird implements Flyable {

fly(): void {

console.log(`${this.name} is flying low.`);

}

}

// Testing

const eagle = new Eagle("Eagle");

eagle.eat();

eagle.fly();

const sparrow = new Sparrow("Sparrow");

sparrow.eat();

sparrow.fly();

**Exercise 4: Interface Inheritance**

interface Person {

name: string;

speak(): void;

}

interface Employee extends Person {

salary: number;

work(): void;

}

class Developer implements Employee {

constructor(public name: string, public salary: number) {}

speak(): void {

console.log(`${this.name} says hello!`);

}

work(): void {

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

}

}

// Testing

const dev = new Developer("SML", 50000);

dev.speak();

dev.work();

**Advanced Exercise 1: Abstract Class for Banking System**

abstract class BankAccount {

protected balance: number;

constructor(protected accountNumber: string, balance: number) {

this.balance = balance;

}

deposit(amount: number): void {

this.balance += amount;

console.log(`Deposited ${amount}. New balance: ${this.balance}`);

}

abstract withdraw(amount: number): boolean;

getBalance(): number {

return this.balance;

}

}

class SavingsAccount extends BankAccount {

private withdrawalLimit: number = 5000;

withdraw(amount: number): boolean {

if (amount > this.withdrawalLimit || amount > this.balance) {

console.log("Withdrawal failed");

return false;

}

this.balance -= amount;

console.log(`Withdrawn ${amount}. New balance: ${this.balance}`);

return true;

}

}

class CheckingAccount extends BankAccount {

private overdraftLimit: number = 1000;

withdraw(amount: number): boolean {

if (amount > this.balance + this.overdraftLimit) {

console.log("Withdrawal failed");

return false;

}

this.balance -= amount;

console.log(`Withdrawn ${amount}. New balance: ${this.balance}`);

return true;

}

}

// Testing

const savings = new SavingsAccount("SA123", 10000);

savings.withdraw(6000);

savings.deposit(2000);

const checking = new CheckingAccount("CA456", 2000);

checking.withdraw(2500);

checking.deposit(1000);

**Advanced Exercise 2: Interface for Payment System**

interface PaymentMethod {

pay(amount: number): void;

refund(amount: number): void;

}

class CreditCard implements PaymentMethod {

constructor(private balance: number) {}

pay(amount: number): void {

if (amount > this.balance) {

console.log("Payment failed");

} else {

this.balance -= amount;

console.log(`Paid ${amount}. Remaining balance: ${this.balance}`);

}

}

refund(amount: number): void {

this.balance += amount;

console.log(`Refunded ${amount}. New balance: ${this.balance}`);

}

}

class DebitCard implements PaymentMethod {

constructor(private balance: number) {}

pay(amount: number): void {

if (amount > this.balance) {

console.log("Insufficient funds");

} else {

this.balance -= amount;

console.log(`Paid ${amount}. Remaining balance: ${this.balance}`);

}

}

refund(amount: number): void {

this.balance += amount;

console.log(`Refunded ${amount}. New balance: ${this.balance}`);

}

}

class UPI implements PaymentMethod {

constructor(private balance: number) {}

pay(amount: number): void {

if (amount > this.balance) {

console.log("Payment failed");

} else {

this.balance -= amount;

console.log(`Paid ${amount}. Remaining balance: ${this.balance}`);

}

}

refund(amount: number): void {

this.balance += amount;

console.log(`Refunded ${amount}. New balance: ${this.balance}`);

}

}

// Testing

const creditCard = new CreditCard(5000);

creditCard.pay(2000);

creditCard.refund(500);

**Advanced Exercise 3: Combining Abstract Class and Interface**

abstract class PaymentGateway {

processPayment(amount: number): void {

console.log(`Processing payment of ${amount}`);

}

processRefund(amount: number): void {

console.log(`Processing refund of ${amount}`);

}

abstract validateTransaction(transactionId: string): boolean;

}

interface SecurePayment {

authenticateUser(userId: string): boolean;

}

class PayPal extends PaymentGateway implements SecurePayment {

validateTransaction(transactionId: string): boolean {

console.log(`Validating transaction: ${transactionId}`);

return true;

}

authenticateUser(userId: string): boolean {

console.log(`Authenticating user: ${userId}`);

return true;

}

}

// Testing

const paypal = new PayPal();

paypal.authenticateUser("user123");

paypal.processPayment(3000);

paypal.validateTransaction("txn456");

**Advanced Exercise 4: Multiple Interface Implementation**

interface Loan {

applyForLoan(amount: number): void;

}

interface Insurance {

applyForInsurance(policy: string): void;

}

class Customer implements Loan, Insurance {

private loans: number[] = [];

private insurances: string[] = [];

applyForLoan(amount: number): void {

this.loans.push(amount);

console.log(`Loan of ${amount} applied.`);

}

applyForInsurance(policy: string): void {

this.insurances.push(policy);

console.log(`Insurance policy '${policy}' applied.`);

}

}

// Testing

const customer = new Customer();

customer.applyForLoan(50000);

customer.applyForInsurance("Health");