**Exercise 1: Basic Abstract Class  
Create an abstract class Shape with:**

An abstract method calculateArea(): number  
A constructor that takes color as a property  
A method getColor(): string that returns the color  
Then, create two subclasses Circle and Rectangle that implement calculateArea().

 Task  
Implement Shape, Circle, and Rectangle  
Create instances and call the methods

abstract class Shape {

constructor(private 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;

}

}

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

console.log(`Circle Area: ${circle.calculateArea()}, Color: ${circle.getColor()}`);

const rectangle = new Rectangle("Blue", 4, 6);

console.log(`Rectangle Area: ${rectangle.calculateArea()}, Color: ${rectangle.getColor()}`);

**Exercise 2: Implementing an Interface  
Create an interface Vehicle with:**

A property speed: number  
A method move(): void  
Then, implement Vehicle in two classes: Car and Bicycle.

 Task  
Implement Vehicle, Car, and Bicycle  
Create instances and call the move() method

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.`);

}

}

const car = new Car(120);

car.move();

const bicycle = new Bicycle(20);

bicycle.move();

**Exercise 3: Combining Abstract Class and Interface  
Create an interface Flyable with:**

A method fly(): void  
Create an abstract class Bird with:

A property name: string  
A method eat(): void  
Then, create two classes Eagle and Sparrow that extend Bird and implement Flyable.

 Task  
Implement Bird and Flyable  
Create Eagle and Sparrow and call eat() and fly()

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.`);

}

}

const eagle = new Eagle("Eagle");

eagle.eat();

eagle.fly();

const sparrow = new Sparrow("Sparrow");

sparrow.eat();

sparrow.fly();

**Exercise 4: Interface Inheritance  
Create two interfaces:**

Person with name: string and speak(): void  
Employee (extends Person) with salary: number and work(): void  
Create a class Developer that implements Employee.

Task  
Implement Person, Employee, and Developer  
Create an instance and call speak() and work()

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: "I love coding!"`);

}

work(): void {

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

}

}

const dev = new Developer("Alice", 80000);

dev.speak();

dev.work();

**Advanced example**

**Exercise 1: Abstract Class for Banking System  
Create an abstract class BankAccount with:**

Protected properties: accountNumber, balance  
Constructor to initialize accountNumber and balance

Methods:  
deposit(amount: number): void (increases balance)  
Abstract method withdraw(amount: number): boolean (returns true if successful, false otherwise)  
Method get Balance(): number  
Then, create two subclasses Savings Account and Checking Account that:

Implement with draw()  
Savings Account should have a withdrawal limit  
Checking Account should allow overdraft up to a certain limit  
Tasks  
Implement Bank Account, Savings Account, and Checking Account  
Create instances and test deposit/withdraw functions

abstract class BankAccount {

protected balance: number;

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

this.balance = initialBalance;

}

deposit(amount: number): void {

this.balance += amount;

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

}

abstract withdraw(amount: number): boolean;

getBalance(): number {

return this.balance;

}

}

class SavingsAccount extends BankAccount {

private withdrawalLimit: number = 500;

withdraw(amount: number): boolean {

if (amount > this.withdrawalLimit) {

console.log("Withdrawal limit exceeded.");

return false;

}

if (amount > this.balance) {

console.log("Insufficient balance.");

return false;

}

this.balance -= amount;

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

return true;

}

}

class CheckingAccount extends BankAccount {

private overdraftLimit: number = 200;

withdraw(amount: number): boolean {

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

console.log("Overdraft limit exceeded.");

return false;

}

this.balance -= amount;

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

return true;

}

}

const savings = new SavingsAccount("SA123", 1000);

savings.deposit(200);

savings.withdraw(600);

savings.withdraw(400);

const checking = new CheckingAccount("CA456", 500);

checking.deposit(300);

checking.withdraw(900);

checking.withdraw(200);

**Exercise 2: Interface for Payment System  
Create an interface Payment Method with:**

A method pay(amount: number): void  
A method refund(amount: number): void  
Then, create three classes Credit Card, Debit Card, and UPI that:

Implement Payment Method  
Each class should have a balance and implement pay() and refund()  
Tasks  
Implement Payment Method, Credit Card, Debit Card, and UPI  
Create instances and simulate payments & refunds

interface PaymentMethod {

pay(amount: number): void;

refund(amount: number): void;

}

class Credit Card implements Payment Method {

private balance: number;

constructor(initial Balance: number) {

this.balance = initial Balance;

}

pay(amount: number): void {

if (this.balance >= amount) {

this.balance -= amount;

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

} else {

console.log("Credit Card: Insufficient funds.");

}

}

refund(amount: number): void {

this.balance += amount;

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

}

}

class DebitCard implements Payment Method {

private balance: number;

constructor(initialBalance: number) {

this.balance = initialBalance;

}

pay(amount: number): void {

if (this.balance >= amount) {

this.balance -= amount;

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

} else {

console.log("DebitCard: Insufficient funds.");

}

}

refund(amount: number): void {

this.balance += amount;

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

}

}

class UPI implements PaymentMethod {

private balance: number;

constructor(initialBalance: number) {

this.balance = initialBalance;

}

pay(amount: number): void {

if (this.balance >= amount) {

this.balance -= amount;

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

} else {

console.log("UPI: Insufficient funds.");

}

}

refund(amount: number): void {

this.balance += amount;

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

}

}

const creditCard = new CreditCard(1000);

creditCard.pay(200);

creditCard.refund(50);

const debitCard = new DebitCard(500);

debitCard.pay(600);

debitCard.refund(100);

const upi = new UPI(300);

upi.pay(100);

upi.refund(20);

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

 Abstract class PaymentGateway with:

 A method processPayment(amount: number): void  
A method processRefund(amount: number): void  
An abstract method validateTransaction(transactionId: string): boolean  
Interface SecurePayment with:

Method authenticateUser(userId: string): boolean  
Then, create a class PayPal that:

Extends PaymentGateway  
Implements SecurePayment  
Implements validateTransaction() and authenticateUser()  
Tasks  
Implement PaymentGateway, SecurePayment, and PayPal  
Create an instance of PayPal and test methods

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 transactionId.length > 5; // Example validation

}

authenticateUser(userId: string): boolean {

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

return userId.startsWith("user\_"); // Example authentication

}

}

const paypal = new PayPal();

paypal.processPayment(150);

paypal.processRefund(50);

console.log("Transaction Valid:", paypal.validateTransaction("TXN123"));

console.log("User Authenticated:", paypal.authenticateUser("user\_abc"));

**Exercise 4: Multiple Interface Implementation  
Create:**

 Interface Loan with applyForLoan(amount: number): void  
Interface Insurance with applyForInsurance(policy: string): void  
Then, create a class Customer that:

 Implements Loan and Insurance  
Stores loan and insurance details in properties  
Tasks  
Implement Loan, Insurance, and Customer  
Create an instance of Customer and test loan & insurance applications

interface Loan {

applyForLoan(amount: number): void;

}

interface Insurance {

applyForInsurance(policy: string): void;

}

class Customer implements Loan, Insurance {

private loanAmount: number = 0;

private insurancePolicy: string = "";

applyForLoan(amount: number): void {

this.loanAmount = amount;

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

}

applyForInsurance(policy: string): void {

this.insurancePolicy = policy;

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

}

}

const customer = new Customer();

customer.applyForLoan(5000);

customer.applyForInsurance("Health Insurance");