**TYPE SCRIPT**

1. Develop a program show casing the usage of Tuple data type. Assume you are getting a collection of Customer records and each element of customer records might have values of different data type. Represent the same as Collection of Tuple, iterate it, manipuate it and print it.

let Customer\_Records:[String,number,String,String][];

Customer\_Records=[["Ali",9999999999,"India","creditCard"],["Nani",8888888888,"India","CheckBook"],

["sruthi",8888877777,"India","Transferring"]];

Customer\_Records.push(["priya",7777788888,"Australia","Visa-Gold-Card"]);

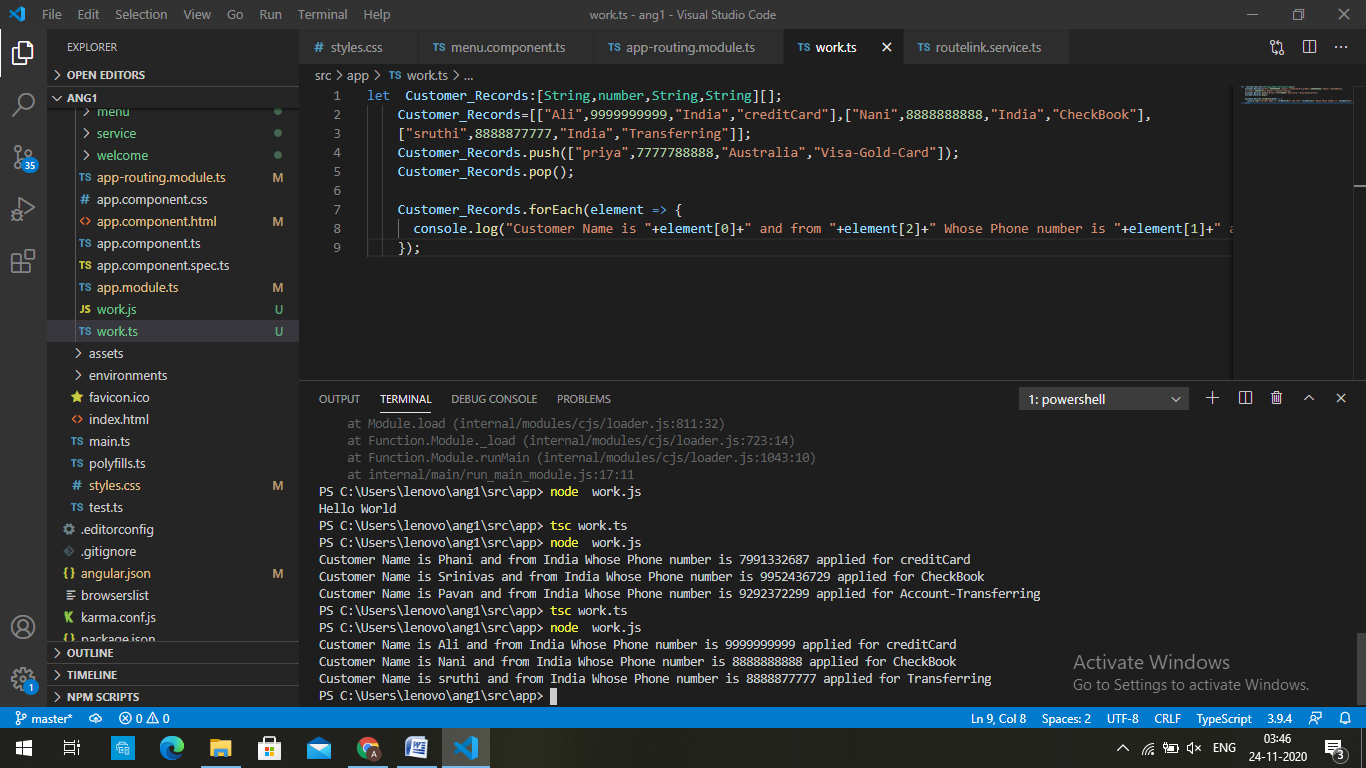
Customer\_Records.pop();

Customer\_Records.forEach(element => {

console.log("Customer Name is "+element[0]+" and from "+element[2]+" Whose Phone number is "+element[1]+" applied for "+element[3]);

});

Output:



2. Develop a program that will calculate the surface area of a)Rectangle, b)Square,c)Triangle and make sure the precision with 2 decimal places.

export class Triangle{

triangle\_area=(height:number,breadth:number)=>{

return 0.5\*height\*breadth;

}

}

export class Square{

private V1:number;

constructor(V1:number){

this.V1=V1;

}

square\_area=()=>{

return this.V1\*this.V1;

}

}

export class Rectangle{

private length: number;

private breadth:number;

constructor(length:number,breadth:number){

this.length=length;

this.breadth=breadth;

}

rectangle\_area=()=>{

console.log("The surface area of a rectangle is "+(this.length\*this.breadth))

}

}

let rectangle:Rectangle = new Rectangle(3,4);

rectangle.rectangle\_area();

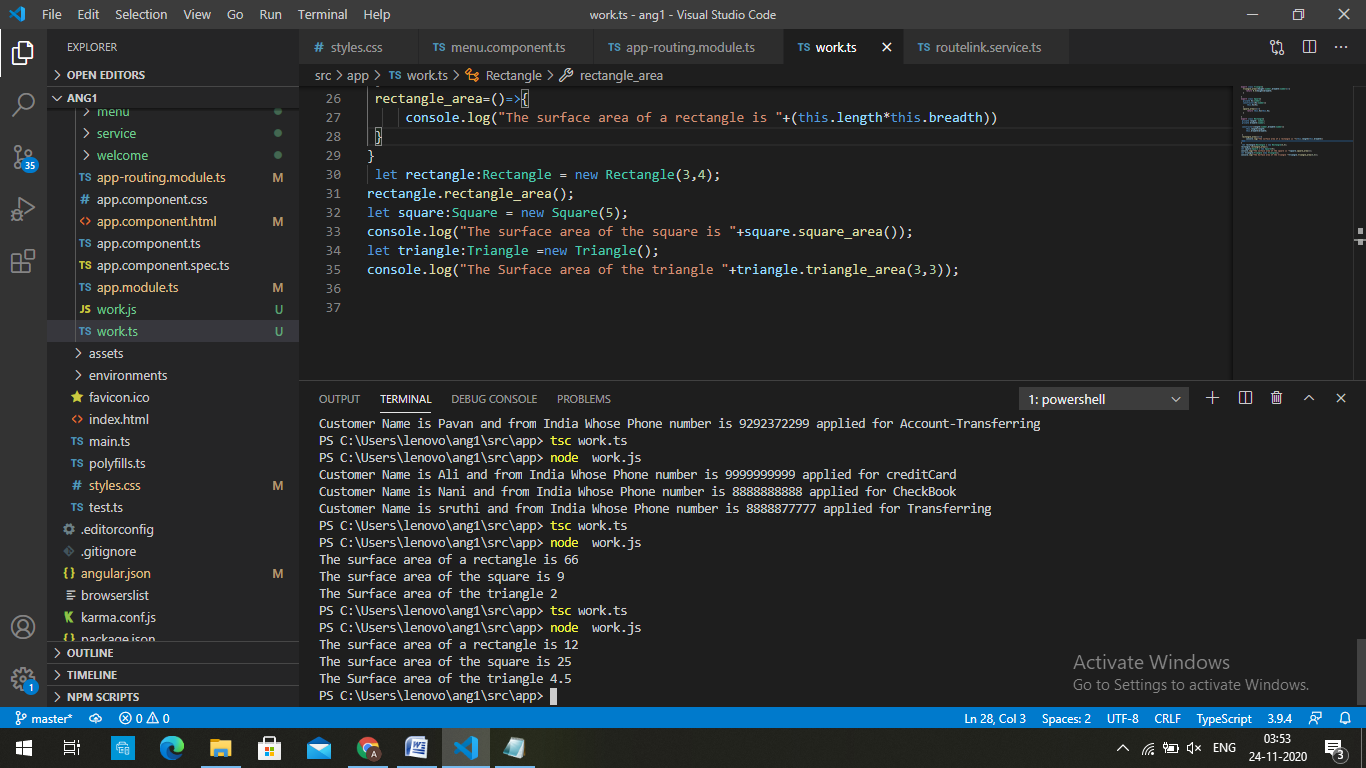
let square:Square = new Square(5);

console.log("The surface area of the square is "+square.square\_area());

let triangle:Triangle =new Triangle();

console.log("The Surface area of the triangle "+triangle.triangle\_area(3,3));

Output:



3. Develop a program that exhibits inferred typing in Angular

let x = 3;

// ^ = let x: number

The type of the x variable is inferred to be number.

When a type inference is made from several expressions, the types of those expressions are used to calculate a “best common type”. For example,

let x = [0, 1, null];

// ^ = let x: (number | null)[]

4. Develop a program that stores the days in a week in enum, and use the enum values in the program

export enum Days {

Sunday = "Mystical",

Monday = "let's play",

Tuesday = "secret",

Wednesday = "Mine",

Thrusady = "Unordinary",

Friday = "I am sorry",

Saturday = "Eggnoid",

}

export class TimeTable{

upd=(Day:String)=>{

for(var d in Days){

if(Day==d)

console.log("Welcome to "+d+". Today's update is "+ Days[d]);

}

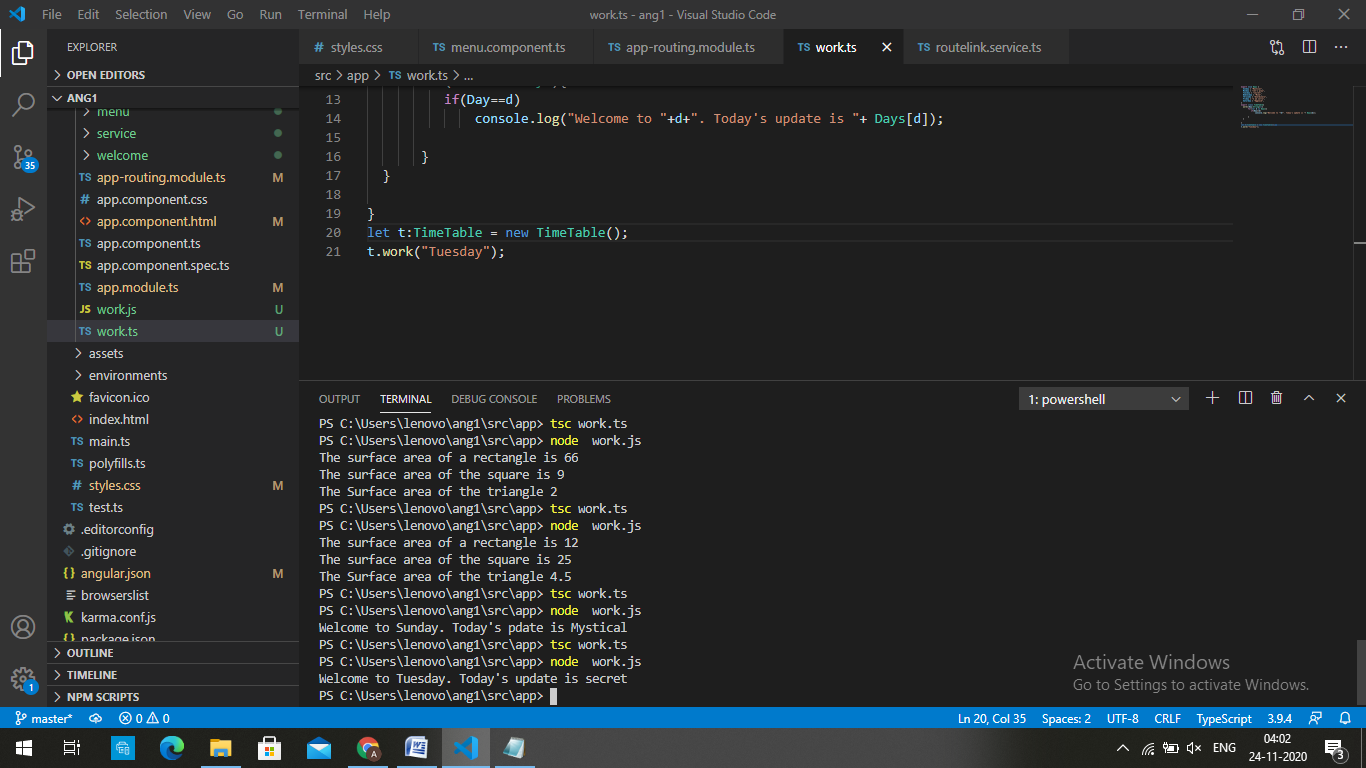
}

}

let t:TimeTable = new TimeTable();

t.upd ("Tuesday");

Output:



5. Develop a program that exhibits union data type. And show case how an operations can be performed on these data stored in this variable. For example, store string and function data type on these variables

export class Union{

Type=(c: (string | number))=>

{

if(typeof(c) === "number")

console.log(c+' is number.')

else if(typeof(c) === "string")

console.log(c + ' is string.');

}

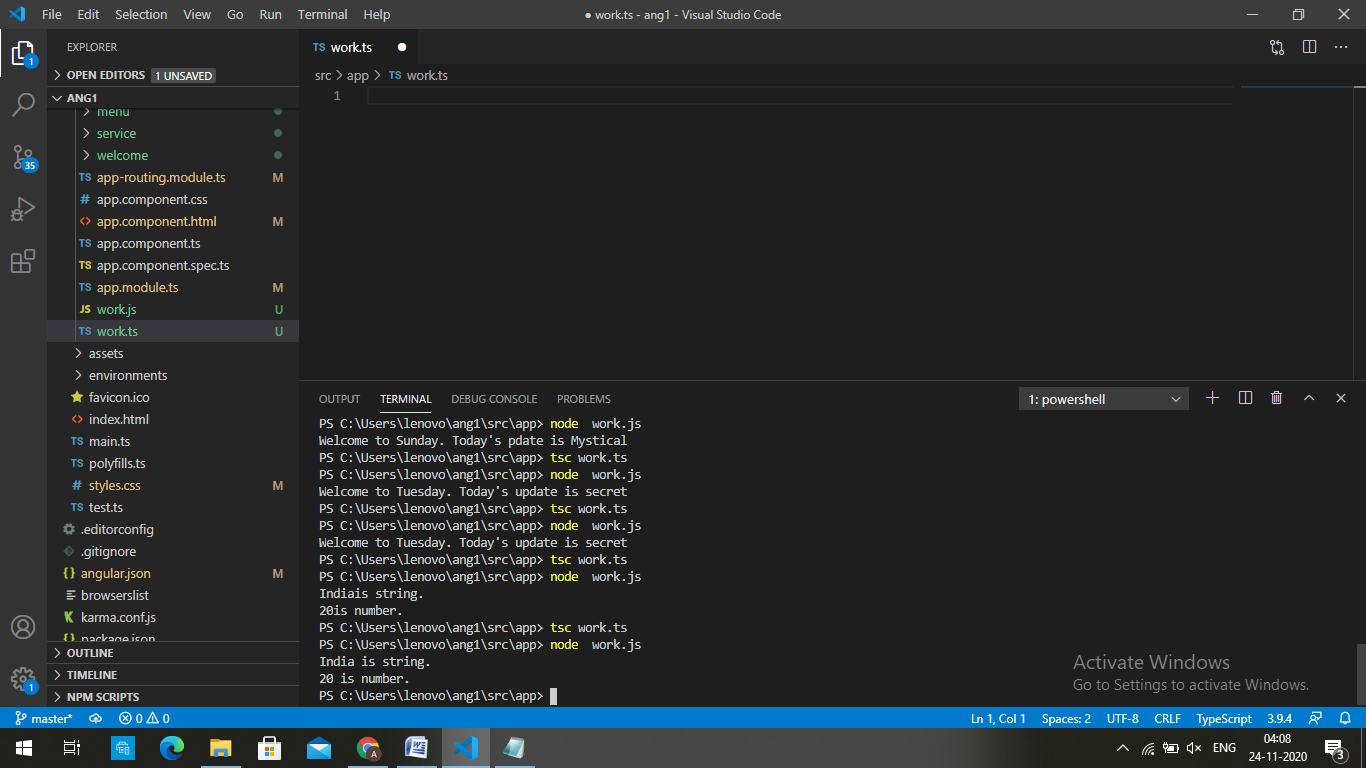
}

let u:Union = new Union()

u.Type("India");

u.Type(20);

Output:



6. Develop a Car Class in Typescript that has following attributes :- Car Color,Engine Capacity,No OfCylinders. And Methods:- StartCar(), StopCar(), AccelerateCar(), OpenCarLock(), CloseCarLock(). And Develop a program that will create instance of this class, and able to call its methods

export class car{

private car\_color;

private Engine;

private Capacity;

private Noofcylinders;

constructor(car\_color:String,Engine:String,Capacity:Number,Noofcylinders:number){

this.car\_color=car\_color;

this.Engine=Engine;

this.Capacity=Capacity;

this.Noofcylinders=Noofcylinders;

}

StartCar=()=>{

return "The car is started";

}

StopCar=()=>{

return "The Car is stopped";

}

Accelarate=()=>{

return "The Car is accelerating";

}

OpenCarLock=()=>{

return "The Car door is opened";

}

CloseCarLock=()=>{

return "The Car door is closed";

}

}

let Car:car = new car("Brown","DOHC I-4/111kW Drive Motor",55,6);

console.log(Car.OpenCarLock())

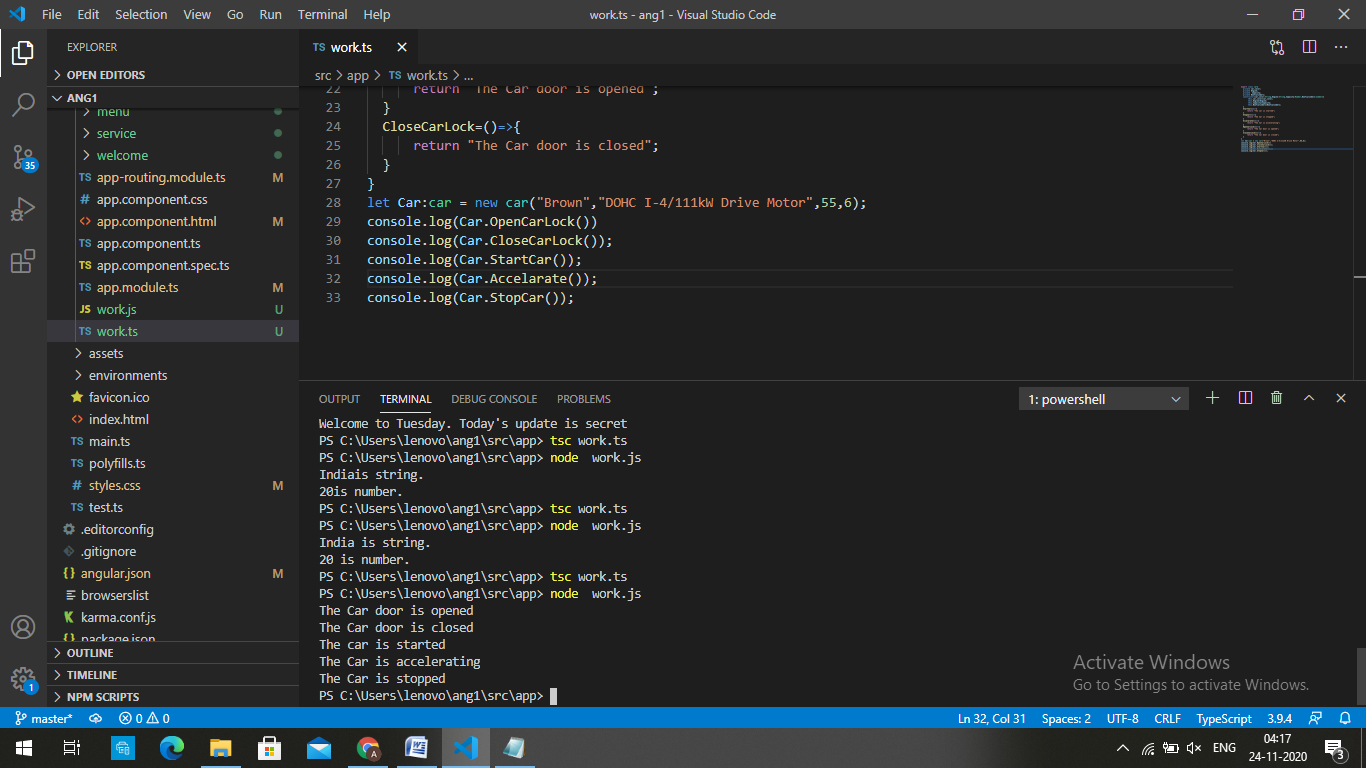
console.log(Car.CloseCarLock());

console.log(Car.StartCar());

console.log(Car.Accelarate());

console.log(Car.StopCar());

Output:



7. Covert the above Car class into Abstract class, and create following child classes :- SHV,HatchBack,Sedan. And create abstract methods in the base Car class, such as :- StartCar(),StopCar(). And create specific methods and behaviors in the related child classes.

abstract class car{

private car\_color;

private Engine;

private Capacity;

private Noofcylinders;

constructor(car\_color:String,Engine:String,Capacity:Number,Noofcylinders:number){

this.car\_color=car\_color;

this.Engine=Engine;

this.Capacity=Capacity;

this.Noofcylinders=Noofcylinders;

}

Accelerate=()=>{

return `The car is accelerating through ${this.Engine}`;

}

OpenCarLock=()=>{

return "The car door is opened";

}

CloseCarLock=()=>{

return "The car door is closed";

}

abstract startcar(): string;

abstract stopcar(): string;

}

class shv extends car {

private carname:String;

constructor(carname:String,car\_color:String,Engine:String,Capacity:Number,Noofcylinders:number){

super(car\_color,Engine,Capacity,Noofcylinders);

this.carname=carname;

}

startcar():string{

return `The ${this.carname} is started`;

}

stopcar(): string{

return `The ${this.carname} is stopped`;

}

}

class sedan extends car{

private carname:String;

constructor(carname:String,car\_color:String,Engine:String,Capacity:Number,Noofcylinders:number){

super(car\_color,Engine,Capacity,Noofcylinders);

this.carname=carname;

}

startcar():string{

return `The ${this.carname} is started`;

}

stopcar(): string{

return `The ${this.carname} is stopped`;

}

}

class Hatchback extends car{

private carname:String;

constructor(carname:String,car\_color:String,Engine:String,Capacity:Number,Noofcylinders:number){

super(car\_color,Engine,Capacity,Noofcylinders);

this.carname=carname;

}

startcar():string{

return `The ${this.carname} is started`;

}

stopcar(): string{

return `The ${this.carname} is stopped`;

}

}

let Car:car = new shv("shv","blue","nd-23",25,30);

console.log(Car.OpenCarLock());

console.log(Car.startcar());

console.log(Car.Accelerate());

console.log(Car.stopcar());

console.log(Car.CloseCarLock());

console.log(' ');

let Car1:car = new sedan("sedan","white","Ng-23",31,39);

console.log(Car1.OpenCarLock());

console.log(Car1.startcar());

console.log(Car1.Accelerate());

console.log(Car1.stopcar());

console.log(Car1.CloseCarLock());

console.log(' ');

let Car2:car = new Hatchback("Hatchback","Black","M-23",15,27);

console.log(Car2.OpenCarLock());

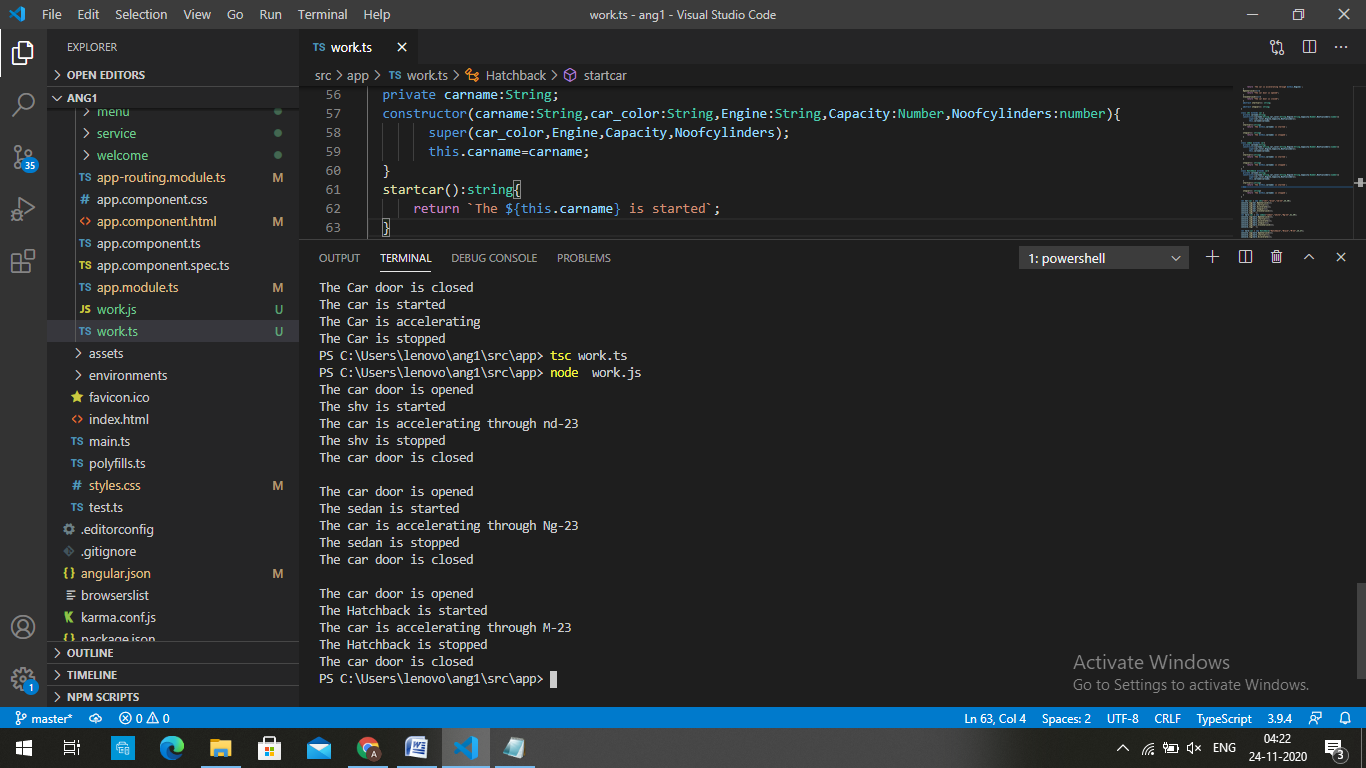
console.log(Car2.startcar());

console.log(Car2.Accelerate());

console.log(Car2.stopcar());

console.log(Car2.CloseCarLock());

Output:



8. Create a Interface Payment manager, that has following abstract methods:- 1) public string doPayment(paymentcreds:string), 2) public string getPaymentStatus(refNumber : string) . And create following 2 implemented classes :- a)UPIPaymentManagerImpl , b) CreditCardPaymentManagerImpl

interface Payment{

doPayment:(String)=>String;

getPaymentStatus:(String)=>String

}

class UPIPaymentManagerImpl implements Payment{

refnum:String;

paymentCredits:String;

doPayment(paymentCredits:String){

this.paymentCredits=paymentCredits;

return `The ${this.paymentCredits} is credited to your account`

}

getPaymentStatus(refnum:String){

this.refnum=refnum;

return `The ${refnum} reference number payment is successfully done`;

}

}

class CreditCardPaymentManagerImpl implements Payment{

refnum:String;

paymentCredits:String;

doPayment(paymentCredits:String){

this.paymentCredits=paymentCredits;

return `The ${this.paymentCredits} is credited to your account`

}

getPaymentStatus(refnum:String){

this.refnum=refnum;

return `The ${refnum} reference number payment is successfully done`;

}

}

let e:Payment=new UPIPaymentManagerImpl();

console.log(e.doPayment("50,000"));

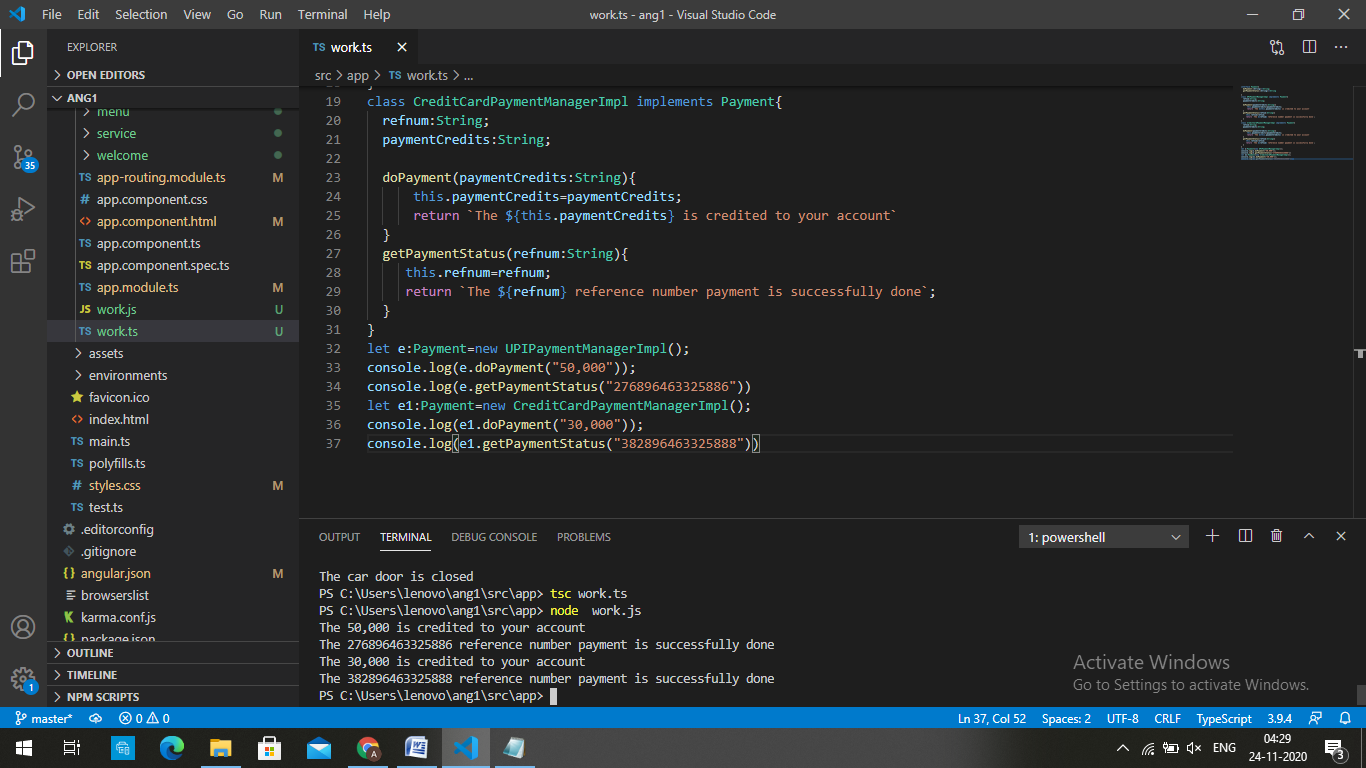
console.log(e.getPaymentStatus("276896463325886"))

let e1:Payment=new CreditCardPaymentManagerImpl();

console.log(e1.doPayment("30,000"));

console.log(e1.getPaymentStatus("382896463325888"))

Output:



9. Develop a program that exhibits duck typing in Typescript

class Dog {

sound = "barking";

}

class Cat {

sound = "Meow";

}

class Rat {

sound = "Squeak";

}

let cat: Cat = new Dog();

console.log(cat.sound);

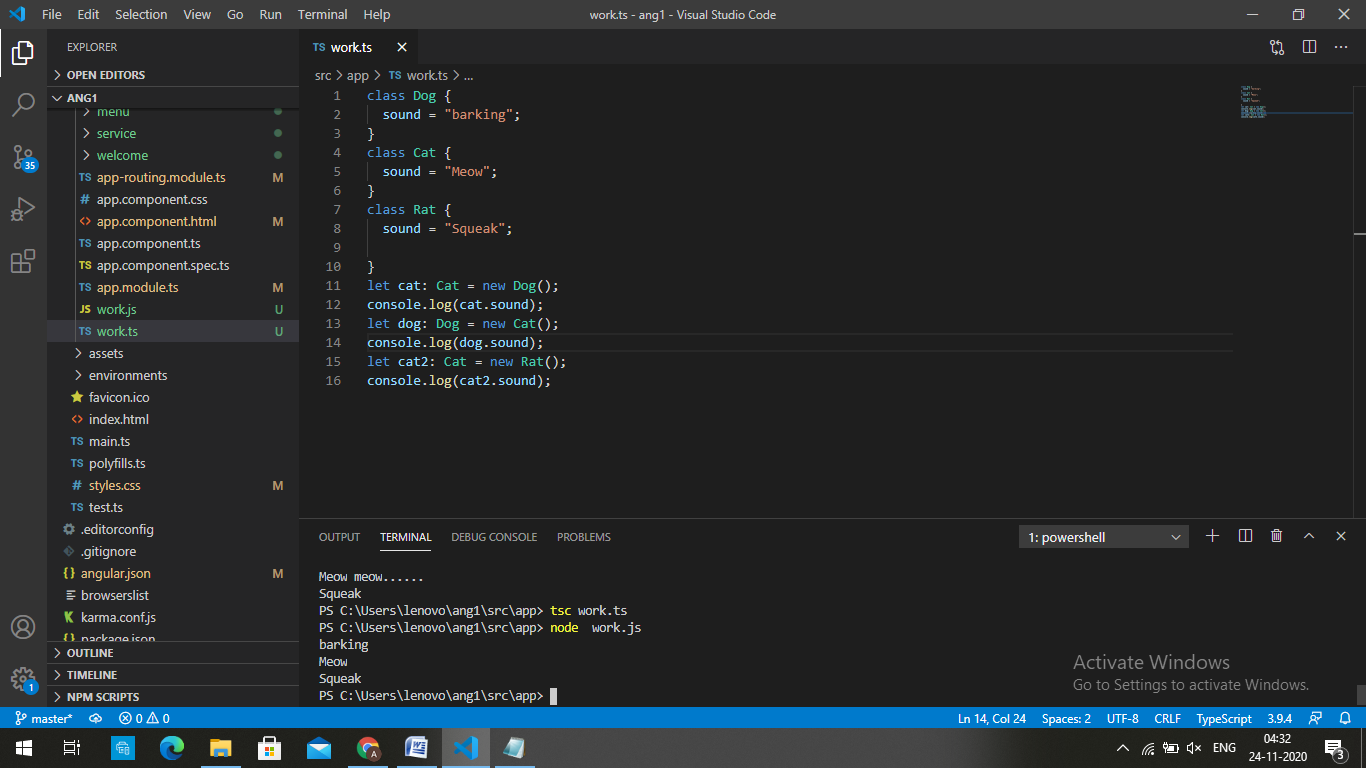
let dog: Dog = new Cat();

console.log(dog.sound);

let cat2: Cat = new Rat();

console.log(cat2.sound);

Output:



10. Develop a program that will exhibit functions as :- a) Function with default parameter, b)Function with optional parameter, c) Function with Rest Parameter

//Function with default parameters

function Attendance(name: string, presence: string = "Present") : string {

return name +" is " +presence + ' ' + '!';

}

console.log(Attendance('Ali'));

console.log(Attendance('divya', 'Absent'));

console.log(Attendance('Jin'));

//Functions with optional parameters

function Attend(presense: string, name?: string ) : string {

return name+" is "+presense + "!";

}

console.log(Attend('Present','Radha'));

console.log(Attend('Present'));

//Function with rest parameters

function Attendies(presense: string, ...names: string[]) {

return names.join(", ")+" are "+presense + "!";

}

console.log(Attendies("Present", "Srinu", "Nani"));

console.log(Attendies("Present"));

Output:

