Lab: Defining Classes

Problems for the "C# Advanced" course @ Software University

You can check your solutions in Judge

1. Car

Create a public **class** named **Car** within the namespace **`CarManufacturer`**:

Car.cs

```
namespace CarManufacturer
{
    class Car
    {
        // TODO: define the Car class members here …
    }
}
```

Define in the above class **private fields** for:

- **`make: string`**
- **`model: string`**
- **`year: int`**

The class should also have **public properties** for:

- **Make: string**
- **Model: string**
- **Year: int**

Create a public class **StartUp** class within the same namespace **CarManufacturer** to hold your program's entry point:

StartUp.cs
```
namespace CarManufacturer
{
    public class StartUp
    {
        static void Main()
        {
            // TODO: define the Main() method here ...
        }
    }
}
```

You should be able to use your **Car** class like this:

```
public static void Main(string[] args)
{
    Car car = new Car();

    car.Make = "VW";
    car.Model = "MK3";
    car.Year = 1992;

    Console.WriteLine($"Make: {car.Make}\nModel: {car.Model}\nYear: {car.Year}");
}
```

2. Car Extension

NOTE: You need a **StartUp** class with the namespace **CarManufacturer**.

Create a class **Car** (you can use the **class** from **the previous task**).

The class should have private fields for:

- **make: string**
- **model: string**
- **year: int**
- **fuelQuantity: double**
- **fuelConsumption: double**

The class should also have properties for:

- **Make: string**
- **Model: string**
- **Year: int**
- **FuelQuantity: double**
- **FuelConsumption: double**

The class should also have methods for:

- **`Drive(double distance): void`** – This method checks if the car fuel quantity minus the distance multiplied by the car fuel consumption is bigger than zero. If it is removed from the fuel quantity, the result of the multiplication between the distance and the fuel consumption. Otherwise, write on the console the following message:
  `"Not enough fuel to perform this trip!"`.
- **`WhoAmI(): string`** – returns the following message:
  ```
  "Make: {this.Make}
   Model: {this.Model}
   Year: {this.Year}
   Fuel: {this.FuelQuantity:F2}"
  ```

You should be able to use the class like this:

```
public static void Main(string[] args)
{
    Car car = new Car();

    car.Make = "VW";
    car.Model = "MK3";
    car.Year = 1992;
    car.FuelQuantity = 200;
    car.FuelConsumption = 200;
    car.Drive(2000);
    Console.WriteLine(car.WhoAmI());
}
```

3. Car Constructors

Using the class from the previous problem create one parameterless constructor with default values:

- **`Make – VW`**
- **`Model – Golf`**
- **`Year – 2025`**
- **`FuelQuantity – 200`**
- **`FuelConsumption – 10`**

Create a second constructor accepting **`make`**, **`model`** and **`year`** upon initialization and call the base constructor with its default values for **`fuelQuantity`** and **`fuelConsumption`**.

```
public Car(string make, string model, int year)
: this()
{
    this.Make = make;
    this.Model = model;
    this.Year = year;
}
```

Create a third constructor accepting **`make`**, **`model`**, **`year`**, **`fuelQuantity`** and **`fuelConsumption`** upon initialization and reuse the second constructor to set the make, model and year values.

```
public Car(string make, string model, int year, double fuelQuantity, double fuelConsumption)
: this(make, model, year)
{
    this.FuelQuantity = fuelQuantity;
    this.FuelConsumption = fuelConsumption;
}
```

Go to **StartUp.cs** file and make 3 different instances of the **class Car**, using the **different** overloads of the constructor.

```
public static void Main(string[] args)
{
    string make = Console.ReadLine();
    string model = Console.ReadLine();
    int year = int.Parse(Console.ReadLine());
    double fuelQuantity = double.Parse(Console.ReadLine());
    double fuelConsumption = double.Parse(Console.ReadLine());

    Car firstCar = new Car();
    Car secondCar = new Car(make, model, year);
    Car thirdCar = new Car(make, model, year, fuelQuantity,
      fuelConsumption);
}
```

4. Car Engine and Tires

Using the Car class, you already created, define another class **Engine**.

The class should have private fields for:

- **horsePower: int**
- **cubicCapacity: double**

The class should also have properties for:

- **HorsePower: int**
- **CubicCapacity: double**

The class should also have a constructor, which accepts **horsepower** and **cubicCapacity** upon initialization:

```
public Engine(int horsePower, double cubicCapacity)
{
    this.HorsePower = horsePower;
    this.CubicCapacity = cubicCapacity;
}
```

Now create a class **Tire**.

The class should have private fields for:

- **year: int**
- **pressure: double**

The class should also have properties for:

- **Year: int**
- **Pressure: double**

The class should also have a constructor, which accepts **year** and **pressure** upon initialization:

```
public Tire(int year, double pressure)
{
    this.Year = year;
    this.Pressure = pressure;
}
```

Finally, go to the **Car** class and create **private fields** and **public properties** for **Engine** and **Tire[]**.Create another constructor, which accepts **make, model, year, fuelQuantity, fuelConsumption, Engine** and **Tire[]** upon initialization:

```
public Car(string make, string model, int year, double fuelQuantity, double fuelConsumption,
  Engine engine, Tire[] tires)
    : this(make, model, year, fuelQuantity, fuelConsumption)
{
    this.Engine = engine;
    this.Tires = tires;
}
```

You should be able to use the classes like this:

```
public static void Main(string[] args)
{
    var tires = new Tire[4]
    {
        new Tire(1, 2.5),
        new Tire(1, 2.1),
        new Tire(2, 0.5),
        new Tire(2, 2.3),
    };

    var engine = new Engine(560, 6300);

    var car = new Car("Lamborghini", "Urus", 2010, 250, 9, engine, tires);
}
```

5. Special Cars

This is the final and most interesting problem in this lab. Until you receive the command **"No more tires"**, you will be given tire info in the format:

{year} {pressure}

{year} {pressure}

…

"No more tires"

You have to collect all the tires provided. Next, until you receive the command **"Engines done"** you will be given engine info and you also have to collect all that info.

{horsePower} {cubicCapacity}

{horsePower} {cubicCapacity}

…

The final step - until you receive **"Show special"**, you will be given information about cars in the format:

{make} {model} {year} {fuelQuantity} {fuelConsumption} {engineIndex} {tiresIndex}

…

Every time you have to create a **new Car** with the information provided. The car engine is the provided **engineIndex** and the tires are **tiresIndex**. Finally, collect all the created cars. When you receive the command **"Show special"**, drive 20 kilometers all the cars, which were manufactured during 2017 or after, have horsepower above 330 and the sum of their tire pressure is between 9 and 10. Finally, print information about each special car in the following format:

"Make: {specialCar.Make}"

"Model: {specialCar.Model}"

"Year: {specialCar.Year}"

"HorsePowers: {specialCar.Engine.HorsePower}"

"FuelQuantity: {specialCar.FuelQuantity}"

Input	Output
2 2.6 3 1.6 2 3.6 3 1.6 1 3.3 2 1.6 5 2.4 1 3.2 No more tires 331 2.2 145 2.0 Engines done Audi A5 2017 200 12 0 0 BMW X5 2007 175 18 1 1 Show special	Make: Audi Model: A5 Year: 2017 HorsePowers: 331 FuelQuantity: 197.6

Exercise: Defining Classes

Problems for the "C# Advanced" course @ Software University
You can check your solutions in Judge

1. Define a Class Person

NOTE: You need a **StartUp** class with the namespace **DefiningClasses**.

Define a class **Person** with **private** fields for **name** and **age** and **public** properties **Name** and **Age**.

Bonus*

Try to create a few objects of type Person:

Name	Age
Peter	20

George	18
Jose	43

NOTE: Use both **the inline initialization** and **the default constructor**.

2. Creating Constructors

NOTE: You need a **`StartUp`** class with the namespace **`DefiningClasses`**.

Add **3** constructors to the **`Person`** class from the last task. Use constructor chaining to reuse code:

- The **first** should take **no arguments** and produce a person with the name "**`No name`**" and **`age = 1`**.
- The **second** should accept only an integer **number** for the **`age`** and produce a person with the name "**`No name`**" and **`age`** equal to the passed **parameter**.
- The **third** one should accept a **string** for the **`name`** and an integer for the **`age`** and should produce a person with the given **`name`** and **`age`**.

3. Oldest Family Member

Use your **`Person` class** from the previous tasks. Create a class **`Family`**. The class should have a **list of people**, a method for adding members - **`void AddMember(Person member)`** and a method returning the oldest family member – **`Person GetOldestMember()`**. Write a program that reads the names and ages of **N** people and **adds them to the family**. Then **print** the **name** and **age** of the oldest member.

Examples

Input	Output
3 Peter 3 George 4 Annie 5	Annie 5
5 Steve 10 Christopher 15 Annie 4 Ivan 35 Maria 34	Ivan 35

4. Opinion Poll

Using the **`Person`** class, write a program that reads from the console **N** lines of personal information and then prints all people, whose **age** is **more than 30** years, **sorted in alphabetical order**.

Examples

Input	Output
3 Peter 12 Sam 31 Ivan 48	Ivan - 48 Sam - 31
5 Niki 33 Yord 88 Teo 22 Lily 44 Stan 11	Lily - 44 Niki - 33 Yord - 88

5. *Date Modifier

Create a class **`DateModifier`**, which stores the difference of the days between two dates. It should have a method that takes **two string parameters, representing dates** as strings and **calculates** the difference in the days between them.

Examples

Input	Output
1992 05 31 2016 06 17	8783
2016 05 31	42

2016 04 19	

6. Speed Racing

Create a program that keeps track of **cars** and their **fuel** and supports methods for **moving** the cars. Define a class **`Car`**. Each Car has the following properties:

- **`string Model`**
- **`double FuelAmount`**
- **`double FuelConsumptionPerKilometer`**
- **`double Travelled distance`**

A car's model is **unique** - there will never be 2 cars with the same model. On the first line of the input, you will receive a number **N** – the **number** of **cars** you need to track. On each of the next **N** lines, you will receive information about a car in the following format:

`"{model} {fuelAmount} {fuelConsumptionFor1km}"`

All **cars start at 0 kilometers traveled**. After the **N** lines, until the command **`"End"`** is received, you will receive commands in the following format:

`"Drive {carModel} {amountOfKm}"`

Implement a method in the **`Car`** class to calculate whether or not a car can **move** that **distance**. If it can, the car's **`fuel amount`** should be **reduced** by the amount of **used fuel** and its **`traveled distance`** should be increased by the number of the **traveled kilometers**. Otherwise, the car should not move (its fuel amount and the traveled distance should stay the same) and you should print on the console:

`"Insufficient fuel for the drive"`

After the **`"End"`** command is received, print **each car** and its **current fuel amount** and the **traveled distance** in the format:

`"{model} {fuelAmount} {distanceTraveled}"`

Print the fuel amount formatted **two digits** after the decimal separator.

Examples

Input	Output
2 AudiA4 23 0.3 BMW-M2 45 0.42 Drive BMW-M2 56 Drive AudiA4 5 Drive AudiA4 13 End	AudiA4 17.60 18 BMW-M2 21.48 56

2	`Insufficient fuel for the drive`
	`Insufficient fuel for the drive`
	`AudiA4 1.00 50`
	`BMW-M2 33.00 0`
	`Ferrari-488Spider 4.41 97`

7. Raw Data

Create a program that tracks **cars** and their **cargo**.

Start by defining a class **`Car`** that holds information about:

- **`Model`: a `string` property**
- **`Engine`: a `class`** with **two properties – speed** and **power,**
- **`Cargo`: a `class`** with **two properties – type** and **`weight`**
- **`Tires`:** a **collection of exactly 4 tires**. Each tire should have **two properties**: **age** and **`pressure`**.

Create a **constructor** that receives all of the information about the **`Car`** and creates and **initializes the `model` and** its inner **components** (**engine**, **cargo** and **`tires`**).

Input

On the first line of input, you will receive a number **N** representing the number of cars you have.

1. On the next **N** lines, you will receive information about each car in the format:

`"{model} {engineSpeed} {enginePower} {cargoWeight} {cargoType} {tire1Pressure} {tire1Age} {tire2Pressure} {tire2Age} {tire3Pressure} {tire3Age} {tire4Pressure} {tire4Age}"`

   - The **speed**, **power**, **weight** and **tire age** are **integers.**
   - The **tire pressure** is a **floating point number.**

2. Next, you will receive a single line with one of the following commands: **`"fragile"`** `or` **`"flammable"`**.

Output

As an output, if the command is:

- **`"fragile"`** - print **all cars,** whose **cargo** is **`"fragile"`** and **have a pressure of a single tire < 1.**
- **`"flammable"`** - print **all cars**, whose **cargo** is **`"flammable"`** and have **engine power > 250.**

The cars should be printed in order of appearing in the input.

Examples

Input	Output

`2`
`ChevroletAstro 200 180 1000 fragile 1.3 1 1.5 2 1.4 2 1.7 4`
`Citroen2CV 190 165 1200 fragile 0.9 3 0.85 2 0.95 2 1.1 1`
`fragile`
	`Citroen2CV`

	`ChevroletExpress` `DaciaDokker`

4 ChevroletExpress 215 255 1200 flammable 2.5 1 2.4 2 2.7 1 2.8 1 ChevroletAstro 210 230 1000 flammable 2 1 1.9 2 1.7 3 2.1 1 DaciaDokker 230 275 1400 flammable 2.2 1 2.3 1 2.4 1 2 1 Citroen2CV 190 165 1200 fragile 0.8 3 0.85 2 0.7 5 0.95 2 flammable	

8. Car Salesman

Define two classes **`Car`** and **`Engine`.**

Start by defining a class **`Car`** that holds information about:

- **`Model:`** a **`string` property**
- **`Engine:`** a **property holding the engine object**
- **`Weight`**: an **`int` property, it is optional**
- **`Color:`** a **`string` property, it is optional**

Next, the **`Engine` class** has the following properties:

- **`Model:`** a **`string` property**
- **`Power:`** an **`int` property**
- **`Displacement:`** an **`int` property, it is optional**
- **`Efficiency:`** a **`string` property, it is optional**

Input

1. On the first line, you will read a number **N,** which will specify how many lines of **engines** you will receive.
   - On each of the next **N** lines, you will receive information about an **`Engine`** in the following format: **`"{model} {power} {displacement} {efficiency}"`**
   - Keep in mind that **`"displacement"` and `"efficiency"`** are optional**,** they **could be missing** from the command.
2. Next, you will receive a number **`M`,** which will specify how many lines of **car** you will receive.
   - On each of the next **`M`** lines, you will receive information about a **`Car`** in the following format: **`"{model} {engine} {weight} {color}"`**.
   - Keep in mind that **`"weight"` and `"color"` are optional,** they could **be missing** from the command.
   - The **`"engine"`** will always be the model of an existing **`Engine`**.
   - When creating the object for a **`Car`**, you should keep a **reference to the real engine** in it, instead of just the engine's model.

Note: The optional properties **might be missing** from the formats.

Output

Your task is to **print** all the **cars** in the order they were received and their information in the format defined below. If any of the optional fields are missing, print **`"n/a"`** in its place:

```
"{CarModel}:
  {EngineModel}:
    Power: {EnginePower}
    Displacement: {EngineDisplacement}
    Efficiency: {EngineEfficiency}
  Weight: {CarWeight}
  Color: {CarColor}"
```

Bonus*

Override the classes' "**`ToString()`**" methods to have a reusable way of displaying the objects.

Examples

Input	Output

`2`
`V8-101 220 50`
`V4-33 140 28 B`
`3`
`FordFocus V4-33 1300 Silver`
`FordMustang V8-101`
`VolkswagenGolf V4-33 Orange`
	`FordFocus:`
	`  V4-33:`
	`    Power: 140`
	`    Displacement: 28`
	`    Efficiency: B`
	`  Weight: 1300`
	`  Color: Silver`
	`FordMustang:`
	`  V8-101:`
	`    Power: 220`
	`    Displacement: 50`
	`    Efficiency: n/a`
	`  Weight: n/a`
	`  Color: n/a`
	`VolkswagenGolf:`
	`  V4-33:`
	`    Power: 140`
	`    Displacement: 28`
	`    Efficiency: B`
	`  Weight: n/a`
	`  Color: Orange`

`4`
`DSL-10 280 B`
`V7-55 200 35`
`DSL-13 305 55 A+`
`V7-54 190 30 D`
`4`
`FordMondeo DSL-13 Purple`
`VolkswagenPolo V7-54 1200 Yellow`
`VolkswagenPassat DSL-10 1375 Blue`
`FordFusion DSL-13`
	`FordMondeo:`
	`  DSL-13:`
	`    Power: 305`
	`    Displacement: 55`
	`    Efficiency: A+`
	`  Weight: n/a`
	`  Color: Purple`
	`VolkswagenPolo:`
	`  V7-54:`
	`    Power: 190`
	`    Displacement: 30`
	`    Efficiency: D`
	`  Weight: 1200`
	`  Color: Yellow`
	`VolkswagenPassat:`
	`  DSL-10:`

	```
	    Power: 280
	    Displacement: n/a
	    Efficiency: B
	  Weight: 1375
	  Color: Blue
	FordFusion:
	  DSL-13:
	    Power: 305
	    Displacement: 55
	    Efficiency: A+
	  Weight: n/a
	  Color: n/a
	```

9. Pokemon Trainer

Define a class **Trainer** and a class **Pokemon**.

Trainers have:

- **Name**
- **Number of badges**
- **A collection of pokemon**

Pokemon have:

- **Name**
- **Element**
- **Health**

All values are **mandatory**. Every Trainer **starts with 0 badges**.

You will be receiving lines until you receive the command **"Tournament"**. Each line will carry information about a pokemon and the trainer who caught it in the format:

"{trainerName} {pokemonName} {pokemonElement} {pokemonHealth}"

TrainerName is the name of the Trainer who caught the pokemon. Trainers' names are **unique**. After receiving the command **"Tournament"**, you will start receiving commands until the **"End"** command is received. They can contain one of the following:

- **"Fire"**
- **"Water"**
- **"Electricity"**

For every command, you must check if a trainer has at least 1 pokemon with the given element. If he does, he receives 1 badge. Otherwise, all of his pokemon **lose 10 health**. If a pokemon falls **to 0 or less health**, **he dies** and must be deleted from the trainer's collection. In the end, you should print all of the trainers, **sorted by the number of badges they have in descending order** (if two trainers have the same amount of badges, they should be sorted by order of appearance in the input) in the format:

"{trainerName} {badges} {numberOfPokemon}"

Examples

Input	Output
Peter Charizard Fire 100 George Squirtle Water 38 Peter Pikachu Electricity 10 Tournament Fire Electricity End	Peter 2 2 George 0 1
Sam Blastoise Water 18 Narry Pikachu Electricity 22 John Kadabra Psychic 90 Tournament Fire Electricity Fire End	Narry 1 1 Sam 0 0 John 0 1

10. SoftUni Parking

Preparation

Download the skeleton provided in Judge. **Do not** change the **StartUp** class or its **namespace**.

Problem Description

Your task is to create a repository, which stores cars by creating the classes described below.

First, write a C# class **Car** with the following properties:

- **Make: string**
- **Model: string**
- **HorsePower: int**
- **RegistrationNumber: string**

```
public class Car
{
    // TODO: implement this class
}
```

The class' **constructor** should receive **make, model, horsePower** and **registrationNumber** and override the **ToString()** method in the following format:

"Make: {make}"

"Model: {model}"

`"HorsePower: {horse power}"`

`"RegistrationNumber: {registration number}"`

Create a C# class **`Parking`** that has **`Cars`** (a collection that stores the entity **`Car`**). All entities inside the class have the **same properties**.

```
public class Parking
{
    // TODO: implement this class
}
```

The class' **constructor** should initialize the **`Cars`** with a new instance of the collection and accept **capacity** as a parameter**.**

Implement the following fields:

- Field **`cars`** – a **collection** that holds added cars.
- Field **`capacity`** – accessed only by the base class (responsible for the parking capacity).

Implement the following **methods**:

`AddCar(Car Car)`

The method first checks if there is already a car with the provided car registration number and if there is, the method returns the following message:

`"Car with that registration number, already exists!"`

Next check if the count of the cars in the parking is more than the capacity and if it returns the following message:

`"Parking is full!"`

Finally, if nothing from the previous conditions is true, it just adds the current car to the cars in the parking and returns the message:

`"Successfully added new car {Make} {RegistrationNumber}"`

`RemoveCar(string RegistrationNumber)`

Removes a car with the given registration number. If the provided registration number does not exist returns the message:

`"Car with that registration number, doesn't exist!"`

Otherwise, removes the car and returns the message:

`"Successfully removed {registrationNumber}"`

`GetCar(string RegistrationNumber)`

Returns the **`Car`** with the provided registration number.

`RemoveSetOfRegistrationNumber(List<string> RegistrationNumbers)`

A void method, which removes all cars that have the provided registration numbers. Each car is removed only if the registration number exists.

And the following **property**:

- **`Count -`** Returns the number of stored cars.

Examples

This is an example of how the **Parking** class is **intended to be used**.

Sample code usage

```
var car = new Car("Skoda", "Fabia", 65, "CC1856BG");
var car2 = new Car("Audi", "A3", 110, "EB8787MN");

Console.WriteLine(car.ToString());
// Make: Skoda
// Model: Fabia
// HorsePower: 65
// RegistrationNumber: CC1856BG

var parking = new Parking(5);
Console.WriteLine(parking.AddCar(car));
// Successfully added new car Skoda CC1856BG

Console.WriteLine(parking.AddCar(car));
// Car with that registration number, already exists!

Console.WriteLine(parking.AddCar(car2));
// Successfully added new car Audi EB8787MN

Console.WriteLine(parking.GetCar("EB8787MN").ToString());
// Make: Audi
// Model: A3
// HorsePower: 110
// RegistrationNumber: EB8787MN

Console.WriteLine(parking.RemoveCar("EB8787MN"));
// Successfullyremoved EB8787MN

Console.WriteLine(parking.Count);
// 1
```

Submission

Zip all the files in the project folder except **`bin`** and **`obj`** folders.