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.











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

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

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.

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.

Input	Output
2	AudiA4 17.60 18
AudiA4 23 0.3	BMW-M2 21.48 56
BMW-M2 45 0.42	
Drive BMW-M2 56	
Drive AudiA4 5	
Drive AudiA4 13	
End	
3	Insufficient fuel for the drive
AudiA4 18 0.34	Insufficient fuel for the drive
BMW-M2 33 0.41	AudiA4 1.00 50
Ferrari-488Spider 50 0.47	BMW-M2 33.00 0
Drive Ferrari-488Spider 97	Ferrari-488Spider 4.41 97















Drive Ferrari-488Spider 35	
Drive AudiA4 85	
Drive AudiA4 50	
End	

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.

Input	Output
2	Citroen2CV
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	
4	ChevroletExpress
ChevroletExpress 215 255 1200 flammable 2.5 1 2.4 2 2.7 1 2.8 1	DaciaDokker
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.









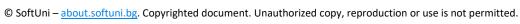






Input	Output
2	FordFocus:
V8-101 220 50	V4-33:
V4-33 140 28 B	Power: 140
3	Displacement: 28
FordFocus V4-33 1300 Silver	Efficiency: B
FordMustang V8-101	Weight: 1300
VolkswagenGolf V4-33 Orange	Color: Silver
	FordMustang:
	V8-101:
	Power: 220
	Displacement: 50
	Efficiency: n/a
	Weight: n/a
	Color: n/a
	VolkswagenGolf: V4-33:
	V4-33: Power: 140
	Displacement: 28
	Efficiency: B Weight: n/a
	Color: Orange
	-
4	FordMondeo:
DSL-10 280 B	DSL-13:
V7-55 200 35	Power: 305
DSL-13 305 55 A+	Displacement: 55
V7-54 190 30 D	Efficiency: A+
4	Weight: n/a
FordMondeo DSL-13 Purple	Color: Purple
VolkswagenPolo V7-54 1200 Yellow	VolkswagenPolo:
VolkswagenPassat DSL-10 1375 Blue	V7-54:
FordFusion DSL-13	Power: 190
	Displacement: 30
	Efficiency: D
	Weight: 1200
	Color: Yellow
	VolkswagenPassat: DSL-10:
	Power: 280
	Displacement: n/a
	Efficiency: B
	Elliciency. D

















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}"

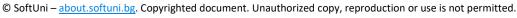
TrainerName is the name of the Trainer who caught the pokemon. Trainers' names are unique. After receiving the command "Tournament", you will start receiving commands until the "End" command is received. They can contain one of the following:

- "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}"

















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

10. SoftUni Parking

Preparation

Download the skeleton provided in <u>Judge</u>. **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

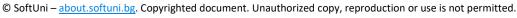
RegistrationNumber: string

```
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.











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.













