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