**C# References:**

## Student System

You are given a **working** **project** for a small **Student** **System**, but the code is very poorly organized. Break up the code **logically** into **smaller** **functional** **units** – **methods** and **classes** and don’t break the functionality.

The program supports the following commands:

* “**Create <studentName> <studentAge> <studentGrade>**”– creates a new student and adds them to the repository.
* “**Show <studentName>**”– prints on the console information about a student in the format:  
  “**<studentName> is <studentAge> years old. <commentary>**”, where the **commentary** is based on the student’s grade.
* “**Exit**” – closes the program.

**Do not** add any **extra validation** or **functionality** to the app!

### Examples

|  |  |
| --- | --- |
| **input** | **output** |
| Create Pesho 20 5.50  Create Mimi 18 4.50  Create Gosho 25 3  Show Pesho  Show Mimi  Exit | Pesho is 20 years old. Excellent student.  Mimi is 18 years old. Average student. |

## Hotel Reservation

Create a class **PriceCalculator** that calculates the total price of a holiday, given the **price** **per** **day**, **number** **of** **days**, the **season** and a **discount** **type**.The **discount** **type** and **season** should be **enums**.

Use the class in your **Main()** method to read input and **print** on the console the **price** of the **whole** **holiday**.

The price per day will be multiplied depending on the season by:

* **1** during **Autumn**
* **2** during **Spring**
* **3** during **Winter**
* **4** during **Summer**

The discount is applied to the total price and is one of the following:

* 20% for VIP clients
* 10% for clients, visiting for a second time
* 0% if there is no discount

### Input

On a **single** **line** you will receive all the **information** about the **reservation** in the format:  
“**<pricePerDay> <numberOfDays> <season> <discountType>**”, where:

* The price per day will be a valid decimal in the range [0.01…1000.00]
* The number of days will be a valid integer in range [1…1000]
* The season will be one of: **Spring**, **Summer**, **Autumn**, **Winter**
* The discount will be one of: **VIP**, **SecondVisit**, **None**, but it **can** also **be** **omitted** from the input

### Output

On a **single** **line**, print the **total** **price** of the **holiday**, rounded to **2** **digits** after the decimal separator.

### Examples

|  |  |
| --- | --- |
| **input** | **output** |
| 50.25 5 Summer VIP | 804.00 |
| 40 10 Autumn SecondVisit | 360.00 |
| 120.20 2 Winter | 721.20 |

## Problem: Vehicles

Write a program that models 2 vehicles (**Car** and **Truck**) and simulates **driving** and **refueling** them. **Car** and **truck** both have **fuel quantity**, **fuel consumption** **in liters** **per km** and can be **driven a given distance** and **refueled with a given amount of fuel.** It’s summer, so both vehicles use air conditioners and their **fuel consumption** per km is **increased** by **0.9** liters for the **car** and with **1.6** liters for the **truck**. Also, the **truck** has a tiny hole in its tank and when its **refueled** it keeps only **95%** of the given **fuel**. The **car** has no problems and adds **all the given fuel to its tank.** If a vehicle cannot travel the given distance, its fuel does not change.

### Input

* On the first line – information about the car in the format: "Car {fuel quantity} {liters per km}"
* On the second line – info about the truck in the format: "Truck {fuel quantity} {liters per km}"
* On the third line – the number of commands N that will be given on the next N lines
* On the next N lines – commands in the format:
* "Drive Car {distance}"
* "Drive Truck {distance}"
* "Refuel Car {liters}"
* "Refuel Truck {liters}"

### Output

* After each Drive command, if there was enough fuel, print on the console a message in the format:
* "Car/Truck travelled {distance} km"
* If there was not enough fuel, print: "Car/Truck needs refueling"
* After the End command, print the remaining fuel for both the car and the truck, rounded to 2 digits after the floating point in the format:
* **"Car: {liters}"**
* **"Truck: {liters}"**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Car 15 0.3  Truck 100 0.9  4  Drive Car 9  Drive Car 30  Refuel Car 50  Drive Truck 10 | Car travelled 9 km  Car needs refueling  Truck travelled 10 km  Car: 54.20  Truck: 75.00 |
| Car 30.4 0.4  Truck 99.34 0.9  5  Drive Car 500  Drive Car 13.5  Refuel Truck 10.300  Drive Truck 56.2  Refuel Car 100.2 | Car needs refueling  Car travelled 13.5 km  Truck needs refueling  Car: 113.05  Truck: 109.13 |

## Vehicles Extension

Use your solution of the **previous** task for the starting point and add more functionality. Add a new vehicle – **Bus**. Add to every **vehicle** a new property – **tank** **capacity**. A vehicle cannot **start** **with** or **refuel** **above** its **tank** **capacity**.

If you **try to put more fuel** in the tank than the **available space,** print on the console **"Cannot fit {fuel amount} fuel in the tank"** and **do not add any fuel** in the vehicle’s tank. If you try to **create** a vehicle with **more** **fuel** than its **tank** **capacity**, **create** it but start with an **empty** **tank**.

Add a **new command** for the bus. You can **drive** the **bus** **with or without people**. **With people**, the **air-conditioner** **is turned on** and its **fuel consumption** per kilometer is **increased by 1.4 liters**. If there are **no people in the bus**, the air-conditioner is **turned off** and **does not increase** the fuel consumption.

Finally, add **validation** for the **amount** of **fuel** given to the **Refuel** **command** – if it is 0 or negative, print **"Fuel must be a positive number"**.

### Input

* On the **first** **three** **lines** you will receive information about the vehicles in the format:
* **"**Vehicle {initial fuel quantity} {liters per km} {tank capacity}**"**
* On the fourth line – the number of commands N that will be given on the next N lines
* On the next N lines – commands in format:
* **"**Drive Car {distance}**"**
* **"**Drive Truck {distance}**"**
* **"**Drive Bus {distance}**"**
* **"**DriveEmpty Bus {distance}**"**
* **"**Refuel Car {liters}**"**
* **"**Refuel Truck {liters}**"**
* **"**Refuel Bus {liters}**"**

### Output

* After each Drive command, if there was enough fuel, print on the console a message in the format:
* "Car/Truck travelled {distance} km"
* If there was not enough fuel, print:
* "Car/Truck needs refueling"
* If you try to refuel with an amount **≤ 0** print:
* "Fuel must be a positive number"
* If the given fuel cannot fit in the tank, print:
* "Cannot fit {fuel amount} fuel in the tank"
* After the End command, print the remaining fuel for all vehicles, rounded to 2 digits after the floating point in the format:
* **"Car: {liters}"**
* **"Truck: {liters}"**
* **"Bus: {liters}"**

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Car 30 0.04 70  Truck 100 0.5 300  Bus 40 0.3 150  8  Refuel Car -10  Refuel Truck 0  Refuel Car 10  Refuel Car 300  Drive Bus 10  Refuel Bus 1000  DriveEmpty Bus 100  Refuel Truck 1000 | Fuel must be a positive number  Fuel must be a positive number  Cannot fit 300 fuel in the tank  Bus travelled 10 km  Cannot fit 1000 fuel in the tank  Bus needs refueling  Cannot fit 1000 fuel in the tank  Car: 40.00  Truck: 100.00  Bus: 23.00 |

## Wild Farm

Your task is to create a **class** **hierarchy** like **described** **below**. The **Animal**, **Bird**, **Mammal**, **Feline** and **Food** classes should be **abstract**. Override the method **ToString()**.

* **Food – int Quantity;**
  + **Vegetable;**
  + **Fruit;**
  + **Meat;**
  + **Seeds;**
* **Animal – string Name, double Weight, int FoodEaten;**
  + **Bird – double WingSize;**
    - **Owl;**
    - **Hen;**
  + **Mammal – string LivingRegion;**
    - **Mouse;**
    - **Dog;**
    - **Feline;**
      * **Cat – string Breed;**
      * **Tiger – string Breed;**

All **animals** should also have the **ability** to ask for food by **producing** a **sound**.

* **Owl – "Hoot Hoot";**
* **Hen – "Cluck";**
* **Mouse – "Squeak";**
* **Dog – "Woof!";**
* **Cat – "Meow";**
* **Tiger – "ROAR!!!";**

Now use the **classes** you’ve created to **instantiate** some **animals** and **feed** **them**.  
Input should be read from the console. Every **even** line (starting from 0) will **contain** **information** about an **animal** in the following format:

* **Felines - "{Type} {Name} {Weight} {LivingRegion} {Breed}";**
* **Birds - "{Type} {Name} {Weight} {WingSize}";**
* **Mice and Dogs = "{Type} {Name} {Weight} {LivingRegion}";**

On the **odd** lines you will receive **information** about a piece of **food** that you should **give** to that **animal**. The line will consist of a **FoodType** and **quantity**, separated by a whitespace.

Animals will only eat a certain type of food, as follows:

* **Hens** eat **everything**;
* **Mice** eat **vegetables** and **fruits**;
* **Cats** east **vegetables** and **meat**;
* **Tigers**, **Dogs** and **Owls** eat **only** **meat**;

If you try to give an animal a different type of food, it will not eat it and you should print:

* **"{AnimalType} does not eat {FoodType}!"**

The **weight** of an **animal** will **increase** with **every** **piece** of **food** it **eats**, as follows:

* **Hen – 0.35;**
* **Owl – 0.25;**
* **Mouse – 0.10;**
* **Cat – 0.30;**
* **Dog – 0.40;**
* **Tiger – 1.00;**

Override the **ToString()** method to print the information about an animal in the formats:

* **Birds –** **"{AnimalType} [{AnimalName}, {WingSize}, {AnimalWeight}, {FoodEaten}]"**
* **Felines – "{AnimalType} [{AnimalName}, {Breed}, {AnimalWeight}, {AnimalLivingRegion}, {FoodEaten}]"**
* **Mice and Dogs –** **"{AnimalType} [{AnimalName}, {AnimalWeight}, {AnimalLivingRegion}, {FoodEaten}]"**

After you read **information** about an **animal** and some **food**, the **animal** will **produce a** **sound** (**print** it on the **console**). Next, you should **try** to **feed** it. After receiving the “**End**” command, **print** information about **every** **animal** in **order** of **input**.

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cat Pesho 1.1 Home Persian  Vegetable 4  End | Meow  Cat [Pesho, Persian, 2.3, Home, 4] |
| Tiger Typcho 167.7 Asia Bengal  Vegetable 1  Dog Doncho 500 Street  Vegetable 150  End | ROAR!!!  Tiger does not eat Vegetable!  Woof!  Dog does not eat Vegetable!  Tiger [Typcho, Bengal, 167.7, Asia, 0]  Dog [Doncho, 500, Street, 0] |
| Mouse Jerry 0.5 Anywhere  Fruit 1000  Owl Toncho 2.5 30  Meat 5  End | Squeak  Hoot Hoot  Mouse [Jerry, 100.5, Anywhere, 1000]  Owl [Toncho, 30, 3.75, 5] |

# Part I: Generics

## Box of T

Create a class **Box<>** that can store anything.

It should have two public methods:

* void Add(element)
* element Remove()
* int Count { get; }

Adding should add on top of its contents. Remove should get the topmost element.

### Examples

|  |
| --- |
| public static void Main(string[] args)  {  Box<int> box = new Box<int>();  box.Add(1);  box.Add(2);  box.Add(3);  Console.WriteLine(box.Remove());  box.Add(4);  box.Add(5);  Console.WriteLine(box.Remove());  } |

### Hints

Use the syntax Box<T> to create a generic class

## Generic Array Creator

Create a class ArrayCreator with a method and a single overload to it:

* static T[] Create(int length, T item)

The method should return an array with the given length and every element should be set to the given default item.

### Examples

|  |
| --- |
| static void Main(string[] args)  {  string[] strings = ArrayCreator.Create(5, "Pesho");  int[] integers = ArrayCreator.Create(10, 33);  } |

# Part II: Generic Constraints

## Generic Scale

Create a class **Scale<T>** that holds two elements - left and right. The scale should receive the elements through its single constructor:

* Scale(T left, T right)

The scale should have a single method:

* T GetHeavier()

The greater of the two elements is heavier. The method should return null if elements are equal.

# Exercises: Generics

## Generic Box

Create a generic class Box that can be initialized with **any** type and **store** the value. **Override** the **ToString()** method to print the type and the value of the data stored in the format **{class full name: value}.**

### Note

This problem does not have tests in Judge but instead, the class is used in the next problems.

In order to get a class' full name, use .GetType().FullName property. You can read more [here](https://msdn.microsoft.com/en-us/library/system.type.fullname(v=vs.110).aspx).

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 123123 | System.Int32: 123123 |
| life in a box | System.String: life in a box |

## Generic Box of String

Use the class that you've created for the previous problem and test it with the class **System.String.** On the first line, you will get **n** - the number of strings to read from the console. On the next **n** lines, you will get the actual strings. For each of them create a box and call its **ToString()** method to print its data on the console.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 2  life in a box  box in a life | System.String: life in a box  System.String: box in a life |

## Generic Box of Integer

Use the description of the previous problem but now, test your generic box with Integers.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  7  123  42 | System.Int32: 7  System.Int32: 123  System.Int32: 42 |

## Generic Swap Method Strings

Create a generic method that receives a list containing any type of data and swaps the elements at two given indexes.

As in the previous problems read **n** number of boxes of type String and add them to the list. On the next line, however you will receive a swap command consisting of two indexes. Use the method you've created to swap the elements that correspond to the given indexes and print each element in the list.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  Pesho  Gosho  Swap me with Pesho  0 2 | System.String: Swap me with Pesho  System.String: Gosho  System.String: Pesho |

## Generic Swap Method Integers

Use the description of the previous problem but now, test your list of generic boxes with Integers.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  7  123  42  0 2 | System.Int32: 42  System.Int32: 123  System.Int32: 7 |

## Generic Count Method Strings

Create a **method** that receives as argument a **list of any type that can be compared** and an **element of the given type**. The method should **return the count of elements that are greater than the value of the given element**. **Modify your Box class** to support **comparing by value** of the data stored.

On the first line, you will receive **n** - the number of elements to add to the list. On the next **n** lines, you will receive the actual elements. On the last line, you will get the value of the element to which you need to compare every element in the list.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  aa  aaa  bb  aa | 2 |

## Generic Count Method Doubles

Use the description of the previous problem but now, test your list of generic boxes with **doubles**.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 3  7.13  123.22  42.78  7.55 | 2 |

## Custom List

Create a generic data structure that can store **any** type that **can** be compared. Implement functions:

* **void Add(T element)**
* **T Remove(int index)**
* **bool Contains(T element)**
* **void Swap(int index1, int index2)**
* **int CountGreaterThan(T element)**
* **T Max()**
* **T Min()**

Create a command interpreter that reads commands and modifies the custom list that you have created. Set the custom list’s type to string. Implement the commands:

* **Add <element>** - Adds the given element to the end of the list
* **Remove <index>** - Removes the element at the given index
* **Contains <element>** - Prints if the list contains the given element **(True or False)**
* **Swap <index> <index>** - Swaps the elements at the given indexes
* **Greater <element>** - Counts the elements that are greater than the given element and prints their count
* **Max** - Prints the maximum element in the list
* **Min** - Prints the minimum element in the list
* **Print** - Prints all elements in the list, each on a separate line
* **END** - stops the reading of commands

There will **not** be any **invalid** input commands.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add aa  Add bb  Add cc  Max  Min  Greater aa  Swap 0 2  Contains aa  Print  END | cc  aa  2  True  cc  bb  aa |

## Custom List Sorter

Extend the previous problem by creating an additional **Sorter class**. It should have a single static **method** **Sort()** which can sort objects of type **CustomList** containing any type that can be compared. **Extend the command list** to support one additional command Sort:

* **Sort** - Sort the elements in the list in ascending order.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add cc  Add bb  Add aa  Sort  Print  END | aa  bb  cc |

## \*Custom List Iterator

For the print command, you have probably used a for loop. Extend your custom list class by making it to implement **IEnumerable<T>.** This should allow you to iterate your list in a foreach statement.

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Add aa  Add bb  Add cc  Max  Min  Greater aa  Swap 0 2  Print  END | cc  aa  2  cc  bb  aa |

## Tuple

There is something, really annoying in C#. It is called a [**Tuple**](https://msdn.microsoft.com/en-us/library/system.tuple(v=vs.110).aspx). It is a class, which may store a few objects but let’s focus on the type of Tuple which contains two objects. The first one is “**item1**” and the second one is “**item2**”. It is kind of like a **KeyValuePair** except – it **simply has items,** which are **neither key nor value**. The annoyance is coming from the fact, that you have no idea what these objects are holding. The class name is telling you nothing, the methods which it has – also. So, let’s say for some reason we would like to try to implement it by ourselves.

The task: Create a class “**Tuple**”, which is holding two objects. Like we said, the first one, will be “**item1**” and the second one - “**item2**”. The tricky part here is to make the class hold generics. This means, that when you create a new object of class - “**Tuple**”, there should be a way to explicitly, specify both the items type separately.

### Input

The input consists of three lines:

* The first one is holding a person name and an address. They are separated by space(s). Your task is to collect them in the tuple and print them on the console. Format of the input:

**<<first name> <last name>>** **<address>**

* The second line holds a **name** of a personand the **amount of beer** (int) he can drink. Format:

**<name> <liters of beer>**

* The last line will hold an **Integer** and a **Double**. Format:

**<Integer> <Double>**

### Output

* Print the tuples’ items in format: {**item1**} -> {**item2**}

### Constraints

Use the good practices we have learned. Create the class and make it have getters and setters for its class variables. The input will be valid, no need to check it explicitly!

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Sofka Tripova Stolipinovo  Az 2  23 21.23212321 | Sofka Tripova -> Stolipinovo  Az -> 2  23 -> 21.23212321 |

## Threeuple

Create a Class **Threeuple**. Its name is telling us, that it will hold no longer, just a pair of objects. The task is simple, our **Threeuple** should **hold three objects**. Make it have getters and setters. You can even extend the previous class

### Input

The input consists of three lines:

* The first one is holding a name, an address and a town. Format of the input:

**<<first name> <last name>> <address> <town>**

* The second line is holding a name, beer liters, and a Boolean variable with value - **drunk** or **not**. Format:

**<name> <liters of beer> <drunk or not>**

* The last line will hold a name, a bank balance (double) and a bank name. Format:

**<name> <account balance> <bank name>**

### Output

* Print the Threeuples’ objects in format: {**firstElement**} -> {**secondElement**} -> {**thirdElement**}

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Sofka Tripova Stolipinovo Plovdiv  MitkoShtaigata 18 drunk  SashoKompota 0.10 NkqfaBanka | Sofka Tripova -> Stolipinovo -> Plovdiv  MitkoShtaigata -> 18 -> True  SashoKompota -> 0.1 -> NkqfaBanka |
| Ivan Ivanov Tepeto Plovdiv  Mitko 18 not  Sasho 0.10 NGB | Ivan Ivanov -> Tepeto -> Plovdiv  Mitko -> 18 -> False  Sasho -> 0.1 -> NGB |

### Note

You may extend your previous solution.

A close-up of text

Description automatically generated

A close-up of a white background

Description automatically generated

A black and white photo

Description automatically generated with medium confidence