Exercises: Encapsulation

You can check your solutions in **Judge system**: https://judge.softuni.bg/Contests/3163/Encapsulation

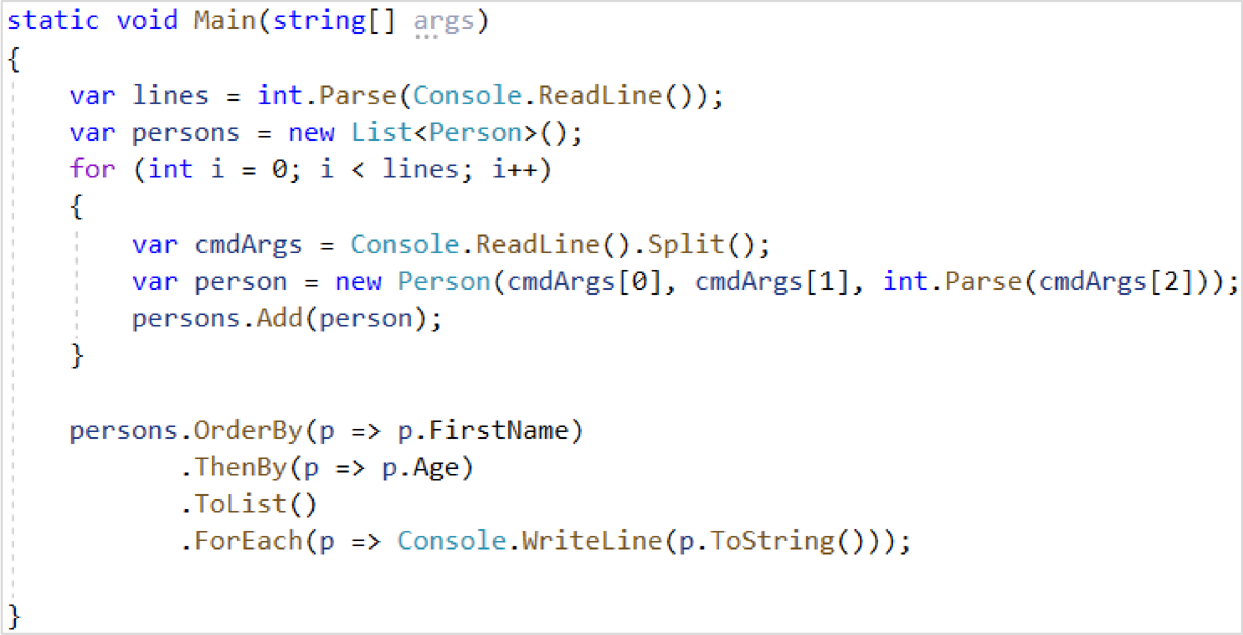
1. Sort Persons by Name and Age

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

Create a class **Person**, which should have **public** properties with **private** setters for:

- **FirstName: string**
- **LastName: string**
- **Age: int**
- **ToString(): string - override**

You should be able to use the class like this:



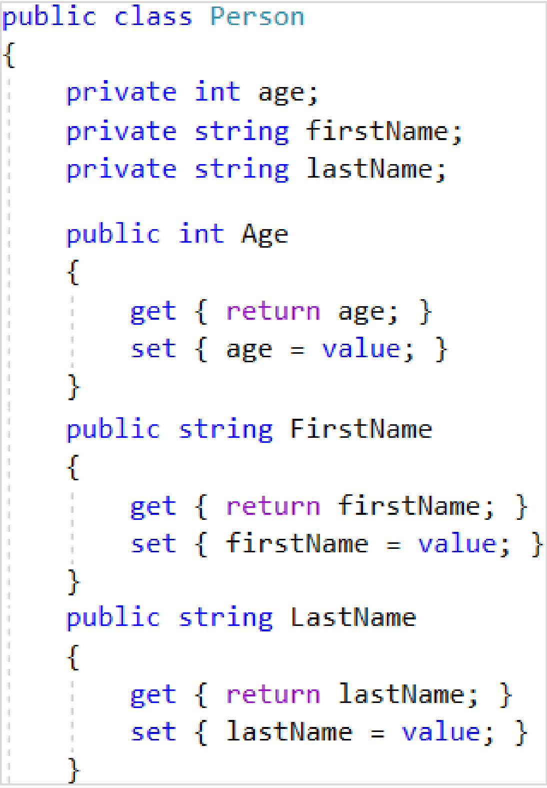
```
static void Main(string[] args)
{
    var lines = int.Parse(Console.ReadLine());
    var persons = new List<Person>();
    for (int i = 0; i < lines; i++)
    {
        var cmdArgs = Console.ReadLine().Split();
        var person = new Person(cmdArgs[0], cmdArgs[1], int.Parse(cmdArgs[2]));
        persons.Add(person);
    }

    persons.OrderBy(p => p.FirstName)
           .ThenBy(p => p.Age)
           .ToList()
           .ForEach(p => Console.WriteLine(p.ToString()));

}
```

Examples

Input	Output
5 Seth Nelson 65 Liam Scott 57 Brian Clark 27 Alisa Bell 44 Sophie Baker 35	Alisa Bell is 44 years old. Seth Nelson is 65 years old. Sophie Baker is 35 years old. Liam Scott is 57 years old. Brian Clark is 27 years old.

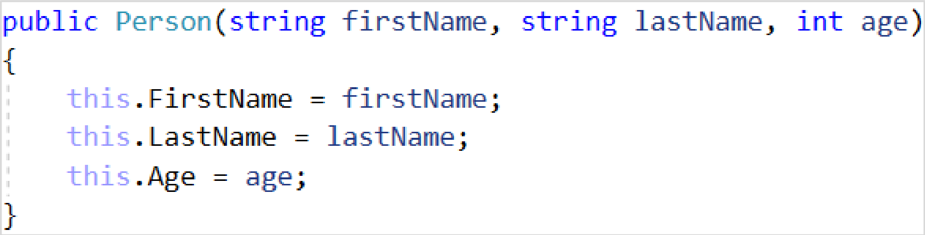
Solution

Create a **new class** and ensure **proper naming**. Define the **public** properties:



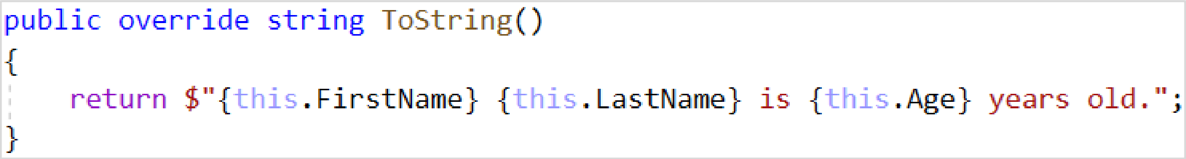
```
public class Person
{
    private int age;
    private string firstName;
    private string lastName;

    public int Age
    {
        get { return age; }
        set { age = value; }
    }
    public string FirstName
    {
        get { return firstName; }
        set { firstName = value; }
    }
    public string LastName
    {
        get { return lastName; }
        set { lastName = value; }
    }
```

Create a constructor for **Person**, which takes 3 parameters **firstName**, **lastName**, **age**:



```
public Person(string firstName, string lastName, int age)
{
    this.FirstName = firstName;
    this.LastName = lastName;
    this.Age = age;
}
```

Override **ToString()** method:



```
public override string ToString()
{
    return $"{this.FirstName} {this.LastName} is {this.Age} years old.";
}
```

2. Salary Increase

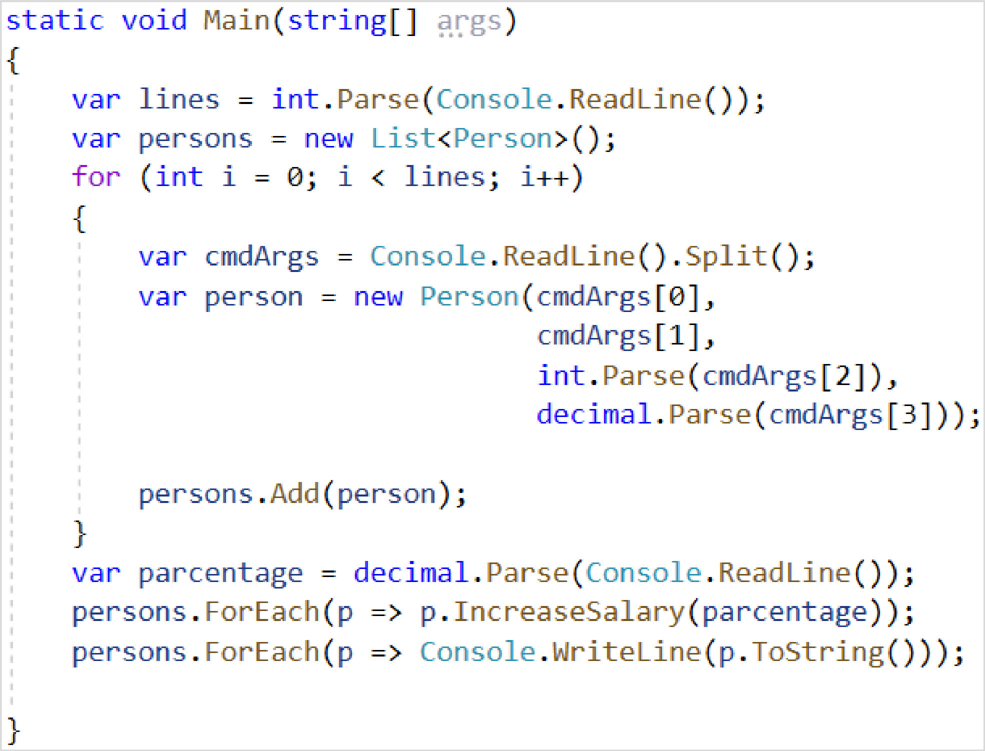
NOTE: You need a public **StartUp** class with the namespace **PersonsInfo**. **Refactor the project from the last task.**

Create objects of the class **Person**. Read their **name**, **age** and **salary** from the console. Read the percentage of the bonus to every **Person salary**. People younger than **30 get half the increase**. Expand **Person** from the previous task.

New **properties** and **methods:**

- **Salary: decimal**
- **IncreaseSalary(decimal percentage)**

You should be able to use the class like this:



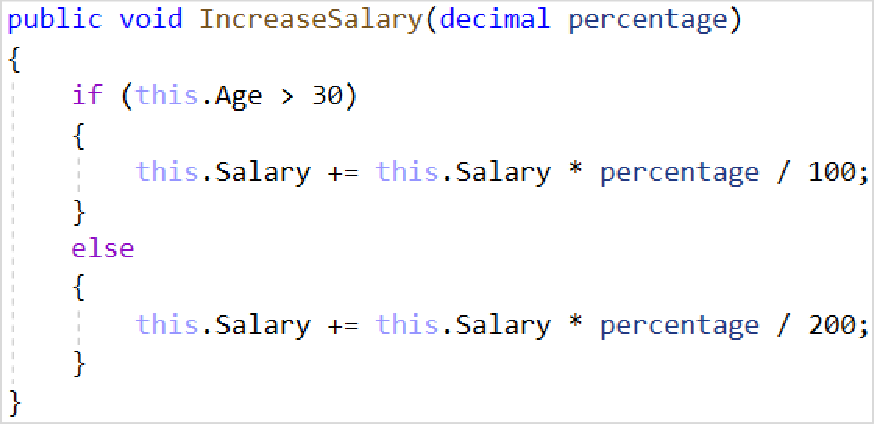
```
static void Main(string[] args)
{
    var lines = int.Parse(Console.ReadLine());
    var persons = new List<Person>();
    for (int i = 0; i < lines; i++)
    {
        var cmdArgs = Console.ReadLine().Split();
        var person = new Person(cmdArgs[0],
                                cmdArgs[1],
                                int.Parse(cmdArgs[2]),
                                decimal.Parse(cmdArgs[3]));

        persons.Add(person);
    }
    var parcentage = decimal.Parse(Console.ReadLine());
    persons.ForEach(p => p.IncreaseSalary(parcentage));
    persons.ForEach(p => Console.WriteLine(p.ToString()));

}
```

Examples

Input	Output
5 Nick Adams 65 2200 Lynda Fisher 57 3333 Paul Walker 27 600 Vera Nelson 44 666.66 Connor Perry 35 559.4 20	Nick Adams receives 2640.00 leva. Lynda Fisher receives 3999.60 leva. Paul Walker receives 660.00 leva. Vera Nelson receives 799.99 leva. Connor Perry receives 671.28 leva.

Solution

Add a new **`public`** property for **`salary`** and **refactor the constructor**. Add a new **method**, which will **update `salary`** with a bonus:



```
public void IncreaseSalary(decimal percentage)
{
    if (this.Age > 30)
    {
        this.Salary += this.Salary * percentage / 100;
    }
    else
    {
        this.Salary += this.Salary * percentage / 200;
    }
}
```

Refactor the **`ToString()`** method for this task.

3. Validation of Data

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

Expand **Person** with proper **validation** for every **field**:

- **Name** must be at **least 3 symbols**
- **Age** must **not** be **zero or negative**
- **Salary can't** be **less than 460 (decimal)**

Print proper messages to the user:

- **"Age cannot be zero or a negative integer!"**
- **"First name cannot contain fewer than 3 symbols!"**
- **"Last name cannot contain fewer than 3 symbols!"**
- **"Salary cannot be less than 460 leva!"**

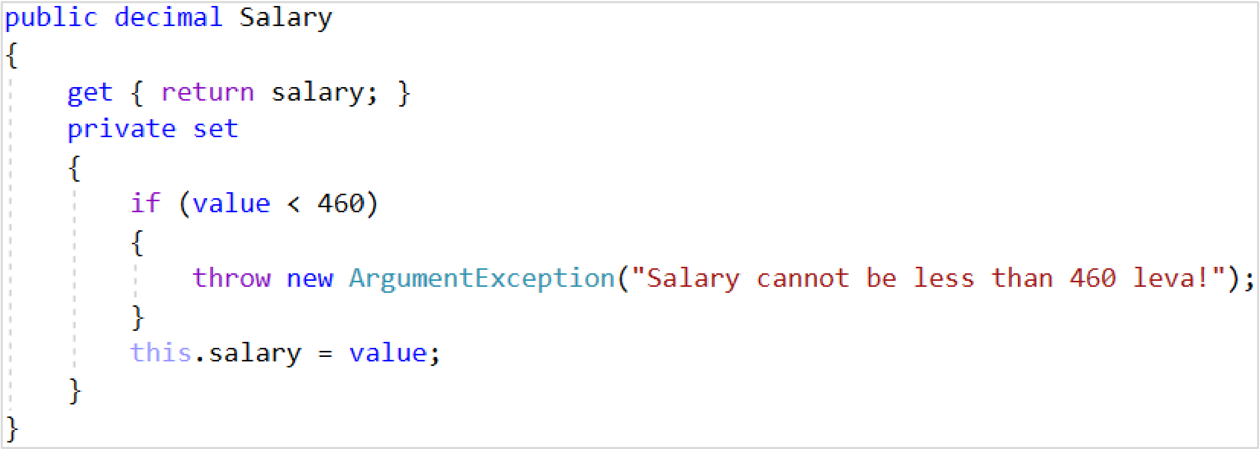
Use **ArgumentExeption** with the listed message.

Examples

Input	Output
5 Miles Parks -6 2200 B Potter 57 3333 Julie Brown 27 600 Alice H 44 666.66 Joey Hall 35 300 20	Age cannot be zero or a negative integer! First name cannot contain fewer than 3 symbols! Last name cannot contain fewer than 3 symbols! Salary cannot be less than 460 leva! Julie Brown gets 660.00 leva.

Solution

Add validation to all of the setters in **Person**. Validation may look like this or something similar:



```
public decimal Salary
{
    get { return salary; }
    private set
    {
        if (value < 460)
        {
            throw new ArgumentException("Salary cannot be less than 460 leva!");
        }
        this.salary = value;
    }
}
```

4. First and Reserve Team

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

Create a **Team** class. Add to this team all of the people you have received. Those who are **younger than 40** go to the **first team**, **others** go to the **reserve team**. At the end print the **sizes of the first** and the **reserved team**.

The class should have **private fields** for:

- **name: string**
- **firstTeam: List<Person>**
- **reserveTeam: List<Person>**

The class should have **constructors**:

- `Team(string name)`

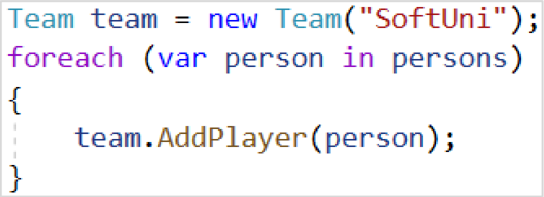
The class should also have **public properties** for:

- **`FirstTeam: List<Person> (read only!)`**
- **`ReserveTeam: List<Person> (read only!)`**

And a **method** for **adding players**:

- **`AddPlayer(Person person): void`**

You should be able to use the class like this:



```
Team team = new Team("SoftUni");
foreach (var person in persons)
{
    team.AddPlayer(person);
}
```

You should **NOT** be able to use the class like this:

StartUp.cs

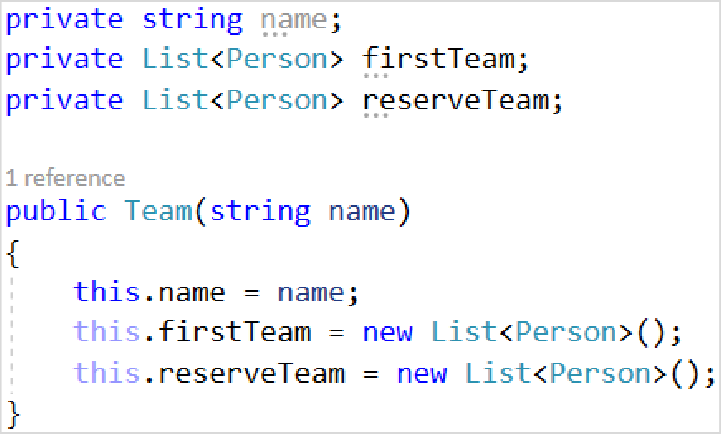
```
Team team = new Team("SoftUni");

foreach (Person person in persons)
{
    if(person.Age < 40)
    {
        team.FirstTeam.Add(person);
    }
    else
    {
        team.ReserveTeam(person);
    }
}
```

Examples

Input	Output
5 Troy Jones 20 2200 Martin Francis 57 3333 Ted Adams 27 600 Alisa Gomez 25 666.66 Lucia Cox 35 555	First team has 4 players. Reserve team has 1 players.

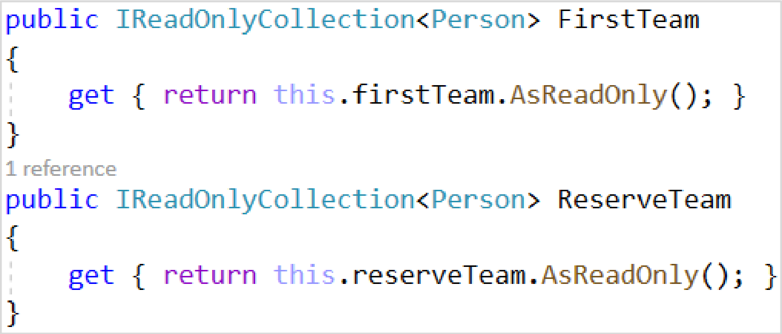
Solution

Add a new class **Team**. Its fields and **constructor** should look like



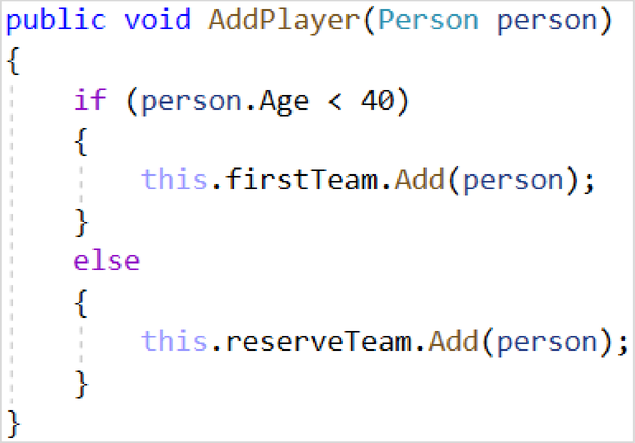
```
private string name;
private List<Person> firstTeam;
private List<Person> reserveTeam;

1 reference
public Team(string name)
{
    this.name = name;
    this.firstTeam = new List<Person>();
    this.reserveTeam = new List<Person>();
}
```

Properties for **`FirstTeam`** and **`ReserveTeam`** have only **getters**:



```
public IReadOnlyCollection<Person> FirstTeam
{
    get { return this.firstTeam.AsReadOnly(); }
}
1 reference
public IReadOnlyCollection<Person> ReserveTeam
{
    get { return this.reserveTeam.AsReadOnly(); }
}
```

There will be only **one method**, which **adds players** to teams:



```
public void AddPlayer(Person person)
{
    if (person.Age < 40)
    {
        this.firstTeam.Add(person);
    }
    else
    {
        this.reserveTeam.Add(person);
    }
}
```

5. Class Box Data

You are given a geometric figure box with parameters **`length`**, **`width`** and **`height`**. Model a class **Box** that can be instantiated by the same **three parameters**. Expose to the outside world **only methods for its surface area**, **lateral surface area** and its **volume** (formulas: http://www.mathwords.com/r/rectangular_parallelepiped.htm).

A box's **side** should **not be zero or a negative number**. Add **data validation** for each **parameter** given to the **constructor**. Make a **`private` setter** that performs data **validation internally**.

Input

- On the **first three lines** you will get the **`length`**, **`width`** and **`height`**.

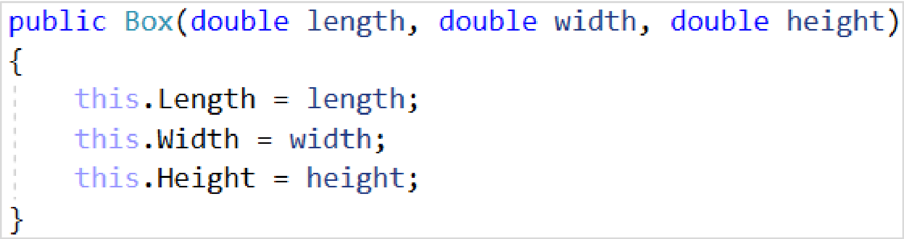
Output

- On the **next three lines** print the **surface area**, **lateral surface area** and the **volume** of the box:

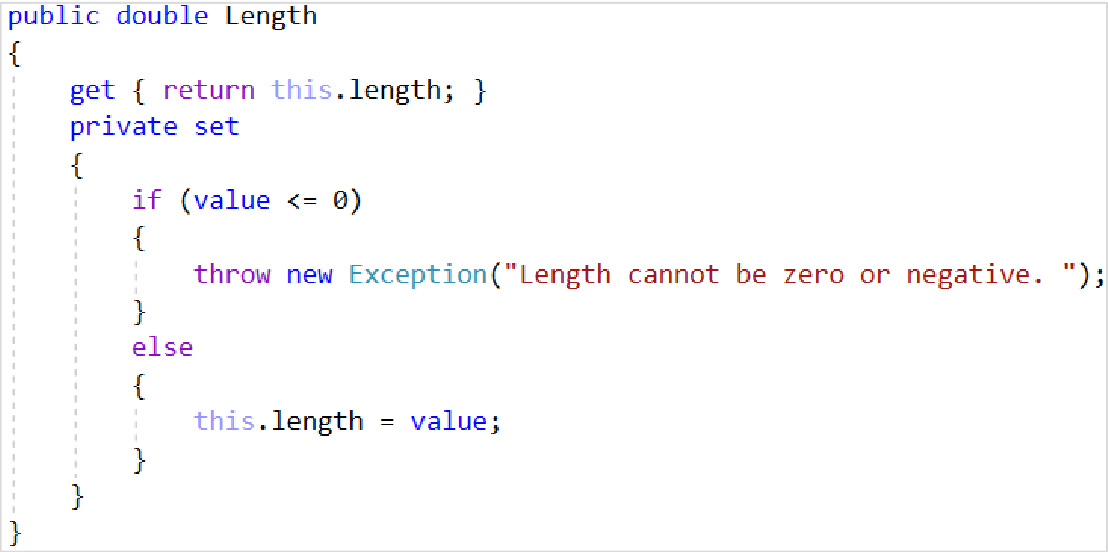
Examples

Input	Output
2 3 4	Surface Area - 52.00 Lateral Surface Area - 40.00 Volume - 24.00
1.3 1 6	Surface Area - 30.20 Lateral Surface Area - 27.60 Volume - 7.80
2 -3 4	Width cannot be zero or negative.

Hints:



```
public Box(double length, double width, double height)
{
    this.Length = length;
    this.Width = width;
    this.Height = height;
}
```



```
public double Length
{
    get { return this.length; }
    private set
    {
        if (value <= 0)
        {
            throw new Exception("Length cannot be zero or negative. ");
        }
        else
        {
            this.length = value;
        }
    }
}
```

6. Animal Farm

For this problem you have to **download** the provided **skeleton**.

You should be familiar with **encapsulation** already. For this problem, you'll be working with the **`AnimalFarm` project**. It contains a class **`Chicken`**. **`Chicken`** contains several **fields**, a **constructor**, several **properties** and **methods**. Your task is to **encapsulate** or **hide** anything that is **unintended for viewing** or **modification** from **outside** the class.

Step 1. Encapsulate Fields

Fields should be **`private`**. Leaving fields open for modification from outside the class is potentially **dangerous**. Make **all fields** in the **`Chicken`** class **`private`**. In case the value inside the field is needed elsewhere, use **getters** to reveal it.

Step 2. Ensure Classes Have a Correct State

Having **getters and setters** is useless, if you don't actually use them. The **`Chicken`** constructor **modifies the fields directly**, which is **wrong** when there are suitable **setters** available. **Modify** the constructor to fix this issue.

Step 3. Validate Data Properly

Validate the chicken's **name** (it cannot be **null**, **empty** or **whitespace**). In case of **invalid name**, print Exception message: **`"Name cannot be empty."`** .

Validate the **age** properly, **minimum** and **maximum age** are provided, make use of them. In case of an **invalid age**, print Exception message: **`"Age should be between 0 and 15."`**. Don't forget to **handle properly** the possibly **thrown Exceptions**.

Step 4. Hide Internal Logic

If a **method** is intended to be used only by **descendant** classes or **internally** to perform some action, there is no point in keeping them **`public`**. The **`CalculateProductPerDay()`** method is used by the **`ProductPerDay`** public getter. This means the method can safely be **hidden** inside the **`Chicken`** class by declaring it **`private`**.

Step 5. Submit Code to Judge

Submit your code as a **zip file** in Judge. Zip everything **except** the **bin** and **obj folders** within the project and submit the **single zip file** in judge.

Examples

Input	Output
Lucia 10	Chicken Lucia (age 10) can produce 1 eggs per day.
Lucia 17	Age should be between 0 and 15.

7. Shopping Spree

Create two classes: **class `Person`** and **class `Product`**. Each person should have a **`name`**, **`money`** and a **`bag of products`**. Each product should have a **`name`** and a **`cost`**. Name cannot be an **empty `string`**. Money cannot be a **negative number**.

Create a program in which **each command** corresponds to a **person buying a product**. If the person can **afford** a product, **add** it to his bag. If a person **doesn't have enough** money, print an **appropriate message** ("**`{personName} can't afford {productName}`**").

On the **first two lines** you are given **all people** and **all products**. After all purchases print **every person** in the order of **appearance** and **all products** that he has **bought** also in order of **appearance**. If **nothing was bought**, print the name of the person followed by "**`Nothing bought`**".

In case of **invalid input** (negative money Exception message: "**`Money cannot be negative`**") or an empty name (empty name Exception message: "**`Name cannot be empty`**") **break** the program with an appropriate message. See the examples below:

Examples

Input	Output

`Mark=11;Lesley=4`
`Bread=10;Milk=2`
`Mark Bread`
`Lesley Milk`
`Lesley Milk`
`Mark Milk`
`END`
	`Mark bought Bread`
	`Lesley bought Milk`
	`Lesley bought Milk`
	`Mark can't afford Milk`
	`Mark - Bread`
	`Lesley - Milk, Milk`

`Philip=0`
`Coffee=2`
`Philip Coffee`
`END`
	`Philip can't afford Coffee`
	`Philip - Nothing bought`

`Sandy=-3`
`Pepper=1`
`Sandy Pepper`
`END`
	`Money cannot be negative`

8. Pizza Calories

A pizza is made of dough and different toppings. You should model a **class `Pizza`,** which should have a **name**, **dough** and **`toppings`** as fields. Every type of **ingredient** should have its **own class**. Every ingredient has different properties: the **dough** can be white or wholegrain and in addition, it can be crispy, chewy or homemade. The **`topping`** can be of type meat, veggies, cheese or sauce. **Every ingredient** should have a **`weight`** in grams and a method for **calculating** its calories according to its type. Calories per gram are calculated through **modifiers**. Every ingredient has 2 calories per gram as a **base** and a **modifier** that **gives** the **exact** calories. For example, a white dough has a modifier of 1.5, a chewy dough has a modifier of 1.1, which means that a **white chewy** dough, weighting **100 grams** will have 2 * 100 * 1.5 * 1.1 = 330.00 **total calories**.

Your job is to model the classes in such a way that they are **properly encapsulated** and to provide a **`public`** method for every pizza that **calculates its calories according to the ingredients it has**.

Step 1. Create a Dough Class

The base ingredient of a **`Pizza`** is the dough. First, you need to create a **class** for it. It has a **flour type,** which can be **white** or **wholegrain**. In addition, it has a **baking technique,** which can be **crispy**, **chewy** or **homemade**. A dough should have a **weight** in grams. The calories per gram of a dough are calculated **depending** on the **flour type** and the **baking technique**. Every **dough** has **2 calories per gram** as a base and a **modifier** that gives the exact calories. For example, a white dough has a modifier of 1.5, a chewy dough has a modifier of 1.1, which means that a **white chewy dough**, weighting **100 grams** will have (2 * 100) * 1.5 * 1.1 = 330.00 **total calories**. You are given the **modifiers** below:

Modifiers:

- **`White - 1.5;`**
- **`Wholegrain - 1.0;`**
- **`Crispy - 0.9;`**
- **`Chewy - 1.1;`**
- **`Homemade - 1.0;`**

Everything that the class should expose is a **getter** for the **calories per gram**. Your task is to create the class with a proper **constructor**, **fields**, **getters** and **setters**. Make sure you use the **proper access modifiers**.

Step 2. Validate Data for the Dough Class

Change the internal logic of the **Dough** class by adding a **data validation** in the **setters**.

Make sure that if **invalid flour type** or an **invalid baking technique** is given a proper **`Exception`** is thrown with the message **`"Invalid type of dough."`**.

The allowed weight of a dough is in the **range** [1..200] grams. If it is **outside** of this **range** throw an **`Exception`** with the message **`"Dough weight should be in the range [1..200]."`**.

Exception Messages

- **`"Invalid type of dough."`**
- **`"Dough weight should be in the range [1..200]."`**

Make a test in your main method that reads Doughs and prints their calories until an "**END**" command is given.

Examples

Input	Output

`Dough White Chewy 100`
`END`
	`330.00`

`Dough Tip500 Chewy 100`
`END`
	`Invalid type of dough.`

`Dough White Chewy 240`
`END`
	`Dough weight should be in the range [1..200].`

Step 3. Create a Topping Class

Next, you need to create a **Topping class**. It can be of four different types - **meat**, **veggies**, **cheese** or a **sauce**. A **Topping** has a **weight** in grams. The **calories per gram** of topping are **calculated depending on its type**. The **base calories per gram** are **2**. Every different type of topping has a **modifier**. For example, **meat** has a **modifier of 1.2**, so a **meat** topping will have **1.2 calories per gram** (1 * 1.2). Everything that the class should expose is a **getter** for **calories per gram**. You are given the **modifiers** below:

Modifiers:

- **`Meat - 1.2;`**
- **`Veggies - 0.8;`**
- **`Cheese - 1.1;`**
- **`Sauce - 0.9;`**

Your task is to create the class with a **proper constructor**, **fields**, **getters** and **setters**. Make sure you use the **proper access modifiers**.

Step 4. Validate Data for the Topping Class

Change the internal logic of the **Topping** class by adding **data validation** in the **setter**.

Make sure the **Topping** is one of the provided types, otherwise throw a proper **Exception** with the message **"Cannot place [name of invalid argument] on top of your pizza."**.

The allowed weight of a **Topping** is in the range [1..50] grams. If it is **outside of this range** throw an **Exception** with the message **"[Topping type name] weight should be in the range [1..50]."**.

Exception Messages

- **"Cannot place [name of invalid argument] on top of your pizza."**
- **"[Topping type name] weight should be in the range [1..50]."**

Make a test in your main method that reads a single dough and a topping after that and prints their calories.

Examples

Input	Output
Dough White Chewy 100 Topping meat 30 END	330.00 72.00
Dough White chewy 100 Topping Krenvirshi 500 END	330.00 Cannot place Krenvirshi on top of your pizza.
Dough White Chewy 100 Topping Meat 500 END	330.00 Meat weight should be in the range [1..50].

Step 5. Create a Pizza Class!

A **Pizza** should have a **name**, some **toppings** and a **dough**. Make use of the **two classes you made earlier**. In addition, a **Pizza** should have **public getters** for its **name**, **number of toppings** and the **total calories**. The **total calories** are **calculated by summing the calories of all the ingredients a Pizza has**. Create the class using a **proper constructor**, expose a **method** for **adding a topping**, a **public setter** for the dough and a **getter** for the **total calories**.

The input for a **Pizza** consists of **several lines**. On the first line is the **Pizza name** and on the second line, you will get input for the **dough**. On the next lines, you will receive every topping the **Pizza** has.

If the creation of the **Pizza** was **successful,** print on a single line the name of the **Pizza** and the **total calories** it has.

Step 6. Validate Data for the Pizza Class

The **name** of the **Pizza** should **not** be an **empty string**. In addition, it should **not be longer than 15 symbols**. If it does not fit, throw an **Exception** with the message **"Pizza name should be between 1 and 15 symbols."**.

The **number of toppings** should be in range [0..10]. If not, throw an **Exception** with the message **"Number of toppings should be in range [0..10]."**.

Your task is to print the **name** of the **Pizza** and the **total calories** it has according to the examples below.

Examples

Input	Output
Pizza Meatless Dough Wholegrain Crispy 100 Topping Veggies 50 Topping Cheese 50 END	Meatless - 370.00 Calories.
Pizza Burgas Dough White Homemade 200 Topping Meat 123 END	Meat weight should be in the range [1..50].
Pizza Bulgarian Dough White Chewy 100 Topping Sauce 20 Topping Cheese 50 Topping Cheese 40 Topping Meat 10 Topping Sauce 10 Topping Cheese 30 Topping Cheese 40 Topping Meat 20 Topping Sauce 30 Topping Cheese 25 Topping Cheese 40 Topping Meat 40 END	Number of toppings should be in range [0..10].
Pizza Bulgarian Dough White Chewy 100 Topping Sirene 50 Topping Cheese 50 Topping Krenvirsh 20 Topping Meat 10 END	Cannot place Sirene on top of your pizza.

9. Football Team Generator

A football **Team** has variable **number of players**, a **name** and a **rating**. A **Player** has a **name** and **stats,** which are the basis for his skill level. The stats a player has are **endurance**, **sprint**, **dribble**, **passing** and **shooting**. Each stat can be an **integer** in the range [0..100]. The overall **skill level** of a **player** is calculated as the **average** of his **stats**. Only the **name** of a player and his **stats** should be visible to the entire outside world. **Everything else** should be **hidden**.

A **Team** should expose a **name**, a **rating** (calculated by the average skill level of all players in the team and **rounded** to the **integer** part only) and **methods** for **adding** and **removing players**.

Your task is to **model** the **Team** and the **Player** classes following the proper principles of **Encapsulation**. Expose **only** the properties that need to be visible and **validate data** appropriately.

Input

Your application will receive commands until the "**END**" command is given. The command can be one of the following:

- **"Team;{TeamName}"** - add a new **Team**;
- **"Add;{TeamName};{PlayerName};{Endurance};{Sprint};{Dribble};{Passing};{Shooting}"** - add a new **Player** to the **Team**;
- **"Remove;{TeamName};{PlayerName}"** - remove the **Player** from the **Team**;
- **"Rating;{TeamName}"** - print the **Team** rating, rounded to an integer.

Data Validation

- A name cannot be null, empty or white space. If not, print **"A name should not be empty."**
- Stats should be in the range 0...100. If not, print **"[Stat name] should be between 0 and 100."**
- If you receive a command to remove a missing **Player**, print **"Player [Player name] is not in [Team name] team."**
- If you receive a command to add a **Player** to a missing **Team**, print **"Team [team name] does not exist."**
- If you receive a command to show stats for a missing **Team**, print **"Team [team name] does not exist."**

Examples

Input	Output
Team;Arsenal Add;Arsenal;Kieran_Gibbs;75;85;84;92;67 Add;Arsenal;Aaron_Ramsey;95;82;82;89;68 Remove;Arsenal;Aaron_Ramsey Rating;Arsenal END	Arsenal - 81
Team;Arsenal Add;Arsenal;Kieran_Gibbs;75;85;84;92;67 Add;Arsenal;Aaron_Ramsey;195;82;82;89;68 Remove;Arsenal;Aaron_Ramsey Rating;Arsenal END	Endurance should be between 0 and 100. Player Aaron_Ramsey is not in Arsenal team. Arsenal - 81
Team;Arsenal Rating;Arsenal END	Arsenal - 0