Exercises: Inheritance

This document defines the exercises for "C# OOP Basics" course @ Software University. Please submit your solutions (source code) of all below described problems in Judge.

Problem 1. Person

You are asked to model an application for storing data about people. You should be able to have a person and a child. The child is derived of the person. Your task is to model the application. The only constraints are:

- People should **not** be able to have **negative age**
- Children should **not** be able to have age **more than 15**.
- **Person** represents the base class by which all others are implemented
- Child represents a class which is derived by the Person.

Note

Your class's names MUST be the same as the names shown above!!!

```
Sample Main()
static void Main()
    string name = Console.ReadLine();
    int age = int.Parse(Console.ReadLine());
    try
    {
        Child child = new Child(name, age);
        Console.WriteLine(child);
    catch (ArgumentException ae)
        Console.WriteLine(ae.Message);
    }
```

Create a new empty class and name it Person. Set its access modifier to public so it can be instantiated from any project. Every person has a name, and age.

```
Sample Code
public class Person
{
   // 1. Add Fields
   // 2. Add Constructor
   // 3. Add Properties
```

















```
// 4. Add Methods
}
```

Step 2 - Define the fields

Define a field for each property the class should have (e.g. Name, Age)

Step 3 - Define the Properties of a Person

Define the Name and Age properties of a Person. Ensure that they can only be changed by the class itself or its descendants (pick the most appropriate access modifier).

```
Sample Code
public virtual string Name
{
    get
    {
        //T0D0
    }
    set
    {
        //TODO
    }
}
public virtual int Age
{
    get
    {
        //TODO
    }
    set
    {
         //TODO
    }
```

Step 4 - Define a Constructor

Define a constructor that accepts name and age.

```
Sample Code
public Person(string name, int age)
{
    this.Name = name;
    this.Age = age;
}
```

















Step 5 - Perform Validations

After you have created a field for each property (e.g. Name and Age). Next step is to perform validations for each one. The getter should return the corresponding field's value and the setter should validate the input data before setting it. Do this for each property.

```
Sample Code
public virtual int Age
{
    get
    {
         return this.age;
    }
    set
    {
        if (value < 0)</pre>
             throw new ArgumentException("Age must be positive!");
        }
        //TODO set field age with value
    }
```

Constraints

- If the age of a person is negative exception's message is: "Age must be positive!"
- If the age of a child is bigger than 15 exception's message is: "Child's age must be less than 15!"
- If the name of a child or a person is no longer than 3 symbols exception's message is: "Name's length should not be less than 3 symbols!"

Step 6 - Override ToString()

As you probably already know, all classes in C# inherit the **Object** class and therefore have all its **public** members (ToString(), Equals() and GetHashCode() methods). ToString() serves to return information about an instance as string. Let's override (change) its behavior for our Person class.

```
Sample Code
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append(String.Format("Name: {0}, Age: {1}",
                         this.Name,
                         this.Age));
    return stringBuilder.ToString();
}
```

And voila! If everything is correct, we can now create **Person objects** and display information about them.

















Step 7 - Create a Child

Create a Child class that inherits Person and has the same constructor definition. However, do not copy the code from the Person class - reuse the Person class's constructor.

```
Sample Code
public Child(string name, int age)
    : base(name, age)
{
```

There is no need to rewrite the Name and Age properties since Child inherits Person and by default has them.

Step 8 – Validate the Child's setter

```
Sample Code
public override int Age
    get
    {
        return base.Age;
    }
    set
    {
        //TODO validate childs age
        base.Age = value;
    }
```

Problem 2. Book Shop

You are working in a library. And you are pissed of writing descriptions for books by hand, so you wish to use the computer to speed up the process. The task is simple - your program should have two classes - one for the ordinary books - Book, and another for the special ones - GoldenEditionBook. So let's get started! We need two classes:

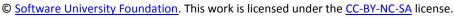
- Book represents a book that holds title, author and price. A book should offer information about itself in the format shown in the output below.
- **GoldenEditionBook** represents a special book holds the same properties as any **Book**, but its **price** is always 30% higher.

Constraints

- If the author's second name is starting with a digit exception's message is: "Author not valid!"
- If the title's length is less than 3 symbols exception's message is: "Title not valid!"
- If the price is zero or it is negative exception's message is: "Price not valid!"
- Price must be formatted to **two** symbols after the decimal separator

Sample Main()



















```
static void Main()
{
    try
    {
        string author = Console.ReadLine();
        string title = Console.ReadLine();
        decimal price = decimal.Parse(Console.ReadLine());

        Book book = new Book(author, title, price);
        GoldenEditionBook goldenEditionBook = new GoldenEditionBook(author, title, price);

        Console.WriteLine(book + Environment.NewLine);
        Console.WriteLine(goldenEditionBook);
    }
    catch (ArgumentException ae)
    {
        Console.WriteLine(ae.Message);
    }
}
```

Examples

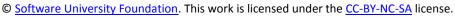
Input	Output
Ivo 4ndonov	Author not valid!
Under Cover	
9999999999999999	
Petur Ivanov	Type: Book
Life of Pesho	Title: Life of Pesho
20	Author: Petur Ivanov Price: 20.00
	Type: GoldenEditionBook Title: Life of Pesho
	Author: Petur Ivanov
	Price: 26.00

Step 1 - Create a Book Class

Create a new empty class and name it **Book**. Set its access modifier to **public** so it can be instantiated from any project.

Sample Code

















```
public class Book
{
    //1. Add Fields

    //2. Add Constructors

    //3. Add Properties

    //4. Add Methods
}
```

Step 2 - Define the Properties of a Book

Define the **Title**, **Author** and **Price** properties of a Book. Ensure that they can only be **changed by the class itself or its descendants** (pick the most appropriate access modifier).

Step 3 - Define a Constructor

Define a constructor that accepts **author**, **title** and **price** arguments.

```
public Book(string author, string title, decimal price)
{
   this.Author = author;
   this.Title = title;
   this.Price = price;
}
```

Step 4 - Perform Validations

Create a **field** for each property (**Price**, **Title** and **Author**) and **perform validations** for each one. The **getter should return the corresponding field** and the **setter should validate** the input data before setting it. Do this for every property.

```
public string Author
{
    get
    {
        return this.author;
    }
    set
    {
            //TODO validate value
            this.author = value;
    }
}
public string Title
```















```
{
    get
    {
        return this.title;
    }
    set
    {
        //TODO validate value
        this.title = value;
    }
}
public virtual decimal Price
{
    get
    {
        return this.price;
    }
    set
    {
        //TODO validate value
        this.price = value;
    }
}
```

Step 5 - Override ToString()

We already mentioned that all classes in C# inherit the **System.Object** class and therefore have all its **public** members. Let's override (change) the ToString() method's behavior again acordingly our Book class's data.

```
Sample Code
public override string ToString()
    var resultBuilder = new StringBuilder();
    resultBuilder.AppendLine($"Type: {this.GetType().Name}")
        .AppendLine($"Title: {this.Title}")
        .AppendLine($"Author: {this.Author}")
        .AppendLine($"Price: {this.Price:f2}");
    string result = resultBuilder.ToString().TrimEnd();
    return result;
}
```

And voila! If everything is correct, we can now create **Book objects** and display information about them.

Step 6 – Create a GoldenEditionBook

Create a GoldenEditionBook class that inherits Book and has the same constructor definition. However, do not copy the code from the Book class - reuse the Book class constructor.

Sample Code



















```
public GoldenEditionBook(string author, string title, decimal price)
    : base(author, title, price)
{
}
```

There is **no need** to rewrite the Price, Title and Author properties since **GoldenEditionBook** inherits **Book** and by default has them.

Step 7 - Override the Price Property

Golden edition books should return a 30% higher price than the original price. In order for the getter to return a different value, we need to override the Price property.

Back to the GoldenEditionBook class, let's override the Price property and change the getter body

```
Sample Code
public override decimal Price
{
    get
    {
        return base.Price * 1.3;
    }
```

Problem 3. Mankind

Your task is to model an application. It is very simple. The mandatory models of our application are 3: Human, Worker and Student.

The parent class – Human should have first name and last name. Every student has a faculty number. Every worker has a week salary and work hours per day. It should be able to calculate the money he earns by hour. You can see the constraints below.

Input

On the first input line you will be given info about a single student - a name and faculty number.

On the second input line you will be given info about a single worker - first name, last name, salary and working hours.

Output

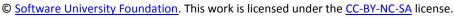
You should first print the info about the student following a single blank line and the info about the worker in the given formats:

Print the student info in the following format:

First Name: {student's first name} Last Name: {student's last name}

Faculty number: {student's faculty number}

















Print the worker info in the following format:

First Name: {worker's first name} Last Name: {worker's second name} Week Salary: {worker's salary}

Hours per day: {worker's working hours} Salary per hour: {worker's salary per hour}

Print exactly two digits after every double value's decimal separator (e.g. 10.00). Consider the workweek from Monday to Friday. A faculty number should be consisted only of digits and letters.

Constraints

Parameter	Constraint	Exception Message
Human first name	Should start with a capital letter	"Expected upper case letter! Argument: firstName"
Human first name	Should be more than 3 symbols	"Expected length at least 4 symbols! Argument: firstName"
Human last name	Should start with a capital letter	"Expected upper case letter! Argument: lastName"
Human last name	Should be more than 2 symbols	"Expected length at least 3 symbols! Argument: lastName "
Faculty number	Should be in range [510] symbols	"Invalid faculty number!"
Week salary	Should be more than 10	"Expected value mismatch! Argument: weekSalary"
Working hours	Should be in the range [112]	"Expected value mismatch! Argument: workHoursPerDay"

Example

Input	Output
Ivan Ivanov 08	Invalid faculty number!
Pesho Kirov 1590 10	
Stefo Mk321 0812111	First Name: Stefo
Ivcho Ivancov 1590 10	Last Name: Mk321
	Faculty number: 0812111
	First Name: Ivcho
	Last Name: Ivancov
	Week Salary: 1590.00
	Hours per day: 10.00
	Salary per hour: 31.80

Problem 4. Online Radio Database

Create an online radio station database. It should keep information about all added songs. On the first line you are going to get the number of songs you are going to try to add. On the next lines you will get the songs to be added in the format <artist name>;<song name>;<minutes:seconds>. To be valid, every song should have an artist name, a song name and length.















Design a custom exception hierarchy for invalid songs:

- InvalidSongException
 - InvalidArtistNameException
 - InvalidSongNameException
 - InvalidSongLengthException
 - InvalidSongMinutesException
 - InvalidSongSecondsException

Validation

- Artist name should be between 3 and 20 symbols.
- Song name should be between 3 and 30 symbols.
- Song length should be between 0 second and 14 minutes and 59 seconds.
- Song minutes should be between 0 and 14.
- Song seconds should be between 0 and 59.

Exception Messages

Exception	Message
InvalidSongException	"Invalid song."
InvalidArtistNameException	"Artist name should be between 3 and 20 symbols."
InvalidSongNameException	"Song name should be between 3 and 30 symbols."
InvalidSongLengthException	"Invalid song length."
InvalidSongMinutesException	"Song minutes should be between 0 and 14."
InvalidSongSecondsException	"Song seconds should be between 0 and 59."

Note: Check validity in the order artist name -> song name -> song length

Output

If the song is added, print "Song added.". If you can't add a song, print an appropriate exception message. On the last two lines print the number of songs added and the total length of the playlist in format {Playlist length: 0h 7m 47s}.

Examples

Exception	Message
3	Song added.
ABBA;Mamma Mia;3:35	Song seconds should be between 0 and 59.
Nasko Mentata; Shopskata salata; 4:123	Song added.
Nasko Mentata; Shopskata salata; 4:12	Songs added: 2
	Playlist length: 0h 7m 47s
5	Song added.
Nasko Mentata;Shopskata salata;14:59	Song added.

















Song added. Nasko Mentata; Shopskata salata; 14:59 Nasko Mentata; Shopskata salata; 14:59 Song added. Nasko Mentata; Shopskata salata; 14:59 Song added. Nasko Mentata; Shopskata salata; 0:5 Songs added: 5 Playlist length: 1h 0m 1s

Problem 5. *Mordor's Cruel Plan

Gandalf the Gray is a great wizard but he also loves to eat and the food makes him loose his capability of fighting the dark. The Mordor's orcs have asked you to design them a program which is calculating the Gandalf's mood. So they could predict the battles between them and try to beat The Gray Wizard. When Gandalf is hungry he gets angry and he could not fight well. Because the orcs have a spy, he has told them the foods that Gandalf is eating and the result on his mood after he has eaten some food. So here is the list:

Cram: 2 points of happiness;

Lembas: 3 points of happiness;

Apple: 1 point of happiness;

Melon: 1 point of happiness;

HoneyCake: 5 points of happiness;

Mushrooms: -10 points of happiness;

Everything else: -1 point of happiness;

Gandalf moods are:

Angry - below -5 points of happiness;

Sad - from -5 to 0 points of happiness;

Happy - from 1 to 15 points of happiness;

JavaScript - when happiness points are more than 15;

The task is simple. Model an application which is calculating the happiness points, Gandalf has after eating all the food passed in the input. After you are done, print on the first line – total happiness points Gandalf had collected. On the second line – print the **Mood's** name which is corresponding to the points.

Input

The input comes from the console. It will hold a single line: all the Gandalf's foods he has eaten separated by a whitespace.

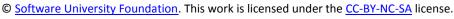
Output

Print on the console Gandalf's happiness points and the **Mood's** name which is corresponding to the points.

Constraints

- The characters in the input string will be no more than: 1000.
- The food count would be in the range [1...100].
- Time limit: 0.3 sec. Memory limit: 16 MB.





















Note

Try to implement a factory pattern. You should have two factory classes - FoodFactory and MoodFactory. And their task is to produce objects (e.g. FoodFactory, produces - Food and the MoodFactory - Mood). Try to implement abstract classes (e.g. classes which can't be instantiated directly)

Examples

Input	Output
Cram melon honeyCake Cake	7
	Нарру
gosho, pesho, meze, Melon, HoneyCake@;	-5
	Sad

Problem 6. Animals

Create a hierarchy of Animals. Your program should have 3 different animals – Dog, Frog and Cat. Deeper in the hierarchy you should have two additional classes - Kitten and Tomcat. Kittens are female and Tomcats are male!

All types of animals should be able to produce some kind of sound (ProduceSound()). For example, the dog should be able to bark.

Your task is to model the hierarchy and test its functionality. Create an animal of each kind and make them all produce sound.

You will be given some lines of input. Each two lines will represent an animal. On the first line will be the type of animal and on the second - the name, the age and the gender. When the command "Beast!" is given, stop the input and print all the animals in the format shown below.

Output

- Print the information for each animal on three lines. On the first line, print: "<AnimalType>"
- On the second line print: "<Name> <Age> <Gender>"
- On the third line print the sounds it produces: "<**ProduceSound()**>"

Constraints

- Each Animal should have a name, an age and a gender
- **All** input values should **not be blank** (e.g. name, age and so on...)
- If you receive an input for the **gender** of a **Tomcat** or a **Kitten**, ignore it but **create** the animal
- If the input is invalid for one of the properties, throw an exception with message: "Invalid input!"
- Each animal should have the functionality to **ProduceSound()**
- Here is the type of sound each animal should produce:
 - o Dog: "Woof!"
 - Cat: "Meow meow"
 - o Frog: "Ribbit"
 - Kittens: "Meow"
 - Tomcat: "MEOW"

Examples

Input Output





















Cat	Cat
Tom 12 Male	Tom 12 Male
Dog	Meow meow
Sharo 132 Male	Dog
Beast!	Sharo 132 Male
	Woof!
Frog	Frog
Kermit 12 Male	Kermit 12 Male
Beast!	Ribbit
Frog	Invalid input!
Sashko -2 Male	Frog
Frog	Sashko 2 Male
Sashko 2 Male	Ribbit
Beast!	

Bonus

Create an interface **ISoundProducable** and implement it in the **Animal** class.

















