# Exercises: OOP Principles Part 1

Problems for the [“Fund Of Programming” course @ Software University](https://softuni.bg/trainings/2133/fundamentals-of-programming-with-c-sharp-november-2018)

You can check your solutions here: <https://judge.softuni.bg/Contests/1456/OOP-Principles>.

# Part I: Encapsulation

## Sort Persons by Name and Age

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

Create a class **Person**, which should have **private** fields for:

* **firstName**: **string**
* **lastName**: **string**
* **age**: **int**
* **ToString()**: **string** - **override**

You should be able to use the class like this:

|  |
| --- |
| StartUp.cs |
| public static void Main()  {  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  Asen Ivanov 65  Boiko Borisov 57  Ventsislav Ivanov 27  Asen Harizanoov 44  Boiko Angelov 35 | Asen Harizanoov is 44 years old.  Asen Ivanov is 65 years old.  Boiko Angelov is 35 years old.  Boiko Borisov is 57 years old.  Ventsislav Ivanov is 27 years old. |

### Solution

Create a **new class** and ensure **proper naming**. Define the **private** fields:



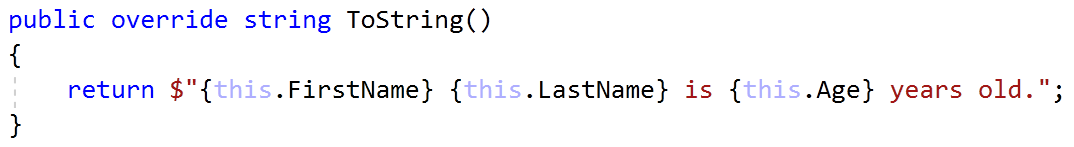
Create a constructor for Person, which takes 3 parameters firstName, lastName, age:



Create properties for these fields, which are as strictly as possible:



Override **ToString()** method:



## Salary Increase

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

**Refactor the project from the last task.**

Read a Person with their names, age and salary. 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 **fields** and **methods:**

* **salary**: **decimal**
* **IncreaseSalary**(**decimal** **percentage**)

You should be able to use the class like this:

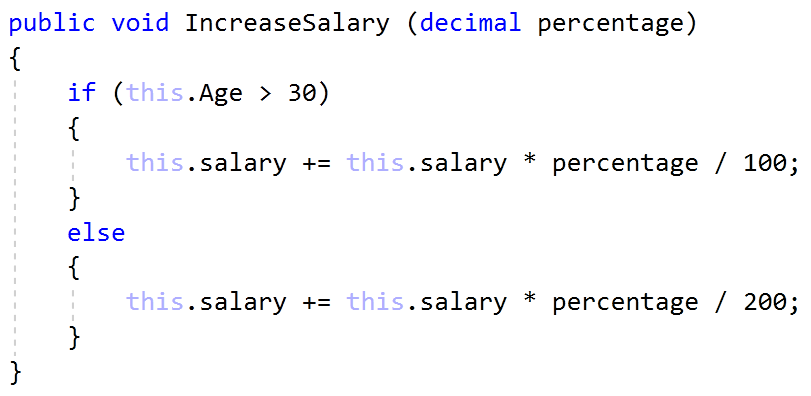
|  |
| --- |
| StartUp.cs |
| var lines = int.Parse(Console.ReadLine());  var persons = new List<Person>();  for (int i = 0; i < lines; i++)  {  var cmdArgs = Console.ReadLine().Split();  var person = new Person(cmdArgs[0],  cmdArgs[1],  int.Parse(cmdArgs[2]),  decimal.Parse(cmdArgs[3]));  persons.Add(person);  }  var bonus = decimal.Parse(Console.ReadLine());  persons.ForEach(p => p.IncreaseSalary(bonus));  persons.ForEach(p => Console.WriteLine(p.ToString())); |

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| 5  Asen Ivanov 65 2200  Boiko Borisov 57 3333  Ventsislav Ivanov 27 600  Asen Harizanoov 44 666.66  Boiko Angelov 35 559.4  20 | Asen Ivanov receives 2640.00 leva.  Boiko Borisov receives 3999.60 leva.  Ventsislav Ivanov receives 660.00 leva.  Asen Harizanoov receives 799.99 leva.  Boiko Angelov receives 671.28 leva. |

### Solution

Add a new **private** field for **salary** and **refactor constructor**. Add new **method**, which will **update** salary with a bonus



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

## Class Box

You are given a geometric figure box with parameters length, width and height. Model a class Box that 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>).

On the first three lines you will get the length, width and height. 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 |

## 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** ("[Person name] can't afford [Product name]").

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** |
| Pesho=11;Gosho=4  Bread=10;Milk=2;  Pesho Bread  Gosho Milk  Gosho Milk  Pesho Milk  END | Pesho bought Bread  Gosho bought Milk  Gosho bought Milk  Pesho can't afford Milk  Pesho - Bread  Gosho - Milk, Milk |
| Mimi=0  Kafence=2  Mimi Kafence  END | Mimi can't afford Kafence  Mimi – Nothing bought |
| Jeko=-3  Chushki=1;  Jeko Chushki  END | Money cannot be negative |

# Part II: Inheritance

## Single Inheritance

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

Create two classes named Animal and Dog.

Animal with a single public method Eat() that prints: **"eating…"**

Dog with a single public method Bark() that prints: **"barking…"**

Dog should inherit from Animal.



### Hints

Use the **: operator** to build a hierarchy

## Multiple Inheritance

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

Create three classes named Animal, Dog and Puppy.

Animal with a single public method Eat() that prints: **"eating…"**

Dog with a single public method Bark() that prints: **"barking…"**

Puppy with a single public method Weep() that prints: **"****weeping…"**

Dog should inherit from Animal. Puppy should inherit from Dog.



## Hierarchical Inheritance

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

Create three classes named Animal, Dog and Cat.

Animal with a single public method Eat() that prints: **"eating…"**

Dog with a single public method Bark() that prints: **"barking…"**

Cat with a single public method Meow() that prints: **"meowing…"**

Dog and Cat should inherit from Animal.



## 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 derives from the person. Your task is to model the application. The only constraints are:

* People should **not** be able to have a **negative age**
* Children should **not** be able to have an age **more than 15**.
* **Person** – represents the base class by which all of the others are implemented
* **Child** - represents a class, which derives from **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 an 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 **the class can only be changed by itself or its descendants** (pick the most appropriate access modifier).

|  |
| --- |
| **Sample Code** |
| public virtual string Name  {  get  {  //TODO  }  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**), the 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)  {  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 three 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;  }  } |

## Book Shop

You are working in a library. You are sick 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 that 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 – the exception’s message is: "Author not valid!"
* If the title’s length is less than 3 symbols – the exception’s message is: "Title not valid!"
* If the price is zero or it is negative – the 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  Under Cover  9999999999999999999 | Author not valid! |
| Petur Ivanov  Life of Pesho  20 | Type: Book  Title: Life of Pesho  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.

|  |
| --- |
| **Sample Code** |
| 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.

|  |
| --- |
| **Sample Code** |
| 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 have already mentioned that all of the 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 according to 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;  }  } |