# Exercises: Inheritance

This document defines the exercises for ["C# DB Advanced" course @ Software University](https://softuni.bg/trainings/1636/c-sharp-oop-basics-june-2017).

## 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’s **names** should be at least **3 symbols** long
* 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 from the **Person.**

### Note

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

Use this Main() method to test your solution locally and in Judge:

|  |
| --- |
| **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  {  //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**). 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 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 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()  {  return $"Name: {this.Name}, Age: {this.Age}";  } |

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)  {  this.Age = 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 and you don’t want to write descriptions for books by hand, so you wish to use a 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 – the exception message is **"****Author not valid!"**
* If the title’s length is less than 3 symbols – the exception message is **"****Title not valid!"**
* If the price is zero or it is negative – the exception message is **"****Price not valid!"**
* Price must be formatted to **two** symbols after the decimal separator

Use this Main() method to test your solution locally and in Judge:

|  |
| --- |
| **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);  }  } |

### Example

|  |  |
| --- | --- |
| **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 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()  {  return $"Type: {this.GetType().Name}" + Environment.NewLine +  $"Title: {this.Title}" + Environment.NewLine +  $"Author: {this.Author}" + Environment.NewLine +  $"Price: {this.Price:f2}";  } |

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**. Inside your new constructor set the price to a **30% higher** value.

|  |
| --- |
| **Sample Code** |
| public GoldenEditionBook(string title, string author, decimal price)  :base(title, author, price)  {  this.Price \*= 1.3m;  } |

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

## 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}**

All numeric values will be **real numbers**, even the working hours. Use the most appropriate data type for each of them.

Print exactly **two digits** after every numeric 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 [5..10] symbols | "Invalid faculty number!" |
| Week salary | Should be more than 10 | "Expected value mismatch! Argument: weekSalary" |
| Working hours | Should be in the range [1..12] | "Expected value mismatch! Argument: workHoursPerDay" |

### Example

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

## 