# Exercises: Inheritance

This document defines the exercises for ["Java OOP Basics" course @ Software University](https://softuni.bg/java-basics-oop). Please submit your solutions (source code) of all below described problems in [Judge](https://judge.softuni.bg/).

## 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:

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

### Note

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

|  |
| --- |
| **Sample Main()** |
| **public static void** main(String[] args) {   Scanner scanner = **new** Scanner(System.***in***);  String name = scanner.nextLine();  Integer age = Integer.*valueOf*(scanner.nextLine());   **try** {  Child child = **new** Child(name, age);  System.***out***.println(child.toString());  String personClassName = Person.**class**.getSimpleName();  String childClassName = Child.**class**.getSimpleName();  } **catch** (IllegalArgumentException error) {  System.***out***.println(error.getMessage());  } } |

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 the Fields  // 2. Add the Constructor  // 3. Add the Properties  // 4. Add the Methods* } |

### Step 1. Define the fields

Define a **field** for each property the class should have (e.g. **name**, **age**)

### Step 2. 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** |
| **(modifier)** String getName() {  *//* ***TODO*** }  **(modifier) void** setName(String name) {  *//* ***TODO*** }  **(modifier)** Integer getAge() {  *//* ***TODO*** }  **(modifier) void** setAge(Integer age) {  *//* ***TODO*** } |

### Step 3. Define a Constructor

Define a constructor that accepts **name, age** and **address** arguments.

|  |
| --- |
| **Sample Code** |
| **public** Person(String name, Integer age){  **this**.setName(name);  **this**.setAge(age);  } |

### Step 4. 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** |
| **protected void** setAge(Integer age) **throws** IllegalArgumentException {  **if** (age < 1) {  **throw new** IllegalArgumentException(**"Age must be positive!"**);  }   *//* ***TODO: Set the age*** } |

### 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 5. Override toString()

As you probably already know, all classes in Java 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** |
| **@Override public** String toString() {  **final** StringBuilder stringBuilder = **new** StringBuilder();  stringBuilder.append(String.*format*(**"Name: %s, Age: %d"**,  **this**.getName(),  **this**.getAge()));   **return** stringBuilder.toString(); } |

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

### Step 6. 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, Integer age){  **super**(name, age); } |

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

### Step 7. Validate the Child’s setter

|  |
| --- |
| **Sample Code** |
| **@Override protected void** setAge(Integer age) **throws** IllegalArgumentException {  *//****TODO: Validate the age***  **super**.setAge(age); } |

## Book Shop

You are working in a library. And you are pissed off by writing descriptions for books by hand, so you wanted to use the computer to make them faster. So 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!"**

|  |
| --- |
| **Sample Main()** |
| **public static void** main(String[] args) **throws** IllegalClassFormatException {  **try** {  Scanner scanner = **new** Scanner(System.***in***);  String author = scanner.nextLine();  String title = scanner.nextLine();  Double price = Double.*valueOf*(scanner.nextLine());  Book book = **new** Book(author,  title,  price);    GoldenEditionBook goldenEditionBook = **new** GoldenEditionBook(author,  title,   price);    Method[] bookDeclaredMethods = Book.**class**.getDeclaredMethods();  Method[] goldenBookDeclaredMethods = GoldenEditionBook.**class**.getDeclaredMethods();  **if** (goldenBookDeclaredMethods.**length** > 1) {  **throw new** IllegalClassFormatException(**"Code duplication in GoldenEditionBook!"**);  }  System.***out***.println(book.toString());  System.***out***.println(goldenEditionBook.toString());  } **catch** (IllegalArgumentException error) {  System.***out***.println(error.getMessage());  } } |

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Ivo 4ndonov  Under Cover  9999999999999999999 | Author not valid! |

### Step 1. Create a Book Class

Create a new 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 the Fields  // 2. Add the Constructor  // 3. Add the Properties  // 4. Add the 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,  **double** price) {   **this**.setAuthor(author);  **this**.setTitle(title);  **this**.setPrice(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** |
| **(modifier)** String getAuthor() {  **return this**.**author**; }  **(modifier) void** setAuthor(String author) {  ***//TODO: Validate as it is written in Constraints***  **this**.**author** = author; }  **(modifier)** String getTitle() {  **return this**.**title**; }  **(modifier) void** setTitle(String title) {  **if** (title.length() < 3) {  **throw new** IllegalArgumentException(**"Title not valid!"**);   }   **this**.**title** = title; }  **(modifier) double** getPrice() {  **return this**.**price**; }  **(modifier) void** setPrice(**double** price) {  **if** (price < 1) {  **throw new** IllegalArgumentException(**"Price not valid!"**);   }   **this**.**price** = price; } |

### Step 5. Override toString()

As you probably already know, all classes in C# inherit the **System.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 **Book** class.

|  |
| --- |
| **Sample Code** |
| **@Override public** String toString() {  **final** StringBuilder sb = **new** StringBuilder();  sb.append(**"Type: "**).append(**this**.getClass().getSimpleName())  .append(System.*lineSeparator*())  .append(**"Title: "**).append(**this**.getTitle())  .append(System.*lineSeparator*())  .append(**"Author: "**).append(**this**.getAuthor())  .append(System.*lineSeparator*())  .append(**"Price: "**).append(**this**.getPrice())  .append(System.*lineSeparator*());  **return** sb.toString(); |

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**.



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** |
| **@Override public double** getPrice() {  **return super**.getPrice() + **super**.getPrice() \* 0.3; } |

## Mankind

Your task is to model an application. It is very simple. The mandatory models of our application are 3: a **Student** and a **Worker** models along with a **Human**, the one they **derive** from. The main model – **Human** consists of two elements: **First name** and **Last name**. Let’s start with the characteristics of the models. Every Student has a **faculty number**. It should not be less than **5 digits/letters** and no more than **10 digits/letters**. That’s it, we are done with the **Student** model. Next step is to model the **Worker**. Every **Worker** has a **week salary**, **work hours per day**. It should be able to calculate the money he earns by hour.

Create a **Student** and a **Worker**. On the first input line you will be given **student** **names** and **faculty number**. On the second input line – the **worker’s first name, last name, salary** and **working hours**. Take a look at the example for better understanding. Collect them correctly to create new objects of type Student and Worker and print their info as it is shown below. If you have done the things right, your output should be the same as the expected output in the table.

### Constraints

* **First name** should start with **capital letter**. If it does not match the rule, print:

**"Expected upper case letter! Argument: firstName"**

* If the **first name’s length** is **less than** **4 symbols**, print:

**"Expected length at least 4 symbols! Argument: firstName"**

* **Last name** should start with **capital letter**, if it does not match the rule, print:

**" Expected upper case letter! Argument: lastName"**

* If the **last name’s length** is **less** **than** **3 symbols**, print:

**"Expected length at least 3 symbols! Argument: lastName"**

* Every **faculty number** should be in the **range [5, 10] symbols** and consists only from **digits/letters**. If it does not, print: **"Invalid faculty number!"**
* Every **worker’s last name** should be **more than 3 symbols**, if it doesn’t, print:

**"Expected length more than 3 symbols! Argument: lastName"**

* Week **salary** should be **more than 10**, if it doesn’t, print:

**"Expected value mismatch! Argument: weekSalary"**

* Working **hours** should be in the **range [1, 12]**, if they are not, print:

**"Expected value mismatch! Argument: workHoursPerDay"**

### Example

|  |  |
| --- | --- |
| **Input** | **Output** |
| Ivan Ivanov 08  Pesho Kirov 1590 10 | Invalid faculty number! |
| Stefo Mk 081211008  Ivcho Ivancov 1590 10 | "Expected length at least 3 symbols! Argument: lastName" |

## \*Mordor’s Cruelty 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 0 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 have 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 single line: all the Gandalf`s foods he has eaten.

### 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 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, banica,Melon!\_,HonEyCake, !HoneYCake,hoNeyCake\_; | 7  Happy |
| gosho, pesho, meze, Melon, HoneyCake@; | -5  Sad |
| HoneyCake honeyCake honeyCake HoneyCakE HoneYCake HonEyCake HoneyCake HoneyCake HoneyCake HoNeyCake | 50  JavaScript |

## 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 adding. 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  ABBA;Mamma Mia;3:35  Nasko Mentata;Shopskata salata;4:123  Nasko Mentata;Shopskata salata;4:12 | Song added.  Song seconds should be between 0 and 59.  Song added.  Songs added: 2  Playlist length: 0h 7m 47s |
| 5  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;14:59  Nasko Mentata;Shopskata salata;0:5 | Song added.  Song added.  Song added.  Song added.  Song added.  Songs added: 5  Playlist length: 1h 0m 1s |

## \*Animals

Create a hierarchy of **Animals**. Your task is simple: there should be a base class which all others derive from. Your program should have 3 different animals – **Dog**, **Frog** and **Cat**. Let’s go deeper in the hierarchy and create two additional classes – **Kitten** and **Tomcat**. **Kittens are female and Tomcats are male!** We are ready now, but the task is not complete. Along with the animals, there should be and a class which classifies its derived classes as sound producible. You may guess that all animals are sound producible. The only one mandatory functionality of all sound producible objects is to **produceSound()**. For instance, the dog should bark.

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

On the console, you will be given some lines of code. Each two lines of code, represents animals and their names, age and gender. On the first line there will be the kind of animal, you should instantiate. And on the next line, you will be given the name, the age and the gender. Stop the process of gathering input, when the command **“Beast!”** is given.

### Output

* On the console, print for each animal you’ve instantiated, its info on two lines. On the first line, print:  
   {**Kind of animal**} {**name**} {**age**} {**gender**}
* On the second line, print: {**produceSound()**}

### Constraints

* Each **Animal** should have **name**, **age** and **gender**
* **All** **properties**’ values should **not be blank** (e.g. name, age and so on…)
* If you enter invalid input for one of the properties’ values, throw exception with message: **“Invalid input!”**
* Each animal should have a functionality to **produceSound()**
* Here is example of what each kind of animal should produce when, **produceSound()** is called
  + **Dog: “BauBau”**
  + **Cat: “MiauMiau”**
  + **Frog: “Frogggg”**
  + **Kittens: “Miau”**
  + **Tomcat: “Give me one million b\*\*\*h”**
  + **Message from the Animal class: "Not implemented!"**

### Examples

|  |  |
| --- | --- |
| **Input** | **Output** |
| Cat  Macka 12 Female  Dog  Sharo 132 Male  Beast! | Cat Macka 12 Female  MiauMiau  Dog Sharo 132 Male  BauBau |
| Frog  Sashky 12 Male  Beast! | Frog Sashky 12 Male  Frogggg |
| Frog  Sashky -2 Male  Beast! | Invalid input! |