Exercises: Inheritance

This document defines the exercises for "Java OOP Basics" course @ Software University. Please submit your solutions (source code) of all below described problems in https://judge.softuni.bg/Contests/226/Inheritance-Exercises.

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:

- **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.
 - o Children should **not** be able to have age **greater than 15**

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 lesser than 15!"
- If the name of a child or a person is less than 3 symbols exception's message is: "Name's length should not be less than 3 symbols!"

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();
    int 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(int age) {
   // TODO
```

Step 3. Define a Constructor

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

```
Sample Code
public Person(String name, int 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(int age) throws IllegalArgumentException {
    if (age < 1) {
        throw new IllegalArgumentException("Age must be positive!");
    }
   // TODO: Set the age
```

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() {
    return String.format("Name: %s, Age: %d",
                    this.getName(),
                     this.getAge()));
```

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, int 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(int age) throws IllegalArgumentException {
    //TODO: Validate the age
    super.setAge(age);
}
```















Problem 2. Book Shop

Your program should have two classes – one for the ordinary books – **Book**, and another for the special ones – GoldenEditionBook.

- 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 has two names and the 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[] 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 | IllegalClassFormatException error) {
        System.out.println(error.getMessage());
    }
}
```

Example

Input	Output
Ivo 4ndonov	Author not valid!
Under Cover	
99999999999999999	

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) {</pre>
        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 JAVA 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();
```

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.

```
public GoldenEditionBook(String author, String title, double price) {
    //TODO : Reuse base 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.

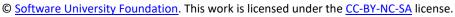
Back to the **GoldenEditionBook** class, let's override the Price property and change the getter body.

```
Sample Code
@Override
public double getPrice() {
    return super.getPrice() + super.getPrice() * 0.3;
}
```

Problem 3. Mankind

Your task is to model an application. It is very simple. The mandatory models of our application are 3:



















- Human
- Worker
- Student

The parent class - Human should have first name and last name. Every student has a faculty number. Every worker has a week salary (7days) 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 and the info about the worker in the given formats:

Print the student info in the following format:

```
○ First Name: {student's first name}
```

o Last Name: {student's last name}

o Faculty number: {student's faculty number}

Print the worker info in the following format:

o First Name: {worker's first name}

o Last Name: {worker's second name}

○ Week Salary: {worker's salary}

○ Hours per day: {worker's working hours}

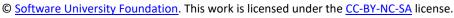
Salary per hour: {worker's salary per hour}

Print exactly **two digits** after every double value's decimal separator (e.g. 10.00)

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 4 or more than 4 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 3 or more than 3 symbols	"Expected length at least 3 symbols!Argument: lastName"
Faculty number	Should be in range [510] symbols	"Invalid faculty number!"
Worker last name	Should be more than 3 symbols	"Expected length more than 3 symbols!Argument: lastName"
Week salary	Should be more than 10	"Expected value mismatch!Argument: weekSalary"













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: 22.71

Problem 4. *Mordor's Cruelty Plan

Gandalf the Gray is a great wizard but he also loves to eat. The food, however, makes him loose his capability of fighting the dark. Mordor's orcs have asked you to design them a program which is calculating Gandalf's mood. This way they could **predict the battles** between them and try to beat The Gray Wizard.

When Gandalf is hungry he cannot fight well. Because the orcs have spies, they know the foods that Gandalf is eating and the result on his mood after he consumed a food:

- 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;

Model an application which is calculating his happiness points.

Input

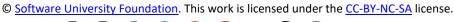
The input comes from the console. It will hold single line: all the foods Gandalf has eaten.

Output

On the first line, print the total happiness points Gandalf currently has.

On the second line – print his mood.



















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.

Examples

Input	Output
Cram melon honeyCake Cake	7
	Нарру
gosho pesho meze gosho pesho meze	-6
	Angry

Problem 5. 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

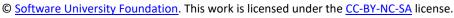
Validation

- 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
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.
Nasko Mentata;Shopskata salata;14:59	Song added.
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

Note: you can create another class Playlist to store songs with public method for calculating total playlist length.

Problem 6. *Animals

Create a hierarchy of Animals. Your task is simple: there should be a base class Animal 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 three lines. On the first line, print: {Kind of animal}
- On the second line, print: {name} {age} {gender}
- On the third 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
 - o Dog: "BauBau"
 - o Cat: "MiauMiau"
 - o Frog: "Froggggg"
 - o Kitten: "Miau"
 - o Tomcat: "Give me one million b***h"
 - o Message from the Animal class: "Not implemented!"

Examples

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



