Exercises Module 17 - Inheritance

Task 1

One of the methods that all objects inherits from the superclass Object is the toString method. This method should return a string that is a good representation of the state of the object, that is the values of the most important variables.

The toString method is automatically called when an object is given as an input argument to the System.out.println method.

Try to print one of the book objects by calling the System.out.println method with the book object as the input argument.

Was the result something like this: com.company.Book@60e53b93?

The method that is inherited from Object doesn't know anything about how to return a good representation of the Book object, it only returns the object reference. This is the best it can do.

For the toString method to be useful we need to override it and provide an implementation that makes sense in for a Book object.

Override the toString method by adding a public method with the name *toString* to the Book class with a return type of String, and no input arguments. Create an implementation of the method by returning a string that is a good representation of the Book object by using the values in the variables of the book.

It's a good convention to also add an Override annotation above the overridden method like this:

```
@Override

public String toString() { ...}
```

Then try to print the book object again by sending it as the input argument to the System.out.println method.

This time it should look a little better!

Task 2

Some books have special requirements. For childrens books for example, we also need information about the recommended age of the book.

We could create a new class called *ChildrensBook* that has the variables title, author, price and recommendedAgeInfo. But then we would have some duplicated code since this would be the same code as a normal *Book* with the addition of the *recommendedAgeInfo* variable.

Lets instead create a new class that inherits the Book class and extends it with one more variable - the recommendedAgeInfo. This way we don't have to declare the same variables again. We reuse the functionality from the superclass.

Your task is to create a new class ChildrensBook that extends Book and has a private variable *recommendedAgeInfo* of type String with public getters and setters.

If you have no default constructor in the Book class, you will notice that you either need to create one in the Book class, or create a constructor in the ChildrensBook class that corresponds to one of the constructors in the Book class. Create a constructor in the ChildrensBook class that takes title, author, price and recommendedAgeInfo as arguments. From the first line in the constructor, call the constructor in the superclass with the keyword super with the arguments title, author and price. Assign the recommendedAgeInfo argument to the instance variable recommendedAgeInfo

Create a ChildrensBook in the main method with a title, author, price and a recommendedAgeInfo of for example "from 4 years". Then add the ChildrensBook to the List of books. It is OK to add it to the list since a ChildrensBook is also a Book.

Task 3

We need a productId in the Book as well as in the Movie. Should we add one more variable in the Book class or could we use inheritance to be able to reuse code and avoid code duplication?

If we compare the Book class and the Movie class we see that we have several similar variables. Besides the productId, we have the title and price variables that are identical. Maybe we could put these in a superclass and let both Book and Movie inherit from this superclass? What name should we give this superclass?

Create a class with the name Product. Add a variable productId to the Product class. Also add a price variable since we are sure that all our products will have a price. But should we also put the title variable in the Product class? Maybe we would like to use this Product superclass for all our products in the store and not all products have a title. Lets' skip the title for now and only add the productId and the price variables with public getters and setters.

Then let the Book class extend the Product class. We can delete the getter and setter methods for price in the Book class since it now extends the Product class and gets the variables from the superclass. Or is there a problem with the price variable in the constructor of the Book?

If you declared the variables in the Product class as private, they are not accessible in the subclass. See what happens if you change between private and protected access to the variables in the Product class. We need the subclass to have access to the variables in the superclass so we have to declare them as protected in the superclass.

Now we have a productId also in the Book and not only in the Movie. But let's reuse the variables in the Product class by extending the Product class also in the Movie class. Remove the price and productId

variables and the getters and setters in the Movie class.

By introducing the Product superclass we now reuse some of the variables that is used in both books and movies.

Task 4

We can use inheritance not only to reuse variables and methods, but also as a way to handle objects in a more general way.

There is no need to have two different lists, one with books and one with movies, since both are now products.

Instead, create a new list of products and add all the books and movies to the same list. Since they extend Product they can be added to a generic list of type Product.

Also add a for-each loop to iterate over the new product list and remove the two old loops where we iterated over books and movies to print out their details. Use this new loop that iterates over the product list to print the book details and movie details.

But how do you access the printBookDetails or printMovieDetails from a Product? You can iterate over the products and get each product but you can only access the variables that are declared in the Product class. When handled in a more general way as a Product, you can only access the more general methods in the Product class. The methods are still there, they haven't disappeared, it's just that they are not accessible through the reference type that is now of type Product.

To be able to access a methods inside the object that is really a book or a movie, but is references as a Product, we can make an explicit cast from the Product class to the Book or Movie class. But how do we now it a Product is really a Book or a Movie?

Try to cast every Product to a Book to be able to access the printBookDetails method. If you have any movies in the list you will get a ClassCastException when trying to cast a Movie to a Book.

You can solve this problem in the loop by first checking the class with the instanceof operator, like this:

```
if (product instanceof Book) {
    Book book = (Book)product;
    book.printBookDetails();
}
```

By checking if the product is really a book, you can safely cast the product to a Book reference and then call the printBookDetails on the book object.

Check with an else-if if the product is a movie, and in that case cast it as a movie to be able to print out the movie details