

Exercises Module 19 - Interfaces

Task 1

What happens if we remove the `printDetails` method in the `Movie` class? Try to rename it to `printDetails2` or anything else than `printDetails`.

IntelliJ will mark the `@Override` annotation as an error since this method isn't overriding any method in a superclass. Delete the `@Override` annotation to keep IntelliJ happy.

Try to run the program.

It should be running since there is still a `printDetails` method in all movies, because they inherit this method from the superclass `Product` that has a `printDetail` method, if they don't provide their own implementation by overriding it.

If we add new products we might forget to override this method and then the method in the superclass will be executed. But do we want our program to behave like this? Is it necessary for these other products to be printed out by a method that only knows about the `productId` and the price? If not then we would like to force all products to implement their own version of this method.

We don't want to use inheritance to reuse methods in the superclass anymore, we want to make sure the method is implemented in all subclasses so that we can handle all product generally in the main method by calling `printDetails` on all products, but we want every subclass to be forced to implement their own method.

We can make this happen by removing the implementation of the `printDetails` method in the `Product` class and making it abstract. Do this and the method in `Product` should look something like this:

```
public abstract void printDetails();
```

If the name of the method is still changed in the `Movie` class, IntelliJ will now complain over the `Movie` class not overriding the abstract method `printDetails`.

Change the name back to `printDetails` in the `Movie` class to make IntelliJ happy again.

Run the program and everything should work like before.

We have accomplished a program that can call the `printDetails` method generally on all products, but that forces all subclasses to provide their own implementation of the method. This is done by having an abstract class with an abstract method.

Task 2

Remove all other methods in Product except the abstract printDetails method. If you have getters and setter to productId and price, remove them from Product. If you use them somewhere in the code, add them instead to the Book and the Movie class. You should now have a Product class that is abstract with only abstract methods (the only method being printDetails). It should look like this:

```
public abstract class Product {  
    protected long productId;  
    protected int price;  
  
    public abstract void printDetails();  
}
```

An abstract class with only abstract methods is similar to an interface. Let's see what it would take to change the Product class into an interface instead of an abstract class.

This will still make it possible to handle all products at a general level, but we will not be able to implement the productId and the price variables in the Product when it is an interface.

Some hints:

Change the "abstract class" of Product to "interface" like this:

```
public interface Product
```

As you can see it is not possible to keep the variables anymore. Remove the variables from Product, and add them to both the Book and Movie class.

Also you can't extend Product anymore in the Book and Movie class since it isn't a class anymore. You now have to *implement* Product instead like this:

```
public class Book implements Product {
```

Try to run the program. It should work, and as you can see in the main method it is ok to handle the products as the type Product even if this is now an interface instead of a class.

An interface is used to force the classes implementing it to conform to its form, and implementing all the methods that it dictates. This way the code in the main class knows that it can call a method called printDetails without knowing the class of the object.

Task 3

Now we are going to use an existing interface to be able to use a method in the Collections class that helps us sort the object in a list.

Since Product is now an interface, we will in this task only sort movies in a list of movies.

Add a few movies to a list of movies and add them in an order that makes them unordered by productId.

You could create a list of movies and add the movies to this list instead of adding them directly to the

ed could create a list of movies and add the movies to this list instead of adding them directly to the products list, and then add the whole list of movies to the product list by using the `addAll` method and provide the whole list of movies like this (if you want the printouts of all products to stay intact):

```
List<Movie> movies = new ArrayList<>();
movies.add(ghostBusters);
movies.add(shawshank);
movies.add(lordOfTheRings);

products.addAll(movies);
```

Now you have a list with only movies, and they should not be added a way that makes the movie list unordered by `productId`.

To use the `sort` method in the `Collections` class, add this line in the `main` method (before adding the movies list to the product list if you want to see the result of the sort):

```
Collections.sort(movies);
```

As you can see this method does not accept our list of movies since it expects a list of `Comparable`. `Comparable` is an interface that forces the classes implementing it to implement a `compareTo` method. By accepting a list of `Comparable` as input argument the `sort` method knows that it can always call a `compareTo` method on all the objects in the list.

The `Comparable` interface need the type of the class just like a `List` or a `Map`, like this:
`Comparable<Movie>`.

Let's make our `Movie` class implement `Comparable` as well as the `Product` interface. A class can implement many interfaces by just adding them comma separated after the keyword `implements`, like this:

```
public class Movie implements Product, Comparable<Movie> {
```

Now IntelliJ tells us that the `Movie` class must implement the `compareTo` method to follow the rules of the `Comparable` interface.

Lets do that by adding a method `compareTo` with this signature:

```
@Override
public int compareTo(Movie movie) {
```

This method is returning an `int`. The `int` should be 0 if the `productId` of the object itself is equal to the `productId` of the `Movie` sent in as a method argument. And it should be -1 if the `productId` is less than the one in the input argument, and it should otherwise return 1.

This is the implementation of the `compareTo` method, it returns -1 or 0 or 1 to tell if the object is less than, equal or bigger than the object sent in as the input argument.

Try to implement this functionality in the `compareTo` method, and try to run the program. Comment out the `Collections.sort` line to check the result without sorting.

Did you succeed with sorting the movies?

Task 4

We will continue to look at inheritance vs composition. The example is taken from a blog written by Mattias Petter Johansson who is Swedish Javascript developer with his own youtube channel "Fun Fun Function". The example is rewritten with java code.

Link to the original blog with Javascript:

<https://medium.com/humans-create-software/composition-over-inheritance-cb6f88070205> 

(<https://medium.com/humans-create-software/composition-over-inheritance-cb6f88070205>)

We start by creating two animals, one cat and one dog.

```
class Cat {  
    public void meow() { /*...*/}  
}  
  
class Dog {  
    public void bark() { /*...*/}  
}
```

Because nature calls, we add .poop() to the Cat and the Dog class:

```
class Cat {  
    public void meow() { /*...*/}  
    public void poop() { /*...*/}  
}  
  
class Dog {  
    public void bark() { /*...*/}  
    public void poop() { /*...*/}  
}
```

Now we have code duplication. We create a superclass for the poop() method and let the Cat and Dog inherit this class.

```
class Animal {  
    public void poop() { /*...*/}  
}  
  
class Cat extends Animal {  
    public void meow() { /*...*/}  
}  
  
class Dog extends Animal {  
    public void bark() { /*...*/}  
}
```

Now we also need cleaning robots that can clean up after our cats and dogs.

```
class CleaningRobot {
```

```
public void clean() { /*...*/}

public void drive() { /*...*/}

}
```

The management also wants us to build killer robots in order to "discipline" the animals that do not behave well.

```
class MurderRobot {
    public void kill() { /*...*/}
    public void drive() { /*...*/}
}
```

Here again we see code duplication, so we solve this problem with one more superclass.

```
class Robot {
    public void drive() { /*...*/}
}

class CleaningRobot extends Robot {
    public void clean() { /*...*/}
}

class MurderRobot extends Robot {
    public void kill() { /*...*/}
}
```

So now we have the following classes:

```
class Animal {
    public void poop() { /*...*/}
}

class Cat extends Animal {
    public void meow() { /*...*/}
}

class Dog extends Animal {
    public void bark() { /*...*/}
}

class Robot {
    public void drive() { /*...*/}
}

class CleaningRobot extends Robot {
    public void clean() { /*...*/}
}

class MurderRobot extends Robot {
    public void kill() { /*...*/}
}
```

Everything is fine for a while, then one day the management comes and says they want a killer-robot-dog that should be able to drive(), kill() and bark(), but it should not be able to call poop() because it is a robot. How do we solve this in a good way with the structure we have?

Here we can use composition and interfaces to solve our problem instead of inheritance! Inheritance is used when you want to design your classes according to what they *are*, and composition is used when you want to design your classes according to what they can do.

We want classes that behave like this:

```
Cat           = Pooper + Meower
Dog           = Pooper + Barker
CleaningRobot = Driver + Cleaner
MurderRobot   = Driver + Killer
MurderRobotDog = Driver + Killer + Barker
```

With the help of these interfaces we can get our classes to behave in that way!

```
interface Pooper {
    void poop();
}

interface Meower {
    void meow();
}

interface Barker {
    void bark();
}

interface Driver {
    void drive();
}

interface Cleaner {
    void clean();
}

interface Killer {
    void kill();
}

// Our classes:

class Cat implements Pooper, Meower {
    @Override
    public void poop() { /*...*/}

    @Override
    public void meow() { /*...*/}
}

class Dog implements Pooper, Barker {
    @Override
    public void poop() { /*...*/}

    @Override
    public void bark() { /*...*/}
```

```

}

class CleaningRobot implements Driver, Cleaner {
    @Override
    public void drive() { /*...*/}

    @Override
    public void clean() { /*...*/}
}

class MurderRobot implements Driver, Killer {
    @Override
    public void drive() { /*...*/}

    @Override
    public void kill() { /*...*/}
}




class MurderRobotDog implements Driver, Killer, Barker {
    @Override
    public void drive() { /*...*/}

    @Override
    public void kill() { /*...*/}


    @Override
    public void bark() { /*...*/}
}

```

So should you use interfaces or inheritance? This is an ongoing discussion and people have different opinions. In other languages, such as C ++, the concept of interfaces doesn't exist, instead abstract classes are used. Many people prefer composition over inheritance. Read the original article or some of these articles for more information:

- http://blogs.perl.org/users/sid_burn/2014/03/inheritance-is-bad-code-reuse-part-1.html 
(http://blogs.perl.org/users/sid_burn/2014/03/inheritance-is-bad-code-reuse-part-1.html)
- <https://www.quora.com/Is-inheritance-bad-practice-in-OOP-Many-places-that-teach-design-patterns-say-to-opt-for-composition-over-inheritance-but-what-about-when-multiple-classes-share-logic-from-an-abstract-class-such-as-in-the-Template-Method-design-pattern> 
(<https://www.quora.com/Is-inheritance-bad-practice-in-OOP-Many-places-that-teach-design-patterns-say-to-opt-for-composition-over-inheritance-but-what-about-when-multiple-classes-share-logic-from-an-abstract-class-such-as-in-the-Template-Method-design-pattern>)
- <https://softwareengineering.stackexchange.com/questions/134097/why-should-i-prefer-composition-over-inheritance> 
(<https://softwareengineering.stackexchange.com/questions/134097/why-should-i-prefer-composition-over-inheritance>)

More information:

- Inheritance and Composition (Is-a vs Has-a relationship) in Java
<https://www.baeldung.com/java-inheritance-composition>  (<https://www.baeldung.com/java-inheritance-composition>)

