Week 04: Deserialized our Media and functional programming

Challenge 01: Save our Media playlist (45 mins)

There is an essential feature missing from our streaming platform – it is puzzling that we are only addressing this issue now, but to be fair, we have not yet covered serialization and deserialization until this week. Although it can be seen as simply reading and writing conventional streams, handling objects is a “little” bit different (as you’ve experienced this week).

Fittingly, if you are a digital media streaming provider, your subscribers should be able to save their media library (if they prefer not to download these media to their devices), and we should allow them to have instant access. So, let’s implement object serialization and deserialization to our program.

TODO#1 – In the Subscriber class, we need to implement a method `saveMedia()` that will save all the media object into a serialized file (given a storage path).

TODO#2 – Of course, we then need to implement another method to deserialize the Media object that we’ve previously saved. You can do with the `getMedia()` method.

TODO#3 – In order to ensure, that the Media objects are properly save, we need to implement Serializable – pop quiz, where should we do this?

That is not difficult at all 😊 By completing this challenge, you now have covered most of the Java Input / Output streams library – hmmm, we might be forgetting one more, Java 8 streams, but this is nothing like the conventional I/O stream that we’ve discussed. More on this next week.

Challenge 02: Back-end Utility Functions – with Functional Interfaces (45 mins)

Thus far, most of the features that we’ve implemented in `YourPrime` are intended for the subscribers. However, enterprise application also deals with the owner (this is us!). We need to be able to manage our subscribers, as well as analysing the performance of our streaming platform.

Since you are now well beyond the ordinary Java OOP programming style, we are going to do this in functional programming style by utilising functional interfaces. This will hopefully blow your mind! If we compared it with functors, the implementation is relatively easier – even though it is the same concept, but functional interfaces are easier to the eyes (much easier).

We are going to write a few administrative utility function (hence the package name `adminSite`). You might have noticed that you can write the same codes using conventional Java OOP style, which is nothing wrong, but if you can do complex things much easier, why in the world you don’t want to do that? Having said that, it is also important to point out, that if you can do complex things with simple codes, that is more preferrable (remember, K.I.S.S. you should also check out here). Let’s get into it:

TODO#1 – We need to be able to add new subscriber to our platform, and to do this we need a method (appropriately) named `addSubscriber(Subscriber)`. Notice we will pass a subscriber object in the argument. We can use the relevant built-in Map collection method to help us to add a new user to our database (i.e., the map object in this challenge)

** You need to create a supplier method that will generate a new user Id for the new subscriber before we can add the subscriber into the database.

TODO#2 – Another important function is to allow our subscribers to change their password. Ideally this should be encrypted – but we are going to keep it as it is here (you can explore this yourself, we've already discussed encryption during the last lab session). Since we are using Map, you can also use the appropriate built-in method to do this.

TODO#3 – If you can add subscriber, you should also be able to remove your subscriber (we don't want to do this, but if our subscribers are unhappy, we have to oblige, besides, we are freeing storage server space – which is vital in this business). Use the relevant built-in method in Map collection to do this.

TODO#4 – Hang on a minute, this is turning into a programming challenge for Map collection. Well, not quite because technically, those built-in methods are built on functional interfaces, but anyway, let's write our own functional interface (predicate) for `searchSubscriber(String)`. Obviously, this is to search for existing subscriber using a String keyword that should match any user id or subscriber's name.

TODO#5 – We are going to explore a bit more, this time complete the `calculateOverdueFees()` method using the predicate interface. The method will return the total fees calculated for each media belonging to the subscriber.

TODO#6 – Lastly, we need a print-out info method in the form of the subscriber's name and their outstanding fees amount. This is necessary, since our original `toString()` overriding method in `Media` prints way to much information. The management is not interested in looking at all those details.

By the completion of these TODOs, you'll realise how easy it is to code in the functional way (of the force – just joking). You can take it even further by going fully (darkside of the force – joking again) functional with lambda expression and Java 8 Streams. These will be the last topic that we'll cover in your advanced Java portion of the module (and trust me, you need to check out the learning materials – especially the coding session *wink wink).

May the force be with you …