Q1.

Here, I have decided to use the *Composition Design* pattern. I have created two classes called DisjunctionFilter and ConjunctionFilter, these classes implement the WatchListFilter and are a Composite class that we can use to combine other WatchListFilter with logical AND or OR. In my design the WatchListFilter interface is the Component, the Filter Classes are the Leafs, and the DisjunctionFilter and ConjunctionFilter classes are the composites.

Since we don't need to dynamically add filters, and also maintaining the immutability of the filters would be cumbersome with an add method, I have decided to just create a constructor for the classes and not add an add() method and store the filters in a private final ArrayList<WatchListFilter>. I have also created two filter classes to filter Watchables based on their Studio and Episode based on their EpisodeNumber. I will be continuing to use the generalWatchlist() from the Library class, for the actual filtering process of a Library. I thought about creating another class that takes in an ArrayList of Watchables and a Filter and returns a filtered Watchlist so that we can also build on it later using the *Decorator pattern*, but for this assignment, the Library method made more sense to me since we want to filter a Library.

You might notice that I have made some changes to the WatchListFilter interface, before it used to have 3 methods to filter different Watchables one for each Watchable we had in our code (Episode, Movie, TVShow). Since the assignment is asking us to filter Watchlists and the interface is called WatchListFilter, I figured that the interface should only be dealing with the common functionalities to Watchables and since Episode, Movie, TVShow is all watchable there is no point in creating a different method to deal with each Watchable and just having one method that filters a Watchable is enough.

Q2.

Here, in the TVShow class, I have created a private field that stores an Episode prototype and is initialized to a default value with no custom info. Since we are assuming that the clients can't create an Episode, they can set the custom info they want to add to the prototype by using the setPrototype method and passing the custom info they want in two Maps. I have decided to store the prototype in the TVShow class as an Episode rather than two Maps passed by the client because: we can build on the prototype later on if we want to prototype other things like titles and paths, and storing the 2 Maps doesn't seem elegant to me. The client can use the addEpisodeFromPrototype method to create an Episode based on the prototype. I have also created an extra constructor in the Episode class that takes an Episode and some extra inputs and creates a new episode based on the Episode passed to it, in this case, it only takes the custom info. I have created a Prototype interface that implements Cloneable, however, I'm not using clone in this case because cloning the prototype Episode would mean that the path gets copied as well, and being able to change the path would mean I need a setter that would compromise the class immutability, however, I decided to create the interface and write clone() to be complete.

Please note that I do understand that an alternative approach to this problem would be to store the custom info directly in the class rather than an Episode prototype and just add the custom info to an Episode being created, but I was torn between the two methods and I have some reasons for my choice that we can discuss in the live session.

Q3.

I have created test cases for the work that I have done in Q1 and Q2. I have created a test for the classes TVShow and Episode and a set of cases for the Watchlist Filtering Functionality of the code (not based on a specific class). I have written tests based on both white box and black box methods and have commented the techniques used for some cases as an example.