<u>Chapter 12 Important Notes: lambdas and streams in the what concept not the how one :</u>

- ✓ There is for each method inside the list class and I takes lambda expression to tell it the behavior you want it to follow
- ✓ The benefit of the for each rather than using the for loop is that for each focus on telling the compiler what to do not focus on how to make that and the second thing using it reduce the errors that may happened when you use the for loop cause just small thing in handling the for loop details will lead you to an error and sometimes an exception
- ✓ Java 8 introduces Streams that the stream API is coming in java 8 to make dealing with the methods in more easy way and that in just by using the lambda expressions and that makes the code more shorter and easy to maintain and less erroring may happen cause it is just care about telling the method what to do and we do not do anything related to the how of the method
- ✓ So the streams API is set of operations we can perform on a collection so when er read these operations on the code we can understand easily what we are trying to do without entering in the details of how we can make it
- ✓ Streams and lambda expressions were introduced in Java 8
- ✓ There is 2 rules when we will use any stream that the generics on it if we find <T > that means the data type of the object in the stream but <R> that means the data type of the result of the stream
- ✓ Firs thing to can use the stream methods the first thing we need to make is to make a stream object like that :
  List<String> strings = List.of("I", "am", "a", "list", "of", "Strings") ;
  Stream<String> stream = strings.stream() ; → Source collection
  And then now after making that we can normally using any stream methods easily:
  Stream<String> limit = stream.limit(4); → intermediate operations
- ✓ We know somethings like streams , intermediate operations, terminal operations all of this if we put them together it give us STREAM PIPELINE we need at least to make the source collection and the terminal operations steps to can use the API stream and in fact we do not need to assign each step into its own variable like we make in the previous code cause the operations are designed to be chained so we can use it easily in just one line like that :
  List<string> result= strings.stream().limit(4).collect(collestors.tolist());
  And this stream pipeline represents a query on the original collection

- ✓ Every intermediate operation acts on a stream and return stream that means we can call as many as we want from this intermediate ones before calling the terminate operation in the end
- ✓ Streams are lazy (Me Too) and that cause It is collecting all the intermediate operations together and once it collecting them all when you call the terminal operation it apply all of this once in the data and this makes it is most efficient way rather than iterate over the original collection for each and every intermediate operation
- ✓ You can not reuse streams once you call and run the terminate operation you can not call the stream again and run it again something like that :
  Stream<String> limit = strings.stream(). Limit (4) ;
  List<String> result = limit.collect(Collectors.toList()) ;
  List<String> result2 = limit.collect(Collectors.toList()); → this will give you an error telling you that the stream is already closed
- ✓ Stream operations doesn't change the original collection so you will not need to worry about it
- ✓ We said before we can adding as many as intermediate operations in the same chain but there is note you must give it attention you need to remember that not just the computers have to read and understand your code it is humans too so if the stream pipeline too long you can divide it and assigning in variables and give them understood names to make it easy to read and maintain

- ✓ You don't have to write detailed code telling the JVM exactly what to do and how to do it. You can use library methods, including the Streams API, to query collections and output the results.

- ✓ Lambdas are not magic they are just classes like everything else

- ✓ A lambda expression might have more than one line and in this case you will need to put them in {} like normal methods code

- ✓ Originally the only kind of methods allowed in interfaces were the abstract methods cause the interface can be 100 % pure abstract class and this abstract methods needs to be overridden by any class that implement this interface but java 8 making interfaces can contain default and static methods too and default and static methods both have a method body with defines behavior and any methods in the interface and not defined as default or as static methods this methods are considered abstract methods and must be overridden

- ✓ And why we know that cause in the functional interfaces they are the interfaces that having a single abstract method only and can be implemented with a lambda expression

- ✓ The map operation in the stream API its goal it to convert something from specific type to another specific type

- ✓ The power of the Streams API is that we can use it to build up complex queries with understandable building blocks

- ✓ There is another solution rather than using the lambda expressions and it is the method reference :: and it can replace the lambda expression but you do not have to use them but its benefit is it sometimes make the code easier to understand and this is an example for using it rather than the lambda:

  Function<Song, String> getGenre = Song: :getGenre;

- ✓ Rather than using the duplicate intermediate operation you can store the result and the collectors to convert it to set directly which known about it that it is storing only the distinct things
- ✓ Do you want a list that will not be changed like no elements can be added , replaced , removed ? the solution for that is using Collectors. toUnmodifableList instead of using Collectors.tolist but ‘This is only available from List Java 10 onward and this unmodifiable function is available for set and map too not just for list
- ✓ There is Collectors.joining terminate method and It is using to joining the all stream elements in just one single string and you can define delimiter like comma for example to separate between each element and this can give you String Comma Separated Values (CSV)
- ✓ There is even more terminal operations not just the collect one there is operations like count , checking if specific something exists or no , find a specific thing and more and some of them depend upon the type of stream you are working with and remember the API documentation will be helpful if there is operations already does what you want to do
- ✓ Since java 8 the methods are normally declared as sometimes it might not return a result and that’s mean they will return optional and there is object wraps the result and you need to check about it if it is having result or it is empty to can know what you will do next

- ✓ And the optional in the last one it means there is class called OPTIONAL and we make an object from it and check about it is result and the important thing about the optional results that it can be empty so that you must beed to check about it first to see if it is empty or no cause if you do not do that you may find an exception in your face