Chapter 11 Important Notes : Data Structures

- There is something called MOCKING and that's when you want to write some temporary code that stands in for the real code that will come later like in this chapter our real code will treat with the file itself to read data from it but now we do not have a file and want to test the functions we have so we make this mocking (و هي معناها اصلا انك بتقلد الحاجه لان الحاجه الاصليه مش معاك) and that by using static data for example and then when the real code comes later in this case we can replace the new real code with the mocking one
- Cause java over time has evolves to remove unnecessary code duplication from its syntax cause the compiler can figure out the type so you always do not need to write it out in full something like this line
  ArrayList<String> songs = new ArrayList<String>() ;
  We can write it like that
  ArrayList<String> songs = new ArrayList< >() ;
  And the compiler will still understand what you want to do and without any duplicated code
  And this is called DIAMOND OPERATOR and may be you need to write the full syntax and that's for only one reason is that to make your code more readable

- The first way to sort we can use is by using COLLECTIONS.SORT(list) and pass to it the thing we want it to be sorted and this function will sort them according to this things first by special characters , numbers , capitals , lower cases

- Now there is something called GENERICS and what is generics is that the type we define it for the array list like ArrayList<Cat> this is generics and its goal is to make more safety and that cause before generics we define only the array list without any type so the compiler knows that this is array list of type object so you can put any type of objects inside this array list but the problem will appear when you try to get any object from this array list so you will get this object and store it in CAT object but you may find an exception in your face and that's cause maybe you add dog object inside this arraylist and that what you get now and the compiler will not alert you cause all if this are objects so when you use the generics this will save you from this things cause it will catch thins things in the compile time rather than the run time

- in generics, ‘extends’ means ‘extends or implements which make it works for both classes and interfaces

- there is a second sort method on the collections of java api that takes a comparator and its use is when you want to sort the same thing in more than one way

- and there is difference between the comparable and the comparator the comparable is interface and the class that we will sort its objects must implements this comparable interface and then inside it you override the compare to method and write your sorting logic and its cons here the sorting logic is just one option and will still fixed until you change the logic of the compare to manually but what about if you want to sort in different ways that means different logics? Here comes the benefit of the comparator and it is also an interface and you build a class for each sorting logic you want and make this class implements this comparator logic and then inside it you make an override the compare function … like this codes :

```
class Song implements Comparable<Song> {
  String title;
  String artist;
  int bpm;

  Song(String title, String artist, int bpm) {
    this.title = title;
    this.artist = artist;
    this.bpm = bpm;
  }

  // طريقة المقارنة الافتراضية (بالـ title)
  @Override
  public int compareTo(Song other) {
    return this.title.compareTo(other.title);
  }

  @Override
  public String toString() {
    return title + " - " + artist + " (" + bpm + " BPM)";
```

```
  }
}

import java.util.*;

class ArtistComparator implements Comparator<Song> {
  @Override
  public int compare(Song s1, Song s2) {
    return s1.artist.compareTo(s2.artist);
  }
}

class BpmComparator implements Comparator<Song> {
  @Override
  public int compare(Song s1, Song s2) {
    return Integer.compare(s1.bpm, s2.bpm);
  }
}
```

- which one is better? The better approach would be to handle all of the sorting definitions in classes that implement comparator cause by using the comparator you make another classes outside your main class which makes your code easy to maintain and extendable and east to change and all of thins things.
- Or rather than making the all previous things we can use the lambda expression and if we use it inside the sorting it will be something like that **songList.sort((one, two) -> one.getTitle() .compareTo(two.getTitle()));**

- There is interfaces have only one method to implement like interfaces of comparator we only have to implement SINGLE ABSTRACT METHOD which is called SAM and this special types of interfaces which have only one method to implement we can implement it but with using the lambda expression rather than creating a complete interface for it

- So we have this 3 important java collections list which when sequence and indexing matters and the second thing is the set and it is when the unique items matters cause it checks that there is no 2 items the same and the third

thing is by using the map and it is important when finding something by key value pairs

- There is difference between the reference equality and the object equality the reference equality is meaning that 2 reference variables but both points to the same object in the heap so if we compare them by using o1==o2 it always will get true this is the first thing the second thing is related to the object quality and it means that 2 references and 2 objects in the heap but the objects are considered as a meaning both equivalent and this one need to be checked but by using the equal overridden method cause if we use the == like the previous one it always will return false cause both are different reference variables
- ***If two objects have the same hash code value, they are NOT required to be equal. But if they're equal, they MUST have the same hash code value.***
- ***So, if you override equals(), you MUST override hashCode() and both with using the same thing if one will use title the second one must use title too.***
- ***The default behavior of hashCode() is to generate a unique integer for each object on the heap. So if you don't override hashCode() in a class, no two objects of that type can EVER be considered equal.***
- ***There is another data structure we can use rather than using the hash set and it is the treeset and it is the same as hash set in preventing the duplicates but it has extra benefit which is making all the tree is sorted always any time you will access it you will find it sorted and you will not need to resorting them again and if you want it to its sorting logic normally being another different one you can easily pass the lambda expression to the constructor when you define the tree set***
- ***Each element in the map is actually 2 objects a key object and the value object and we can have duplicated values but we can not have duplicated keys and it is important that the map is containing 2 objects cause that means if you want to define a map with string and integer value you can not use the int primitive type cause it is not an abject you will need to use the Integer class that we mentioned it before***
- ***return Collections.unmodifiableList (songs) ; this is to return the collection we use but preventing that no one else will change it you can no add to them or even making sort to it***
- ***If you declare a method to take List<Animal>, it can take ONLY a List<Animal>, not List<Dog> or List<Cat>. Is there is solution for this problem ? yes and that's by using the WILDCARD like this definition***

- ***public void takeAnimals(List<? extends Animal> animals) and here we will talk about the benefits of the wildcard and they are :***
  - ***when we use the wildcard it makes the compiler stops you from doing anything that could hurt the list referenced by the method parameter so it will prevent you from adding anything***
  - ***so you can not add elements to the list (you can not adding new things into the list)***