Lab 14: Collections

Objective

In this lab, we will take advantage of the **Collection** interface in order to add more flexibility to some of our **Inventory** system classes.

Overview

In this lab you will:

- Change some arrays to collection types
- Add objects to collections
- Retrieve objects from collections

Step by Step Instructions

Exercise 1: Using Collections

The Inventory system contains several arrays. We will convert some of these to collections. In the interest of time, we will leave the others as arrays. We will make sure that the different collections specify their generic type to help ensure runtime type safety.

1. This lab will enhance the inventory application in the **Store** project. The **Artist** class should contain two arrays at this point -- `memberNames` and `memberInstruments`. Convert `memberNames` to a **SortedSet**. Don't create any get or set methods for it yet. We'll handle that a little later. Make sure to specify that this **SortedSet** is of generic type String.

2. Convert `memberInstruments` to a **Map**. We want a **Map** so that we can key the instruments to the names of the members. Each entry in the **Map** will contain a key which is the member name, and a value which is a **SortedSet** holding a list of the instruments the member plays.

   Specifying the generic type for the **Map** is a little tricky. We have to tell the compiler what types the **Map** will use for its key and value. The key is easy – it is a **String**. The value, however, is a **SortedSet**, another generic type. So as part of the generic declaration of the **Map**, we need to provide a generic declaration of one of its generic parts. We are declaring that the **Map** will contain a key value pair in which the key is a **String** and the value is a **SortedSet** that contains **String** objects

   ```
   private Map<String, SortedSet<String>> memberInstruments;
   ```

3. Add a default constructor which initializes memberNames to a **TreeSet** containing only String references; It should in similar fashion initialize memberInstruments to a **TreeMap**.

   Add a constructor using the Source->Generate Constructor Using fields wizard to include all fields.

   Modify the original Artist(String name) constructor so that it also initializes the two collectors, e.g.,

```
public Artist(String name) {
    this();   // Use the default constructor to initialize the collections
    setName(name);
}
```

4. Create a new method called `addMember()` which accepts a **String** for the member name and an **ArrayList** of instruments. Be sure to specify the generic type of instruments.

   Add `name` to the `memberNames` **ArrayList**. Add an entry to the `memberInstruments` **Hashtable** as well. Your code should look as follows:

```
public void addMember(String name, SortedSet<String> instruments) {
    memberNames.add(name);
    memberInstruments.put(name, instruments);
}
```

5. Right about now, it has probably occurred to you that `memberNames` is redundant and therefore unnecessary. You're right, and this will happen at times in programming. As you get deeper into the details, you will re-factor your design and your code in order to add missing details or increase efficiency. For our system, we will just leave it as is since we want to experiment with two different types of containers.

6. Now let's exercise the **Artist** class to see if our **Collections** are working properly. We will add two members with a list of instruments for each and then extract one of them to print out the name and instruments. Create a class named **ArtistExerciser** that has a `main` method. Create an instance of **Artist** named `hotPlate`.

7. In the `main()` method, create a **TreeSet** called `instruments1` with a generic type of **String** and add these instruments to it: "Piano", "Clarinet", "Hurdy Gurdy" and "Tuba". [Hint: don't forget to import `java.util.*;`].

8. Call `addMember`(), passing it the name "**Tom**" and the **TreeSet** `instrument1`.

9. Repeat steps 7 and 8, creating another **TreeSet** of generic type **String** and adding the instruments of your choice. Add these via `addMember`(), using the name "**Steve**".

10. Now, how will we extract Tom's instruments? First, we need to get a list of band member names. Write a method in the **Artist** class named `getMembers()` that returns the **SortedSet** that contains all of the member names. Remember that the **SortedSet** contains **Strings**, so **String** should be used as the generic in the declaration of the method. This should look as follows:

```
/**
 * @return the memberNames
 */
public SortedSet<String> getMembers() {
    return memberNames;
}
```

11. Write a method in the **Artist** class named `getInstruments()` that accepts a member name as a **String** and returns the list of instruments as an **SortedSet**. Examine the API for the **Map** class to discover the correct method to access the **Map** and return a value based on a key.

12. Go back to the **ArtistExerciser** class and print out information for the `HotPlate` **Artist**.. create a private static void method call printMemberInstruments. to the class. It should accept an Artist and a String for memberName. Print out the member name .Then using an enhanced for loop, print out the instruments that member plays. Example below:

```
private static void printMemberInstruments(Artist artist, String memberName) {
    out.println("HotPlate band member " + memberName + " plays: ");
    for (String instrument : artist.getInstruments(memberName)) {
        out.println('\t' + instrument);
    }
}
```