Exercise 9: Common Interface Types

Lab Objectives

- Use existing common Interface types:
  - **Iterable** - so objects of your aggregate collection class can be used with a for-each loop or Iterator
  - **Comparable** - so objects of your class can be compared and sorted into order

Associated Reading:

Horstmann,

Chapter 8, Sections 8.1 - 8.4: Interfaces and Polymorphism

Implementing existing Interface types

Interfaces provide a powerful way of deploying abstraction and forming a contract with a developer such that any classes implementing that interface can be guaranteed to support common behaviour, without having to share any implementation.

An Interface is very simple; it contains method declarations, without any implementation, i.e. method headers. It may also include constants, but you should not worry about this. During this lab exercise you will not be defining your own interfaces, but instead making use of two existing interface types that exist within the Java API: *Iterable* and *Comparable*.

When you wish to implement an interface, you firstly place the implements reserved word and the interface type you wish to implement in the class header. You then need to provide the implementation of the method(s) declared in the interface, i.e. the method body. Effectively, you are overriding these methods although it is enforced, unlike overriding superclass methods with inheritance.

In fact you may be wondering what the difference is between interfaces and inheritance - there are many and they should not be confused. When classes are related by inheritance, i.e. they all extend the same superclass, they share implementations with that superclass and would therefore be expected to be closely related in all of their underlying logic. When classes implement an interface, they share no implementation and need not know anything about each other - they only need be related by simply implementing the method(s) defined in that interface forming a common contract between them.

If several different classes that appear completely unrelated all implement the same interface, they can be processed in a common way, as we can be certain that they will each have their own implementation of the method(s) declared in that interface. Therefore the technique of polymorphism also applies with interfaces.

Question 9.1 To get started with this lab exercise, inside your "EclipseWorkspaces" folder create a new folder called "**LabExercise9**", then open up Eclipse, selecting this as your workspace. Following this, create a new Java Project called "**iterable**" and then ensure the classes "**PlayList**", "**Song**" and "**PlayListTest**" are in it - you may download them from blackboard.

a) In the main method of **PlayListTest**, attempt to write a for-each loop that iterates through each Song object within a PlayList instance, e.g.

```
for (Song track : playlist) {
        track.play();
}
```

b) You should receive an error that states: "*Can only iterate over an array or an instance of java.lang.Iterable*". In other words, the for-each loop only works on arrays or classes that implement the Iterable interface. Go to the Java API for the Iterable interface: http://docs.oracle.com/javase/7/docs/api/java/lang/Iterable.html and study its description and the method it defines. You should also notice that ArrayList is one of the "**All Known Implementing Classes**". This is why you have been able to use a for-each loop to iterate through elements within an ArrayList instance.

c) If you want to be able to use a for-each loop for your own collection classes, then you need to ensure they implement the Iterable interface. In your **PlayList** class firstly change the class header to be:

```
public class PlayList implements Iterable<Song> {
```

d) Then add the following method, noting how the method header matches that specified within the API, with the type parameter *T* replaced with Song in this case:

```
public Iterator<Song> iterator() {
      return songlist.iterator();

}
```

e) Conveniently, as ArrayList also specifies an iterator() method you can simply delegate to that. You will also need to import: `import java.util.Iterator;`.

f) Your for-each loop in the PlayListTest program from part a) should now compile and work as expected. Test it to ensure that it does.

g) You should also be aware that you can use the Iterator that is returned by your iterator() method to process each of the elements in your collection. Go to the Java API for the Iterator interface: http://docs.oracle.com/javase/7/docs/api/java/util/Iterator.html and study its description and the 3 methods it defines.

h) At the bottom of the main method in your **PlayListTest** program add the following code:

```
Iterator<Song> itr = playlist.iterator();
while (itr.hasNext()) {
      itr.next().play();

}
```

i) This is effectively identical to the for-each loop and the compiler will actually convert your for-each loop to something very similar to this. The for-each loop looks more elegant and slightly less complex and is therefore often chosen over creating an Iterator. There is however one benefit of using the Iterator, it allows you to modify the collection whilst looping; specifically you can remove elements once they have been returned. Modify your loop to use the remove() method and then check the size of the list:

```
while (itr.hasNext()) {
      itr.next().play();
      itr.remove();
}
System.out.println("Size: " + playlist.numberOfSongs());
```

j) Another benefit of the Iterator approach is that you could use the hasNext() method within the loop to find the last element and do something differently in this circumstance - this is not possible with a for-each loop. Generally speaking, for-each is much more commonly used but it's worth being aware that Iterator exists as an alternative.

Question 9.2 Create a new Java Project called "**myiterable**" and then using one of your own Aggregation classes do the following:

a) Ensure the class **MultipleDice** (from Exercise 6) and that which it aggregates (i.e. **Die**) is in your project, as well as the corresponding test program **MultipleDiceTest**.

b) Firstly, ensure your aggregation class implements Iterable and defines the iterator() method.

c) In your test program, add to the existing code by creating a new instance of MultipleDice, then using a for-each loop, iterate through each element, calling the toString() method.

d) (*optional challenge*) Do the same but using an Iterator instance (from the java.util package) and in addition you should remove each element after it has been processed.

e) (*optional challenge*) Think how using the hasNext() method internally within the while loop you can print a message "Last element" accordingly and attempt to do this.

Question 9.3 Look up the documentation for the **java.lang.Comparable** interface in the Java API and then attempt to answer the following tutorial questions:

a) If **x.compareTo(y) > 0** is true, what would you expect **y.compareTo(x)** to be?

b) If **x.compareTo(y) == 0** is true, what would you expect **x.equals(y)** to be?

c) What is the expected value of **x.compareTo(x)**?

d) If **x.compareTo(y) < 0** is true, and if **y.compareTo(z) < 0** is true, what can you say about **x.compareTo(z)**?

Question 9.4 The String class implements the Comparable interface. Create a new Java Project called "**comparestrings**" and within it create a new Java class called "**CompareTest**".

a) Go and look at the method description for the String class in the API: http://docs.oracle.com/javase/7/docs/api/java/lang/String.html you will see that the return result is either negative, zero or positive.

b) Copy the following code within the main method of your CompareTest program:

```
String s1 = "Jim";
String s2 = "Fred";
if (s1.compareTo(s2) < 0 ) { // then s1 is before s2
```

```
            System.out.println(s1 + " is before " + s2);
        }
        else if (s1.compareTo(s2) == 0 ) { // then s1 and s2 same
            System.out.println(s1 + " is the same as " + s2);
        }
        else if (s1.compareTo(s2) > 0 ) { // then s1 is after s2
            System.out.println(s1 + " is after " + s2);

        }
```

c) Experiment with different values for s1 and s2, including lower and upper case characters - can you see how strings are compared based on their ascii value?

Question 9.5 Create a new Java Project called "**comparable**" then copy the classes "**PlayList**", "**Song**" and "**PlayListTest**" into it:

a) In the PlayList class, add a new method, that attempts to sort the list into an order:

```
public void sortPlaylist() {
    Collections.sort(songlist);
}
```

b) You should receive an error - you may not understand what it is complaining about, but in simple terms the sort(...) method belonging to the Collections class requires the class type that is stored within the collection (passed as a parameter) to implement the Comparable interface. In this case, *songlist* represents an ArrayList<Song> and therefore the Song class needs to implement the Comparable interface so that the sort(...) method knows how to order Song objects.

c) Go to the **Song** class and firstly change the class header to the following:

```
public class Song implements Comparable<Song> {
```

d) You then need to define the compareTo(...) method, which this interface requires. There is no set rule on exactly how you want to order objects of your own custom classes - it's up to you. Let's firstly focus on the song title; add the following method to your **Song** class:

```
public int compareTo(Song other) {
    return this.title.compareTo(other.title);

}
```

e) As the *title* field is of type String and the String class defines a compareTo(...) method itself, you can use that. Now go back to your PlayList class, you should notice that the sortPlaylist() method you wrote in part a) is now not showing any errors. Go to the **PlayListTest** program and add the following code prior to where any elements may be removed:

```
playlist.sortPlaylist();
System.out.println(playlist.getTrackListings());
```

f) If you run the program, you should now see that the tracks have been ordered based upon their title. What if you had two songs with the same title, but a different artist and duration? Manually edit one of the Song objects added to the PlayList towards the top of the test program to ensure it has the same title as another song, but a different duration and artist.

g) Let's now ensure the compareTo(...) method in the **Song** class compares title, followed by

artist, followed by duration:

```
public int compareTo(Song other) {
      int result = this.title.compareTo(other.title);

      if (result == 0) {
            result = this.artist.compareTo(other.artist);

            if (result == 0) {

                  result = this.duration - other.duration;
            }
      }

      return result;
}
```

h) This updated version will firstly compare titles, however, if this results in 0, i.e. they have the exact same title, it compares artists and upon artists being identical it compares duration. **Note**: Because duration is of type `int` and we know duration will always be a zero or positive number, the simplest way to compare is via a subtraction that will result in a negative, positive or zero based result.

i) Go back to your **PlayListTest** program and ensure your compareTo(...) method works correctly by sorting songs with different details and some similarities.

j) Create two Song objects and then use the compareTo(...) method to see whether a positive, negative or zero is returned.

Notes - Ensuring compareTo(...) and equals(...) are consistent:

- In the API for the Comparable interface, it states: "*It is strongly recommended (though not required) that natural orderings be consistent with equals*."
- You may look at the information within the API for a more detailed explanation, but in simple terms if x.compareTo(y) returns 0, then x.equals(y) should return true.
- You should therefore try to ensure your implementation of these two overridden methods uses the same logic.
- A typical example of a mistake that could be made is when one of your fields is a String. The String class defines the methods **equals** and **equalsIgnoreCase**, as well as **compareTo** and **compareToIgnoreCase**. If you use equalsIgnoreCase in your overridden equals method then you should use the equivalent compareToIgnoreCase in your overridden compareTo method to ensure they are consistent.

Question 9.6 [✸Portfolio✸] Create a new Java Project called "**mycomparable**" and then using one of your own Aggregation classes do the following:

a) Ensure the class **Register** (from Exercise 6) and that which it aggregates (i.e. **Name**) is in your project, as well as the corresponding test program **RegisterTest**.

b) Firstly, ensure your Name class implements Comparable.

c) Write a suitable compareTo(...) method within this class. You should compare *familyName* and in the event of them being the same then compare *firstName*.

d) In your collection class, Register, write a sort method, which delegates to the sort method defined in the Collections class within the java.util package.

e) In the main method of your test program, add to the existing code, ensuring the sort method (and compareTo that it uses) works correctly by having a sufficient range of objects added to your Register instance and make sure the items are printed before and after a sort.

f) As this is a portfolio question, you should fully document the Register class with Javadoc, i.e. class header, constructor/method description and tags where appropriate.