Exercise 7: Interfaces

Lab Objectives

- Create your own custom Interface types
- Use existing common Interface types:
  - **Iterable** - so objects of your aggregate collection class can be used with a for-each loop, forEach method 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

Interfaces

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

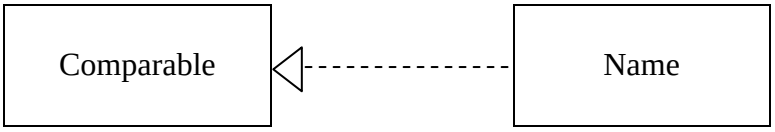
An Interface is very simple; it usually solely contains method declarations, without any implementation, i.e. method headers. As of Java 8 interfaces can contain static and default methods - you will see examples of this in the *Comparator* interface in a later lab exercise.

During the first part of this lab exercise you will explore how to define your own interface types and make use of them to create reusable capabilities. You will then go on to make use of interfaces provided by the Java API that abstract iteration over collections and comparison of objects. Namely: *Iterable*, *Iterator* and *Comparable*.

You define an interface in a similar way to a class, but use the `interface` keyword instead of class. When you wish to implement an interface, you 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.

Notes - The Realization association:

- *Realization* (Implementing an interface) **"can-do"**
- Illustrated in UML class diagrams using the following association symbol:



Realization
("can-do" or "implements").
A name "**can be"** compared.

In these exercises you will also see an illustration of *polymorphism*. If several different classes 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.

Although an interface cannot be instantiated directly it can be used as a type in a variable declaration. Used in this way, it means that the variable be instantiated as an object of any class that implements that interface.

As you work through the exercises, make mental notes whether a type is an interface type or a class type, and identify the (class) types of the objects that invoke the interface methods.

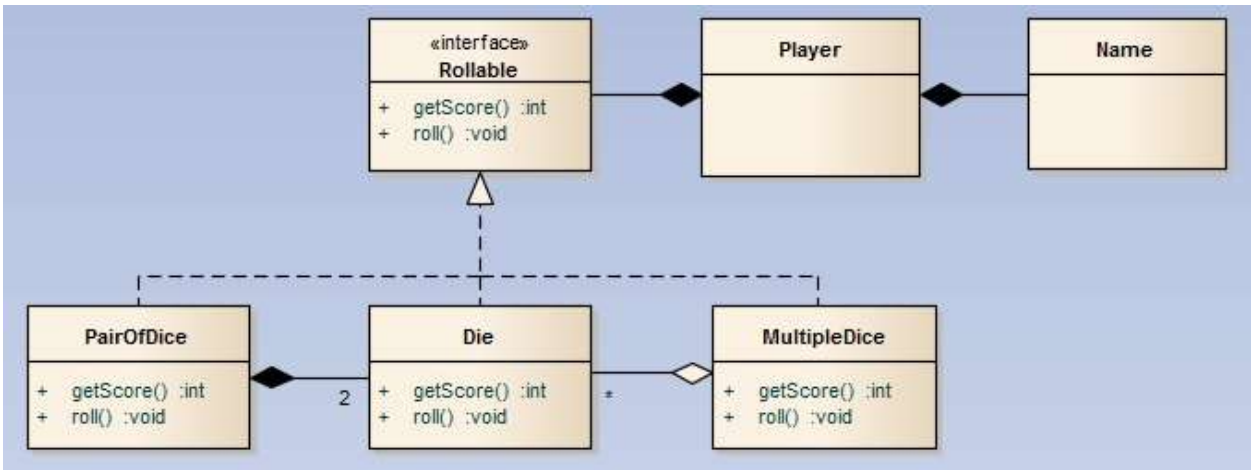
Question 7.1 Reopen your Eclipse workspace called "**CTEC2905_OO_Design**" then download the existing Java Project that can be extracted from the "**Interfaces.zip**" file, and imported into Eclipse. Recall the previous classes you have worked with related to playing dice: **Die**, **PairOfDice** and **MultipleDice**. You may have noticed that each of these implementations share two common methods: **getScore()** and **roll()**. We have abstracted this common functionality into an interface **Rollable**, forming a simple API for all types of dice implementations to adhere to.

In the main package, take a look at the **PolymorphismDemo** program, and how there is a list of type Rollable populated with a variety of different rollable types. The for-each loop processes each of these in a uniform way, but will use dynamic method lookup to invoke the correct roll and getScore method depending on the specific class type of each object. This is an example of polymorphism.

It's also possible to invoke toString in this way, as every class automatically inherits toString, so no matter the type, the compiler can be satisfied the object will have a toString implementation.

Question 7.2 [✸Portfolio A.2✸] Download the existing Java Project that can be extracted from the "**PortfolioPlayer.zip**" file, and imported into Eclipse. This question forms the second part of Portfolio case study A, and you will need to have already attempted Portfolio A.1 from the previous exercise. Copy your **Player** class from the previous exercise into the src > lib package of the PortfolioPlayer project.

Study the UML diagram below - you should see the Rollable hierarchy and a familiar Player class, that you started working on in the previous exercise.



In the test > lib package you have been given a JUnit Test Case **PlayerTest** containing unit tests, which will be used to mark your implementation of the Player class. You can use it to gain progressive feedback on your design, however, some tests will not pass at this stage and are designed to test functionality that you will add later in this exercise. It is important that you use the same method signatures and return types as those specified in the test program, e.g. note the method called "setFullPlayerName" invokved in the testSetFullPlayerName unit test meaning this exact name needs to be used for the corresponding method in your Player class.

a) Update the Player class so that rather than holding a field of type PairOfDice, it instead holds a field of type **Rollable**. You will then need to update the two argument constructor so it accepts any Rollable type, however the other constructors can assign a default object of your choice to this variable, so long that it is rollable.

b) The **getPairOfDice** method should be updated to **getRollable**, with a return type of Rollable.

c) You should run the unit tests in **PlayerTest** and check the progress you have made in this question as well as the general design of Player you started in the previous exercise.

d) In the src > main package, you will notice a class called **PlayerApp** that has a method called execute, which accepts an ArrayList<Player> and a String object and returns a String. Currently, it simply returns an empty string. It should however do the following:

- Get the player at index 0 of the ArrayList<Player> `players` and set the full player name with the String `fullName` (passed as a parameter to the execute method).
- Return a String that contains the name of each player in the ArrayList who has a full name containing the letter "a", in the format: "firstname, SURNAME", e.g. "joe, BLOGGS", each followed by a new line.

In the test > main package there is a further JUnit Test Case **PlayerAppTest** containing a unit test called testExecute, which assesses whether the execute method behaves correctly.

Please note: Your solution to part d) will be marked by using a different input data set of players within the list to ensure your solution works dynamically based upon any given data set and is not hardcoded in any way.

e) As this is a portfolio question, you should update the Javadoc you have previously written for the Player class where necessary to ensure it accurately reflects the updated class.

Please turn over...

Interfaces for iteration and comparison

When we create a class that encapsulates a collection of objects we would normally expect to have the ability (note: 'can-do' behaviour) to access each element in turn (i.e. iteration). For certain types of object it might also make sense to sort them in a particular order. Sorting objects requires the ability to compare them in pairs to establish which one is logically before the other (i.e. comparison). These behaviours are so common in programming, that the Java API provides common interfaces to support them. Java programmers should make use of these interfaces so that their own classes conform to standard practices. They exist in the following packages:

java.lang: *Comparable, Iterable*

java.util: *Iterator*

Question 7.3 Revisit the **Interfaces** project. For this question you will work on the PlayList and Song classes inside the lib.iterable_comparable package and the PlayListApp program in the main package.

a) Let's study the Iterable interface: in the main method of **PlayListApp**, 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. Look at the Java API for the Iterable interface: http://docs.oracle.com/javase/8/docs/api/java/lang/Iterable.html and study its description and the iterator and forEach methods 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 before.

c) Additionally, replace the body of the PlayList's **mergePlaylist(...)** method as follows:

```
p.forEach(s -> songlist.add(s));
```

Currently, the forEach method is not available to your PlayList type.

d) If you want to be able to use a for-each loop or forEach method 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> {
```

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

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

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

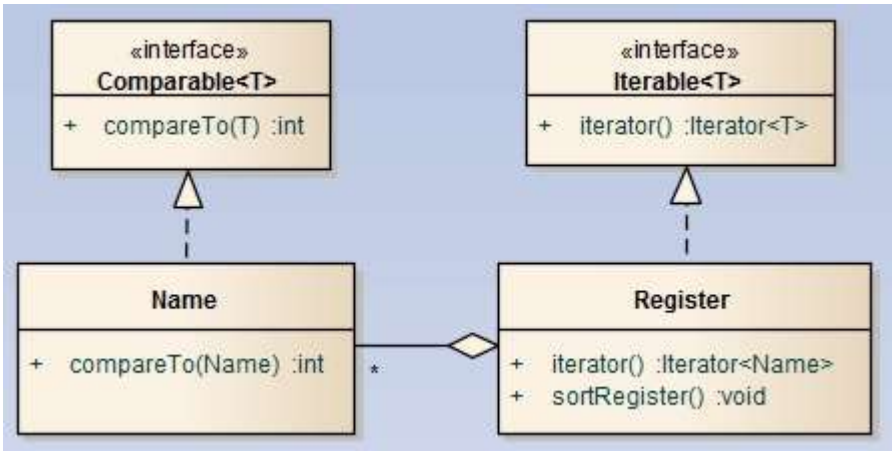
g) Your for-each loop in the PlayListApp program from part a) and the forEach method from part c) should now compile and work as expected. Test them to ensure they do.

Question 7.4 Continue working in the same project, and look inside the lib.polymorphism package at the **MultipleDice** class.

a) Firstly, ensure the aggregation class MultipleDice implements Iterable and defines the iterator() method (as shown in the previous question for the PlayList class). **Note**: Your MultipleDice class already implements the Rollable interface, however, you can legitimately implement multiple interfaces, simply separating them with a comma, e.g. `implements Rollable, Iterable<Die>`.

b) In the main package, create a demo program called IterableDemo, and within the main method create a new instance of MultipleDice, then using a for-each loop, iterate through each element, printing the output of the toString method. Then try the same below this but by using the forEach method.

Question 7.5 [✸Portfolio B.2✸] This question forms the second part of Portfolio case study B, and you will need to have already attempted Portfolio B.1 from the previous exercise. Revisit the **PortfolioRegister** project, and study the UML diagram below.



a) Update the **Register** class so that it implements the Iterable interface and defines an appropriate iterator method.

b) Run the **RegisterTest** Test Case in the test > lib package to ascertain if the **testIterator()** unit test now passes

c) As this is a portfolio question, you should update the Javadoc in the Register class for the newly added method.

Please turn over...

Notes - Using the Iterator:

- 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/8/docs/api/java/util/Iterator.html and study its description and the two non-default methods *next* and *hasNext*.
- When you use a for-each loop the following is generated behind the scenes:

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

Question 7.6 Look up the documentation for the **java.lang.Comparable** interface in the Java API and then attempt 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 7.7 The String class implements the Comparable interface. In the main package of your current project create a new Java class called **CompareStringDemo**.

a) Look at the **compareTo** method description for the String class in the Java API: http://docs.oracle.com/javase/8/docs/api/java/lang/String.html you will see that the return result is either negative, zero or positive.

b) Copy the following code into the main method of your CompareStringDemo 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 the Unicode value of each character in the strings?

Question 7.8 Continue working in the same project and revisit the lib.iterable_comparable package containing the classes **PlayList**, **Song** and the main package that contains **PlayListApp**:

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 delegate to that. Now go back to your PlayList class, you should notice that the sortPlaylist() method you wrote in part a) is not showing errors anymore. Go to the **PlayListApp** program and add the following code:

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

f) If you run the program, you should now see that the tracks have been ordered on 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 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 = Integer.compare(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 the Integer.compare(int, int) utility method

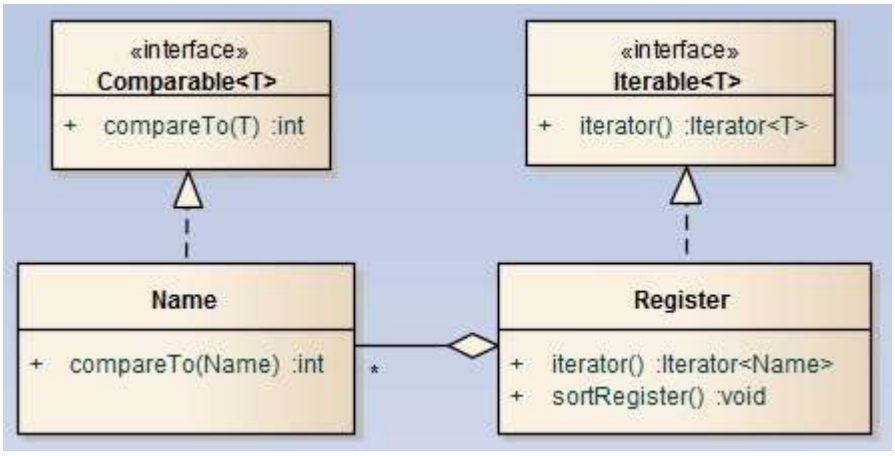
i) Go back to your **PlayListApp** program and ensure your compareTo(...) method works correctly by sorting songs with different details and some similarities.

j) Continuing in the same program, create two Song objects and then use the compareTo(...) method to see whether a positive, negative or zero integer 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 7.9 [✸Portfolio B.3✸] This question forms the third part of Portfolio case study B, and you will need to have already attempted Portfolio B.1 from the previous exercise. Revisit the **PortfolioRegister** project, and study the UML diagram below:



If you look in the Name class, you will see it implements Comparable and has a compareTo(...) method that compares two names by their *familyName*, and in the event of these being the same, then by their *firstName*.

a) In your **Register** class add a sort method **sortRegister()**, that sorts the internal collection into its natural order, i.e. that prescribed by the compareTo method of its elements.

b) Run the **RegisterTest** Test Case in the test > lib package to ascertain if the **testSort()** unit test now passes.

c) As this is a portfolio question, you should update the Javadoc in the Register class for the newly added method.

Question 7.10 [✸Portfolio A.3✸] This question forms the third part of Portfolio question A, and you will need to have already attempted Portfolio A.2 from earlier in this exercise. Revisit the **PortfolioPlayer** project and do the following:

a) Update your **Player** class so that it implements the Comparable interface and has a compareTo(...) method that simply compares two players by their name. It should do so by delegating to the compareTo method already defined in the Name class.

b) Run the **PlayerTest** Test Case in the test > lib package to ascertain if the **testCompareTo()** unit test now passes.

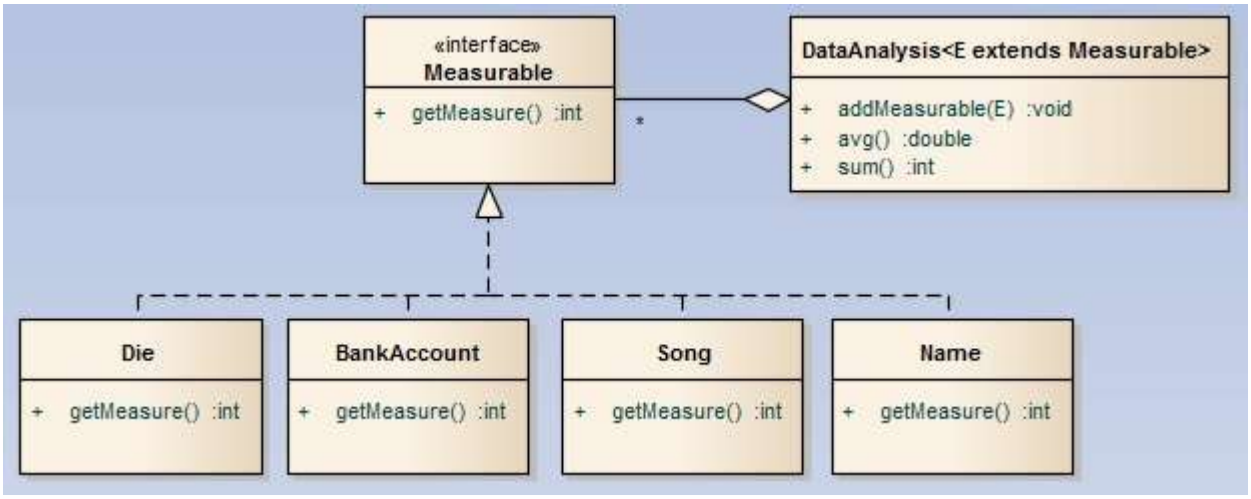
c) As this is a portfolio question, you should update the Javadoc in the Player class for the newly added method.

Please turn over…

A custom Interface for measuring

We may wish to create a custom interface type that defines a behavior allowing objects to be 'measured'. Any class implementing the Measurable interface must provide a numeric measurement of its choice, e.g. a Die could be measured by its score, a Song by its duration, and a Bank Account by its balance. This common way of measuring different objects may be useful when carrying out varying forms of statistical analysis.

Question 7.11 (*optional challenge*) Revisit the Interfaces project and look inside the lib.measurable package. You will notice a DataAnalysis class and the Die, Song and Name classes you have previously seen.



a) Create an interface **Measurable** that defines a single method: `int getMeasure();`

b) Ensure each of the classes Die, Song and Name implements the Measurable interface and write their own respective versions of the getMeasure() method. The measurements they should provide are score, duration, and length (of the full name) respectively.

c) Study the **DataAnalysis** class - this is a generic class that can be instantiated with any type `E`, so long as that type implements the Measurable interface. This is achieved with the bounded type parameter `<E extends Measurable>` adding such a constraint. The class then holds a list of type E, and therefore aggregates a Measurable type, with the ability to `add` elements.

d) You will see a `sum()` method that calculates and returns the cumulative sum of all measurable elements within the list. Attempt to write the `avg()` method, which should return the average of all measurable elements, as a double. If the list is empty you could return -1. **Note**: you can achieve this using a for-each loop or using a Stream pipeline.

e) Write a client program **MeasurableDemo** and within the main method check the sum and avg methods work by creating different instances of DataAnalysis, each populated with either dice, songs or names, e.g. `DataAnalysis<Song> songAnalysis` (**Note**: you will need to import these classes from the lib.measurable package.)

f) Can you add `min()` and `max()` methods to the DataAnalysis class that return the numeric value of the minimum or maximum measurable element in the list? (if the list is empty you could return -1).