420-200-VA Lecture + Assignment 3B Due by April 7, 2020

Comparing PetDog Objects for Order

1 Purpose

The purpose of this assignment is to learn about and practice Java interfaces in the common context of sorting a collection of objects; namely, an array list of PetDog objects from A3-A. We choose the Comparable and Comparator interfaces as examples in our practice, as they provide two most common approaches to sorting array lists using Java's Collections.sort() method.

2 Introduction to Comparable and Comparator Interfaces

Suppose that p1 and p2 are two PetDog object references, and that we want to sort them. To do that, we obviously need a way to decide whether p1 "is less than" p2, p1 "is equal to" p2, or whether p1 "is greater than" p2.

That's seems like a trivial task. We already know how to override the equals method, and have no problem writing a method named less that implements the concept of p1 "is less than" p2, and a method named greater that implements the concept of p1 "is greater than" p2.

Unfortunately, writing a less or greater method would not help here because Collections.sort() would not recognize them, regardless of their names. In fact, depending on how you call it, Collections.sort() has its own rules that it imposes on the objects to be sorted.

For example, using Collections.sort(), here are the two most common ways to sort dogList, our array list of PetDog objects:

way 1) Choose this way when you want to sort dogList in only one way.

```
Collections.sort(dogList);
```

This overload of Collections.sort() requires that class PetDog whose objects are to be sorted implement the Comparable interface. This is the easier approach of the two and works well in most cases.

way 2) Choose this way when you want to sort dogList in multiple ways. For example, to sort dogList by name first and later by age you write:

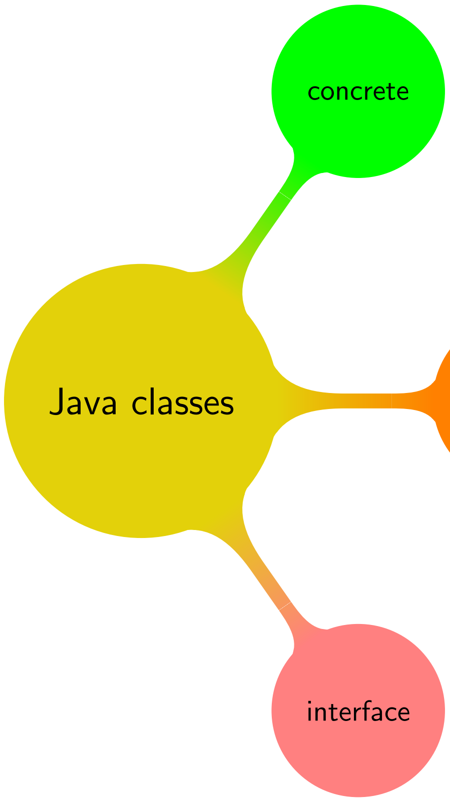
```
Collections.sort(dogList, new SortByName()); // sort by name
// process
Collections.sort(dogList, new SortByAge()); // sort by age
// process
```

This overload of Collections.sort() requires that SortByName and SortByAge be classes that implement the Comparator interface.

3 Java Interfaces

An interface specifies what a class must do but not how.

To understand Java Interfaces, it helps to understand how Java classes are categorized.



A concrete class implements *all* of its methods. You *can* always create an object of a concrete class. The classes that you have written in all of your programming assignments so far have been concrete classes.

An `abstract` class implements *some* of its methods, declaring the rest as `abstract` methods. You *cannot* create an object of an abstract class. To use an abstract class, you write a class that `extends` it.

An interface implements *none* of the methods it specifies (except for default and static methods). So special in Java that you must use the `interface` keyword (not `class`) to declare one. You *cannot* create an object of an interface. To use an interface, you write a class that `implements` it.

For example, the declaration

```
public class PetDog implements Comparable<PetDog> { ... }
```

implies that class `PetDog` must implement every instance method specified in the `Comparable` interface which is shown below; otherwise, `PetDog` itself becomes an `abstract` class.

```
// all interfaces are implicitly public
// all fields and methods in an interface are also implicirtly public
// thus, the use of the 'public' keywrod is redundant
interface Comparable<T>  // T is the type of the objects being compared, like PetDog
{
    int compareTo(T o); // Compares the calling object with the specified object o for order
                        // notice that the method has NO {body}
}
```

Again, an interface specifies what a class must do but not how; that's why methods specified in an interface do not have method bodies.[1]

In other words, it makes no difference to the interface how you compare two `PetDog` objects, as long as your `compareTo()` override in `PetDog` returns an integer.

`Collections.sort()`, in turn, interprets the integer returned by the call `p1.compareTo(p2)` as follows:

if `p1.compareTo(p2)` returns	then `Collections.sort()` concludes that
a negative integer	`p1` is less than `p2`
zero	`p1` is equal to `p2`
a positive integer	`p1` is greater than `p2`

[1]Starting with Java 8, an interface may contain default and static methods with implementation (method bodies).

4 Task 1 of 3: Sorting dogList by Age

step 1) Adjust your PetDog class header as follows:

```java
public class PetDog implements Comparable<PetDog>
```

step 2) Implement the compareTo() method in your PetDog class as shown on lines 3-12

```java
 1 public class PetDog implements Comparable<PetDog>
 2 {
 3    @Override
 4    public int compareTo(PetDog other)
 5    {
 6       if( this.age < other.age )
 7          return -1; // or any other negative integer
 8       else if( this.age > other.age )
 9          return +1; // or any other positive integer
10       else
11          return 0; // because this.age == other.age
12    }
13       // the remaining members not shown for brevity
14       // ...
15 }
```

step 3) Define method main in your driver program as follows:

```java
   public static void main(String[] args) throws FileNotFoundException
   {
      String inFileName = "./dog_infile.txt"; // must exist in the project folder
      ArrayList<PetDog> dogList = TextFileIO.readTextFileToArrayList(inFileName);

      Collections.sort(dogList); // sort using our compareTo() override

      String outFileName = "./dog_outfile_sorted_by_age.txt";
      TextFileIO.writeArrayListToTextFile(dogList, outFileName);

      return;
   }
```

step 4) Run your program

step 5) Check that the contents of your dog_outfile_sorted_by_age.txt file contains these lines:

```
I'm Toby, a 1 year old male Alaskan Malamute
I'm Daisy, a 1 year old female Pug
I'm Max, a 5 year old male Poodle
I'm Teddy, a 6 year old male Maltese
```

```
I'm Teddy, a 6 year old male Cavalier King Charles Spaniel
I'm Archie, a 7 year old male Poodle (Miniature)
I'm Luna, a 7 year old female Poodle
I'm Bear, a 8 year old male Dachshund
I'm Duke, a 8 year old male French Bulldog
I'm Frankie, a 9 year old male Staffordshire Bull Terrier
I'm Max, a 10 year old male Great Dane
I'm Lucy, a 12 year old female Labrador Retriever
I'm Milo, a 12 year old male German Shepherd
I'm Bella, a 12 year old female Yorkshire Terrier
I'm Frankie, a 12 year old male Jack Russell Terrier
```

step 6) End of Task 1

5 Task 2 of 3: Sorting dogList by Name and Age

Suppose we want to sort our dogList list by name; if there are duplicate names, then we sort them according to age.

step 1) Ensure that your PetDog class header looks like this:

```java
public class PetDog implements Comparable<PetDog>
```

step 2) Modify your existing compareTo() method in your PetDog class as shown on lines 3-22:

```java
 1 class PetDog implements Comparable<PetDog>
 2 {
 3    @Override
 4    public int compareTo(PetDog other)
 5    {
 6       // first compare the names
 7        // lucky for us, the String class implements Comparable
 8        // that is, it provides its own compareTo() override
 9        // so we pass the buck to that override
10       int result = this.name.compareTo(other.name);
11
12       // if the names are not the same return result
13       if( result != 0) return result;
14
15       // when the names are the same sort according to age
16       if( this.age < other.age )
17          return -1; // or any other negative integer
18       else if( this.age > other.age )
19          return +1; // or any other positive integer
20       else
21          return 0; // because this.age == other.age
22    }
23       // the remaining members not shown for brevity
24       // ...
25 }
```

step 3) Adjust the name of the output file in main:

```java
   public static void main(String[] args) throws FileNotFoundException
   {
      String inFileName = "./dog_infile.txt"; // must exist in the project folder
      ArrayList<PetDog> dogList = TextFileIO.readTextFileToArrayList(inFileName);

      Collections.sort(dogList); // sort using our compareTo() override

      String outFileName = "./dog_outfile_sorted_by_name_and_age.txt";
      TextFileIO.writeArrayListToTextFile(dogList, outFileName);

```

```
        return;
    }
```

step 4) Run your program

step 5) Check that the contents of your `dog_outfile_sorted_by_name_and_age.txt` file contains these lines:

```
I'm Archie, a 7 year old male Poodle (Miniature)
I'm Bear, a 8 year old male Dachshund
I'm Bella, a 12 year old female Yorkshire Terrier
I'm Daisy, a 1 year old female Pug
I'm Duke, a 8 year old male French Bulldog
I'm Frankie, a 9 year old male Staffordshire Bull Terrier
I'm Frankie, a 12 year old male Jack Russell Terrier
I'm Lucy, a 12 year old female Labrador Retriever
I'm Luna, a 7 year old female Poodle
I'm Max, a 5 year old male Poodle
I'm Max, a 10 year old male Great Dane
I'm Milo, a 12 year old male German Shepherd
I'm Teddy, a 6 year old male Maltese
I'm Teddy, a 6 year old male Cavalier King Charles Spaniel
I'm Toby, a 1 year old male Alaskan Malamute
```

step 6) End of Task 2

6 Pros and Cons of the Comparable Interface

6.1 Pros

1. The `Comparable` interface is implemented frequently in practice. That's because most collection of objects require only one ordering definition on the objects in the collection. In other words, you often choose to implement interface `Comparable` when you want to sort your objects in only one way.

2. Offers a quick 2-step way to turn your objects into comparable objects:

step 1) Have your class implement `Comparable<T>` like so:

```java
public class T implements Comparable<T>
```

where `T` represents the type of the objects being compared, such as `PetDog`; for example:

```java
public class PetDog implements Comparable<PetDog>
```

step 2) Override the `Comparable`'s only method `int compareTo(T o)` in class `T`

```java
class T implements Comparable<T>
{
   @Override
   public int compareTo(T other)
   {
      // compare this and other objects according to how you
      // want to define ordering on the objects of class T
      // return a negative integer to indicate this < other
      // return zero to indicate this = other
      // return a positive integer to indicate this > other
   }
// ...
// ... remaining members
}
```

6.2 Cons

- Poor choice when you need to sort your objects in multiple ways. Each way requires that you to modify the source code in your existing `compareTo()` override and recompile, forcing you to run the risk of introducing errors in an otherwise healthy code, and to potentially lose original source code. You experienced all that, moving between your Task 1 and Task 2 above. Bad idea!

There has to be a better way to define and redefine ordering on the objects of a class without having to mess with the class itself.

7 Welcome to Comparator Interface

Java's `Comparator` interface provides a way for us to compare two objects of the same type outside their class, hence eliminating the need to modify the class itself.

Generally used as follows, the `Comparator` interface specifies only one method, named `compare()` (not `compareTo()`):

```java
class T implements Comparator<T>
{
   @Override
   int compare(T o1, T o2)
   {
      // compare o1 and o2 according to how you want
      // to define ordering on the objects of class T
      // return a negative integer to indicate o1 < o2
      // return zero to indicate o1 = o2
      // return a positive integer to indicate o1 > o2
   }
}
```

Repeating it several times below, here is the main idea in the context of our `PetDog` class.

8 Task 3: Sorting dogList by Name, Age, and Breed, all in the same program

step 1) Define a class named, say, `SortDogsByName` that implements `Comparator<PetDog>`:

```java
class SortDogsByName implements Comparator<PetDog>
{   // used for sorting in ascending order of names
    public int compare(PetDog p1, PetDog p2)
    {   // delegate our task to String's compareTo
        return p1.getName().compareTo(p2.getName());
    }
}
```

step 2) Define a class named, say, `SortDogsByAge` that implements `Comparator<PetDog>`:

```java
class SortDogsByAge implements Comparator<PetDog>
{   // used for sorting in ascending order of age
    public int compare(PetDog p1, PetDog p2)
    {
        return p1.getAge() - p2.getAge(); // equivalent to code lines 16-21, page 6
    }
}
```

step 3) Define a class named, say, SortDogsByBreed that implements Comparator<PetDog>:

```java
class SortDogsByBreed implements Comparator<PetDog>
{   // used for sorting in ascending order of breed
    public int compare(PetDog p1, PetDog p2)
    {   // delegate our task to String's compareTo
        return p1.getBreed().compareTo(p2.getBreed());
    }
}
```

step 4) Define a class named, say, SortDogsByNameAge that implements Comparator<PetDog>:

```java
class SortDogsByNameAge implements Comparator<PetDog>
{   // used for sorting in ascending order of name, sorting by age for same names
    public int compare(PetDog p1, PetDog p2)
    { // delegate our task to String's compareTo
      int result = p1.getName().compareTo(p2.getName());

      // if the names are not the same return result
      if( result != 0) return result;
      // names are the same, so sort by age (effectively equivalent to
      return p1.getAge() - p2.getAge(); // code on lines 16-21, page 6
   }
}
```

step 5) Modify your driver class as follows.

```java
import java.io.FileNotFoundException;
import java.util.ArrayList;
import java.util.Collections;

public class TextFileIoTestDriver
{
   public static void main(String[] args) throws FileNotFoundException
   {
      String inFileName = "./dog_infile.txt"; // must exist in the project folder
      ArrayList<PetDog> dogList = TextFileIO.readTextFileToArrayList(inFileName);

      // sort dogList by name
      Collections.sort(dogList, new SortDogsByName());
      String outFile =  "./dogs_by_name.txt";
      TextFileIO.writeArrayListToTextFile(dogList, outFile);
      System.out.println("finished writing to " + outFile);

      // sort dogList by age
      Collections.sort(dogList, new SortDogsByAge());
      outFile =  "./dogs_by_age.txt";
      TextFileIO.writeArrayListToTextFile(dogList, outFile);
      System.out.println("finished writing to " + outFile);
```

```java

    // sort dogList by breed
    Collections.sort(dogList, new SortDogsByBreed());
    outFile =  "./dogs_by_breed.txt";
    TextFileIO.writeArrayListToTextFile(dogList, outFile);
    System.out.println("finished writing to " + outFile);

    // sort dogList by name
    Collections.sort(dogList, new SortDogsByNameAge());
    outFile =  "./dogs_by_name_age.txt";
    TextFileIO.writeArrayListToTextFile(dogList, outFile);
    System.out.println("finished writing to " + outFile);

    return;
  }
}
```

step 6) End of Task 3

Program Output on Screen

```
finished writing to ./dogs_by_name.txt
finished writing to ./dogs_by_age.txt
finished writing to ./dogs_by_breed.txt
finished writing to ./dogs_by_name_age.txt
```

./dogs_by_name.txt

```
I'm Archie, a 7 year old male Poodle (Miniature)
I'm Bear, a 8 year old male Dachshund
I'm Bella, a 12 year old female Yorkshire Terrier
I'm Daisy, a 1 year old female Pug
I'm Duke, a 8 year old male French Bulldog
I'm Frankie, a 9 year old male Staffordshire Bull Terrier
I'm Frankie, a 12 year old male Jack Russell Terrier
I'm Lucy, a 12 year old female Labrador Retriever
I'm Luna, a 7 year old female Poodle
I'm Max, a 10 year old male Great Dane
I'm Max, a 5 year old male Poodle
I'm Milo, a 12 year old male German Shepherd
I'm Teddy, a 6 year old male Maltese
I'm Teddy, a 6 year old male Cavalier King Charles Spaniel
I'm Toby, a 1 year old male Alaskan Malamute
```

./dogs_by_age.txt

```
I'm Daisy, a 1 year old female Pug
I'm Toby, a 1 year old male Alaskan Malamute
I'm Max, a 5 year old male Poodle
I'm Teddy, a 6 year old male Maltese
I'm Teddy, a 6 year old male Cavalier King Charles Spaniel
I'm Archie, a 7 year old male Poodle (Miniature)
I'm Luna, a 7 year old female Poodle
I'm Bear, a 8 year old male Dachshund
I'm Duke, a 8 year old male French Bulldog
I'm Frankie, a 9 year old male Staffordshire Bull Terrier
I'm Max, a 10 year old male Great Dane
I'm Bella, a 12 year old female Yorkshire Terrier
I'm Frankie, a 12 year old male Jack Russell Terrier
I'm Lucy, a 12 year old female Labrador Retriever
I'm Milo, a 12 year old male German Shepherd
```

./dogs_by_breed.txt

```
I'm Toby, a 1 year old male Alaskan Malamute
I'm Teddy, a 6 year old male Cavalier King Charles Spaniel
I'm Bear, a 8 year old male Dachshund
I'm Duke, a 8 year old male French Bulldog
I'm Milo, a 12 year old male German Shepherd
I'm Max, a 10 year old male Great Dane
I'm Frankie, a 12 year old male Jack Russell Terrier
I'm Lucy, a 12 year old female Labrador Retriever
I'm Teddy, a 6 year old male Maltese
I'm Max, a 5 year old male Poodle
I'm Luna, a 7 year old female Poodle
I'm Archie, a 7 year old male Poodle (Miniature)
I'm Daisy, a 1 year old female Pug
I'm Frankie, a 9 year old male Staffordshire Bull Terrier
I'm Bella, a 12 year old female Yorkshire Terrier
```

./dogs_by_name_age.txt

```
I'm Archie, a 7 year old male Poodle (Miniature)
I'm Bear, a 8 year old male Dachshund
I'm Bella, a 12 year old female Yorkshire Terrier
I'm Daisy, a 1 year old female Pug
I'm Duke, a 8 year old male French Bulldog
I'm Frankie, a 9 year old male Staffordshire Bull Terrier
I'm Frankie, a 12 year old male Jack Russell Terrier
I'm Lucy, a 12 year old female Labrador Retriever
I'm Luna, a 7 year old female Poodle
I'm Max, a 5 year old male Poodle
I'm Max, a 10 year old male Great Dane
I'm Milo, a 12 year old male German Shepherd
I'm Teddy, a 6 year old male Cavalier King Charles Spaniel
I'm Teddy, a 6 year old male Maltese
I'm Toby, a 1 year old male Alaskan Malamute
```

9 Confusion!

Many programmers, including myself, mix up `Comparable` and `Comparator` interfaces. They almost look and sound alike but play different roles. Just remember that

- `Comparable<T>` specifies `int compareTo(T o)`
- `Comparator<T>` specifies `int compare(T o1, T o2)`

10 More Java Interface Examples

Please remember: an interface specifies what a class must do but not how.

WashAndDry.java

```java
interface WashAndDry // implicitly public including all members
{
   void wash(); // does not specify how to wash
   void dry();  // does not specify how to dry
}
```

Car.java

```java
public class Car implements WashAndDry
{
   private String brand;
   private String model;
   private int year;

   @Override
   public void wash(){ // Car specifies how to wash
      System.out.println("washing my " + year + " " + brand + " " + model);
   }

   @Override
   public void dry(){ // Car specifies how to dry
      System.out.println("drying my " + year + " " + brand + " " + model);
   }

   public Car(String brand, String model, int year){
      this.brand = brand;
      this.model = model;
      this.year = year;
   }

   public int getYear(){return year;}
   public void setYear(int year){this.year = year;}
   public String getModel(){return model;}
   public void setModel(String model){this.model = model;}
   public String getBrand(){return brand;}
   public void setBrand(String brand){this.brand = brand;}
}
```

T_shirt.java

```
public class T_shirt implements WashAndDry
{
   private String name;
   private int size;

   @Override
   public void wash(){ // T_shirt specifies how to wash
      System.out.println("washing my size " + size + " " + name +" T_shirt");
   }

   @Override
   public void dry(){ // T_shirt specifies how to dry
      System.out.println("drying my size " + size + " " + name +" T_shirt");
   }

   public T_shirt(String name, int size){
      this.name = name;
      this.size = size;
   }

   public int getSize(){return size;}
   public void setSize(int size){this.size = size;}
   public String getName(){return name;}
   public void setName(String name){this.name = name;}
}
```

Dish.java

```
public class Dish implements WashAndDry{ ... }
```

People.java

```
public class People implements WashAndDry{ ... }
```

Vegtable.java

```
public class Vegtable implements WashAndDry{ ... }
```

Even though Car, T_shirt, Dish, People, and Vegtable are all seemingly unrelated classes, they all implement the same functionality as specified by the WashAndDry interface. Hence, the power of interface: an interface specifies what a class must do but not how.

WashAndDryApp.java

```java
public class WashAndDryApp
{
   // this method can take as argument any object of type T_shirt, Car, Dish,
   // People, and Vegtable because they all implement the same interface WashAndDry
   public static void doSomething(WashAndDry wd)
   {
      wd.wash();
      wd.dry();
   }
   // this method can take as argument only Car objects
   public static void doSomethingCar(Car c)
   {
      c.wash();
      c.dry();
   }
   // this method can take as argument only T_shirt objects
   public static void doSomethingTee(T_shirt t)
   {
      t.wash();
      t.dry();
   }
   public static void main(String[] args)
   {
      T_shirt my_shirt = new T_shirt("Polo", 15);
      Car my_car = new Car("Toyota", "Corolla", 2020);

//      doSomethingCar(my_shirt); // pass a T_shirt to doSomethingCar() // error
                                  // cannot pass a T_shirt for a Car

      doSomethingCar(my_car);     // pass a Car to doSomethingCar()     // ok

      doSomethingTee(my_shirt);   // pass a T_shirt to doSomethingTee() // ok
//      doSomethingTee(my_car);   // pass a Car to doSomethingTee()
                                  // cannot pass a car for a T_shirt    // error

      doSomething(my_shirt);      // pass a T_shirt to doSomething()    // ok
      doSomething(my_car);        // pass a Car to doSomething()        // ok
   }
}
```

Output

```
washing my 2020 Toyota Corolla
drying my 2020 Toyota Corolla
washing my size 15 Polo T_shirt
drying my size 15 Polo T_shirt
washing my size 15 Polo T_shirt
drying my size 15 Polo T_shirt
washing my 2020 Toyota Corolla
drying my 2020 Toyota Corolla
```

11 Evaluation Criteria

Evaluation Criteria

Functionality	Ability to perform as required, producing correct output for any set of input data, Proper implementation of all specified requirements, Efficiency	60%
Robustness	Ability to handle input data of wrong type or invalid value	10%
OOP style	Encapsulating only the necessary data inside objects, Information hiding, Proper use of Java constructs and facilities.	10%
Documentation	Description of purpose of program, Javadoc comment style for all methods and fields, comments on non-trivial steps in all methods	10%
Presentation	Format, clarity, completeness of output, user friendly interface	5%
Code readability	Meaningful identifiers, indentation, spacing, localizing variables	5%