Pre-Class Work 4: Comparable / Abstract Classes

Comparable [Background Reading]

□ Motivation

As you begin to gain more experience in writing classes, one of the questions you might have asked yourself is how we can compare two instances of a class.

For example, suppose you had a class called `Academic` which represents an academic member of the UW community (whether it be student or faculty) and that class stored a person's ID and their name:

```
public class Academic {
    private int id;
    private String name;

    public Academic(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
    public String toString() {
        return "name: " + name + ", id: " + id;
    }
}
```

Now, we can create some academic people:

```
public class Client {
    public static void main(String[] args) {
        Academic brett = new Academic(491, "Brett Wortzman");
        Academic kasey = new Academic(175, "Kasey Champion");
        Academic hunter = new Academic(924, "Hunter Schafer");
        Academic miya = new Academic(328, "Miya Natsuhara");
    }
}
```

If we threw each `Academic` person into a `TreeSet`, we should ideally see "Kasey Champion, Miya Natsuhara, Brett Wortzman, Hunter Schafer" in that order, since Kasey has the earlier ID (lowest ID value) while Hunter Schafer has the later ID (greatest ID value).

At the moment though, if we added each person into a `TreeSet`, we would encounter an error:

```
import java.util.*;
```

```
public class Client {
    public static void main(String[] args) {
        Academic brett = new Academic(491, "Brett Wortzman");
        Academic kasey = new Academic(175, "Kasey Champion");
        Academic hunter = new Academic(924, "Hunter Schafer");
        Academic miya = new Academic(328, "Miya Natsuhara");
        Set<Academic> set = new TreeSet<>();
        set.add(brett);
        set.add(kasey);
        set.add(hunter);
        set.add(miya);
        for (Academic person : set) {
            System.out.println(person);
        }
    }
}
```

While our code compiles, this is what happens during runtime

```
Exception in thread "main" java.lang.ClassCastException: class Academic cannot be cast to class jav
```

The reason why we get an error is because the `TreeSet` has no idea how to arrange each `Academic`! We never defined a concrete way for the `TreeSet` to sort each person. How can we fix this?

`Comparable` Interface

The `Comparable` interface in Java is used to define a natural ordering for a class. Once implemented, it is used to provide a way for objects of a class to compare with one another, so that they can be sorted. To implement `Comparable` is relatively straightforward as we only need to provide an implementation for one method: `compareTo(T other)`.

To implement the `Comparable` interface, you need to include the keywords `implements Comparable<T>`. Note that `T` represents the class you are comparing against (should be the same type as the class it is being written in!).

For example, consider the code below:

```
public class Academic implements Comparable<Academic> {
    private int id;
    private String name;

    public Academic(int id, String name) {
        this.id = id;
        this.name = name;
    }

    @Override
```

```
    public String toString() {
        return "name: " + name + ", id: " + id;
    }

    @Override
    public int compareTo(Academic other) {
        // TODO
    }
}
```

Here, if we want to implement the `Comparable` interface for `Academic`, we need to say `implements Comparable<Academic>`. Now, the only method we need to override now is the `compareTo()` method.

The `compareTo()` method compares the current object to another object of the same class and returns an integer indicating their relative order. The `compareTo()` method should return a **negative integer if the current object is less than the specified object, zero if the current object is equal to the specified object, and a positive integer if the current object is greater than the specified object**.

Conventionally speaking, you will see many implementations `compareTo()` use -1 as the negative integer and 1 as the positive integer.

If we want to create an order based on the ID of each person, how can we do that? Well, we can do so with the following:

```
@Override
public int compareTo(Academic other) {
    if (this.id < other.id) {
        return -1;
    } else if (this.id == other.id) {
        return 0;
    } else {
        return 1;
    }
}
```

Now that we have implemented the `compareTo` method, we can throw `Academic` into a `TreeSet` and it will automatically sort the elements out! Each element inside the `TreeSet` will use the `compareTo` method to compare themselves to the other elements.

For example, consider the following:

```
Academic brett = new Academic(491, "Brett Wortzman");
Academic kasey = new Academic(175, "Kasey Champion");
Set<Academic> set = new TreeSet<>();
set.add(person1);
set.add(person2);
```

Under the hood `brett` and `kasey` will compare themselves to each other with the `compareTo()`

method.

Let's consider the perspective of `brett`. `brett` has an ID of 491 and `kasey` has an ID of 175. When `brett` calls `compareTo()` on `kasey`, the condition `this.id < other.id` will be false for `brett` so he will return a 1 indicating that he should be "greater than" `kasey`.

Conversely, in the perspective of `kasey`, when `kasey` calls `compareTo()` on `brett`, the `else` condition will be false for `kasey` so she will return a -1 indicating that she is "less than" `brett`. With this information, the `TreeSet` knows to put `brett` earlier in the collection and `kasey` later in the collection.

If you printed the `set` out, you would see:

```
["name: Kasey Champion, id: 175", "name: Brett Wortzman, id: 491"]
```

If you wanted `brett` to be appear earlier in the list, we can negate the return values:

```
public int compareTo(Academic other) {
    if (this.id < other.id) {
        return 1;
    } else if (this.id == other.id) {
        return 0;
    } else {
        return -1;
    }
}
```

With this implementation, the `set` would printed out would be:

```
["name: Brett Wortzman, id: 491", "name: Kasey Champion, id: 175"]
```

In this implementation, we are saying that ID values that are higher should be considered "less than" ID values which are lower.

✏ `compareTo` pattern

In cases where you are only comparing a single value, we can actually use the values to return a result! This is a common pattern you'll see in many implementations of `compareTo`. More concretely, instead of all of the `if/else` statements, we can instead have:

```
public int compareTo(Academic other) {
    return this.id - other.id;
}
```

Here, we perform a calculation and use that result as our return value. Why does this work?

Recall that when implementing `compareTo`, we only need a negative integer to indicate if an object is "less than" another object. For something to be considered "greater than" we use a positive integer. It doesn't actually matter what value you use for negative and positive integers, as long as they are negative or positive!

Here, if `this.id` is less than `other.id` then `this.id - other.id` will return a negative value which will indicate that `this` object is "less than" the `other` object. The same principle applies for when `this.id` is greater than `other.id` since `this.id - other.id` would return a positive value!

Finally, notice that if both ID values are equal, it will return a 0 since `this.id - other.id` would cancel each other out!

□ Main Points

- When working with classes, we often want to compare instances of a class, such as by sorting objects or defining their natural ordering.
    - We can use the `Comparable` interface to define a natural ordering for our class, so our class can be compared and sorted.
- To implement `Comparable`, our class should include the declaration `implements Comparable<T>`, where `T` is the class being compared e.g. `public class Academic implements Comparable<Academic> {}`
- When implementing the `Comparable` interface, we should make sure to include the method `compareTo(T other)` in our class.
- The `compareTo(T other)` method compares the current object to another object of the same class and returns an integer.
    - This integer is **negative** if the current object is **less** than the *other* object.
    - This integer is **positive** if the current object is **greater** than the *other* object.
    - This integer is **zero** if the current object is **equal** to the *other* object.
- After our class implements `Comparable`, we can easily sort objects in our class and we will know that data structures such as `TreeSet`s will properly sort objects in our class!

Abstract Classes [Background Reading]

We can think of abstract classes as a combination of two things we have learned about already: inheritance and interfaces. Let's try to understand why abstract classes are useful with an example.

□ Shapes ○

Imagine that you are working for a graphic design company, *Champortzman Designs*. They want you to come up with a way to represent a bunch of shapes, which have a certain color and opacity. You start with the following attempt of just a square and a circle:

```
public class Square {
    private String color;
    private double opacity;
    private double sideLength;

    public Square(String color, double opacity, double sideLength) {
        this.color = color;
        this.opacity = opacity;
        this.sideLength = sideLength;
    }

    public String getColor() {
        return color;
    }

    public double getArea() {
        return sideLength * sideLength;
    }

    public double getDiagonal() {
        return Math.sqrt(2 * sidelength * sideLength);
    }
}

public class Circle {
    private String color;
    private double opacity;
    private double radius;

    public Circle(String color, double opacity, double radius) {
        this.color = color;
        this.opacity = opacity;
        this.sideLength = radius;
    }
```

```
    public String getColor() {
        return color;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
}
```

These two classes definitely get the job done for squares and circles, but notice how much they share in common! They share two fields (`color` and `opacity`), the exact same method and implementation for `getColor()` , and the same method signatures for `draw()` and `getArea()` !

How do we deal with this overlap? ***Abstract Classes!***

□ Abstract Classes

Let's take a look at a possible abstract class for our shapes above:

```
public abstract class Shape {
    private String color;
    private double opacity;

    public abstract double getArea();

    public Shape(String color, double opacity) {
        this.color = color;
        this.opacity = opacity;
    }

    public String getColor() {
        return color;
    }
}
```

Let's notice a couple of key features of this class

- The class *must* contain the `abstract` keyword between `public` and `class`
- Methods that do not contain an implementation *must* contain the `abstract` keyword between `public` and the return type
- Fields can be declared the same way as you would in a non-abstract class
- Non-abstract methods can be declared and implemented as you would in a non-abstract class
- Constructors follow all of the normal rules, except that they *may not be called at runtime!*

Take a look at the `Shape` abstract class. What parts are similar to an interface (like `Comparable`)? What parts are similar to a superclass like the TV Show example from lecture?

Abstract classes like `Shape` are useful when you want to **provide common implementation** to certain classes *and* **guarantee that other methods are implemented**.

Let's see how `Shape` changes our other two classes.

```
public class Square extends Shape {
    private double sideLength;

    public Square(String color, double opacity, double sideLength) {
        super(color, opacity);
        this.sideLength = sideLength;
    }

    public double getArea() {
        return sideLength * sideLength;
    }

    public double getDiagonal() {
        return Math.sqrt(2 * sidelength * sideLength);
    }
}

public class Circle extends Shape {
    private double radius;

    public Square(String color, double opacity, double sideLength) {
        super(color, opacity);
        this.radius = radius;
    }

    public double getArea() {
        return Math.PI * radius * radius;
    }

    public double getCircumference() {
        return 2 * Math.PI * radius;
    }
}
```

Notice that since `Circle` and `Square` extends `Shape` , they do not need the fields `color` or `opacity` or the method `getColor()` . It gets carried over from `Shape` (just like a subclass would). However, these classes do need to implement `getArea()` (just like a class that implements an interface). With this example, we can also see why an unusable constructor in an abstract class might be helpful.

Using the `super` keyword, Circle and Square only need to worry about the fields that they add on (that are unique to them). The abstract class constructor will take care of all of the common fields.

Notice that we use the `extends` keyword, even though we also have some `implements` functionality!

Make sure that the call to the parent constructor using `super` is always the first line of your constructor!

Main Points

- Abstract classes contain both
  - Fields and fully implemented methods
  - `abstract` methods that are described by only a method signature
  - Constructors that can initialize fields (but may not be used since abstract classes may not be instantiated)
- Classes that extend abstract classes
  - Have access to fields and fully implemented methods of the abstract class
  - **Must** implement any `abstract` methods declared by the abstract class
  - May use the constructor of the abstract class using the `super` keyword

Academic [Programming Question]

Modify the `Comparable` interface in `Acacdemic` such that higher ID values are considered "less than" lower ID values and lower ID values are considered "greater than" higher ID values. Implement this without using an `if/else` structure.

Owls [Programming Question]

Implement the Comparable interface in the class `Owl`. Most of the class has been written you just need to write the `compareTo()` method. The order of the Owls should be sorted in **reverse** alphabetical order.

Remember that the `compareTo()` method returns a negative integer, zero, or a positive integer if the **current** object is less than, equal to, or greater than the **given** object.

More Birds! [Programming Question]

Write another class called `Duck` that extends the `Bird` abstract class.

The new Duck class should have the two following requirements:

1. Include an extra method called `swim()` for the `Duck` class. This method should print the name of the duck and that it's swimming around. For example, if I create a `Duck` with the name "Alfred". Calling `swim()` should produce the output:

   ```
   "Alfred the duck is swimming around."
   ```

   The name of the bird should be interchangeable depending on what was given in the constructor.
2. Implement the `fly()` method to print the `wingspan` of the bird and to show that the bird is flying. For example, if we called `fly()` on Alfred, it should produce the output:

   ```
   "Alfred the duck flies with its 7 inch wings at a high speed."
   ```

   The length of the wingspan should depends on the value that is given in the constructor.