Interfaces

Interfaces are a very incomplete version of a class, to the point where, they really can't be called a class any more. Rather, it's essentially a contract to write a set of methods. An interface must list a set of method signatures. There is no code in the interface, but rather a list of methods with the names given, the return type given, as well as the parameter list. Other classes may implement an interface. To do so, the class must implement the methods listed in the interface, matching their signature. Ideally, these methods should do the "task" the interface infers.

Java has some built in interfaces but also allows the user to create their own. One of the most common interfaces built into Java is the Comparable Interface.

Comparable Interface

In order to implement this interface, you must write a method with the following prototype in your class (if you were creating the Time class):

```
public int compareTo(Time time2);
```

Obviously, if you were creating a class other than the Time class, this is what the type of the input parameter would be. The job of this method is as follows:

The method should return a negative integer if the current object (this) comes BEFORE t. The method should return 0 if the two objects (this and t) are equal. The method should return a positive integer if this object comes AFTER t.

Thus, this interface should ONLY be implemented if two different objects from the class can be compared in order. This should be fairly easy to add to our Time class:

```
public int compareTo(Time time2) {
    return this.totalminutes() - t.totalminutes();
}
```

Notice how the mathematical expression listed above corresponds very nicely to the requirements of the compareTo method. Often times, a series of if statements is used to implement a compareTo method, but in this particular case, the mathematics works out nicely so that an if statement can be avoided.

In order to truly implement the interface, this final change has to be made at the beginning of the class:

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

Once we add this, then we can sort Time objects. Consider the main method on the following page that randomly generates several Time objects in an array, prints these out, sorts them, and then prints them out again:

```
    public static void main(String[] args) {

        System.out.println("Times in original order.");
        Random r = new Random();
        Time[] list = new Time[N];
        for (int i=0; i<N; i++) {
            list[i] = new Time(r.nextInt(500));
            System.out.println(list[i]);
        }
        System.out.println();

        Arrays.sort(list);

        System.out.println("Here are my Time objects sorted!");
        for (int i=0; i<N; i++)
            System.out.println(list[i]);
    }
```

The beauty if this is that Java's Arrays.sort method can use Time's compareTo method to do its task and get the items in order. Thus, the user defines the "ordering" of two items via the compareTo method, which must be consistent (we can't have a < b, b < c and c < a, for example), and then using that definition, the method sort is able to sort the items in the list according to the programmer's definition. In order to do this, the class of the array passed to Arrays.sort must implement the Comparable interface.

Defining your own Interface in Java

Most interface definitions are extremely short. An interface could contain multiple method signatures, but often times, they just contain one. The two example interfaces discussed this lecture are the Scorable interface and the Addable interface, each which requires one method. Here is the full code for both:

```
public interface Scorable {
       public int getScore();
}

public interface Addable<T> {
       public T add(T other);
}
```

The latter uses a template, which means that instead of implementing Addable in general, if a class T implements Addable, it will implement Addable<T>, which means that the method signature contains some reference to the class itself that is implementing the interface. In the Addable interface, we want any class that implements it to write a method add that takes in an object of type T (same type as the class that is doing the implementing) and also returns another object of this same time.

Thus, the syntax here is quite easy. You use the keyword public (since interfaces are typically public), then the keyword interface, followed by the name of the interface and an open brace. This is followed by one or more method signatures, each ended with a semicolon, and then finally a close brace for the interface. Here is an interface with multiple method signatures:

```
public interface ArithmeticClass<T> {
        public T add(T other);
        public T subtract(T other);
        public T multiply(T other);
        public T divide(T other);
}
```

One could rewrite the fraction class fairly easily to implement this interface above.

For these notes, we'll write two classes that implement these interfaces:

(1) Card, which will implement the Scorable interface

(2) Monster, which will implement both interfaces

The latter example is to show you that a single class can implement multiple interfaces.

Here is the entire Card class (as it's pretty small):

```
public class Card implements Scorable {

        private String suit;
        private int number;

        public Card(String s, int n) {
                suit = s;
                number = n;
        }

        public int getScore() {
                return number + suit.length();
        }

        public String toString() {
                return number + " of "+suit;
        }
}
```

The programmer can give any definition to getScore(). Here we have a pretty strange one. We add the number to the number of characters in the string suit.

Now, consider creating a Monster class. Each monster will have a name as well as a list of moves it can perform. Here is the beginning of the class to show the syntax for stating that a class implements multiple interfaces:

```
public class Monster implements Scorable, Addable<Monster> {

        private String name;
        private ArrayList<String> moves;
```

Here is the getScore method:

```
        public int getScore() {
                return moves.size();
        }
```

This is fairly simple - it "assumes" that each move is equal in some sense and the score of a monster is simply the number of moves it has.

To add two monster's together, I've decided to slice their names and put them together and then add the moves together to create one megamonster! Here is the code for the add method:

```
public Monster add(Monster other) {

        int len1 = this.name.length();
        int len2 = other.name.length();
        String newname = name.substring(0, len1/2) +
                         other.name.substring(0, len2/2) +
                         name.substring(len1/2) +
                         other.name.substring(len2/2);

        Monster mix = new Monster(newname);

        for (String s: moves)
                mix.addMove(s);
        for (String s: other.moves)
                mix.addMove(s);

        return mix;
}
```

Now that we have both interfaces implemented in the Monster class, we can use these objects outside of these classes and have these methods execute.

We can actually create references of the interface and as long as we cast them to the proper thing (the programmer needs to know what class each actual object belongs to before casting), we can call the appropriate methods.

Here is the main method with the tests:

```
	public static void main(String[] args) {

		// We can have references of type Scorable, but not objects.
		Scorable[] stuff = new Scorable[5];
		stuff[0] = new Card("Clubs", 7);
		stuff[1] = new Monster("Godzilla");
		((Monster)stuff[1]).addMove("punch");
		((Monster)stuff[1]).addMove("stepon");
		((Monster)stuff[1]).addMove("laser");
		stuff[2] = new Card("Diamonds", 4);
		stuff[3] = new Monster("Fizz");
		((Monster)stuff[3]).addMove("fish");
		((Monster)stuff[3]).addMove("sleep");
		stuff[4] = new Monster("Lazy");

		// We can call getScore and it compiles.
		for (int i=0; i<stuff.length; i++)
			System.out.println(stuff[i]+": "+stuff[i].getScore());

		// We have to cast a bunch of stuff here, but it works!)
		Monster biggie = ((Monster)stuff[1]).add((Monster)stuff[3]);
		System.out.println(biggie);

		Monster weird = ((Monster)stuff[4]).add((Monster)stuff[1]);
		System.out.println(weird);
	}
```

We set up an array of Scorable and initialize it. Then we print everything - note that the code compiles and can print out scores dynamically for each object…the cards naturally call the getScore in Card and the monster's naturally call getScore in Monster.

Then, we have to cast both to Monsters to call the add method. We could also write a fun static method as follows:

```
public static Monster combine(ArrayList<Monster> monsters) {

	Monster res = new Monster("");
	for (Monster grover: monsters)
		res = res.add(grover);

	return res;
}
```

Notice that since everything in the array is Monsters, and we know that res is one two, we can iteratively add together each of these Monsters!!! The name becomes really strange at the end!!!