Interfaces

A Java interface specifies a set of methods that any class that implements the interface must have. You can treat an interface like a type, which allows you to define methods for the interface and then have the code apply to all classes that implement the interface. However, an interface is not a class – it is a type that can be satisfied by any class that implements the interface. It basically is a property of a class that says what methods the class must have.

One way to look at an interface is an extreme form of an abstract class where every method is abstract.

Java also allows you to use interfaces as a way to approximate multiple inheritance. You cannot have multiple base classes in java, but you can have multiple interfaces for a class.

To define an interface use the same format as defining a class except:

1. Use the word “interface” instead of “class”

2. None of the methods should have any code, just the headings.

3. You can’t have any instance variables. You can have public static final variables (i.e., constants). If you define a normal instance variable Java treats it as a constant.

Here is a sample interface for a Driveable interface. It specifies what methods should exist for a vehicle that is driveable. If this was real we would have a lot more methods.

```
public interface Driveable
{
  public void forward();
  public void stop();
  public void turnLeft();
  public void turnRight();
}
```

To implement an interface a concrete class must:

1. add `implements InterfaceName` to the start of the class definition
   To implement more than one interface separate the interface names by commas.

2. The concrete class must implement all the method headings listed in the definition of the interface.

Here are two classes that implement the Driveable interface:

```
public class SportsCar implements Driveable
{
  public void forward()
  {
        System.out.println("Step on the accelerator");
  }
  public void stop()
  {
       System.out.println("Step on the brake");
  }
  public void turnLeft()
  {
       System.out.println("Turn steering wheel left");
  }
  public void turnRight()
  {
       System.out.println("Turn steering wheel right");
  }
}

public class Canoe implements Driveable
{
  public void forward()
  {
       System.out.println("Paddle on left and right");
  }
  public void stop()
  {
       System.out.println("Drag paddle on left and right");
  }
  public void turnLeft()
  {
       System.out.println("Paddle on left");
  }
  public void turnRight()
  {
       System.out.println("Paddle on right");
  }
}
```

Note that each class has to implement ALL the methods in the interface. If we skip one, the compiler will complain. It is allowable to add additional methods in the class that are not specified in the interface.

The advantage of defining an interface is that now we can write code that knows what methods exist, but the implementation of those methods can vary. For example, we could write navigation code to drive any vehicle:

```
public static void navigateVehicle(Driveable vehicle)
{
  vehicle.forward();
  vehicle.turnRight();
  vehicle.forward();
  vehicle.Stop();
```

```
    }
```

Our calling code could then send in either a Canoe or a SportsCar and the navigateVehicle method will be able to control either one:

```
SportsCar s = new SportsCar();
Canoe c = new Canoe();
navigateVehicle(s);
navigateVehicle(c);
```

If a class implementing an interface extends a base class, then add “extends” before the implements, e.g.:

```
public class Canoe extends WaterVehicle implements Driveable
```

This brings up another advantage of interfaces. While Java allows only one base class, sometimes it is desirable to have more than one. For example, a Canoe class could have both WaterVehicle and HumanPoweredVehicle as interfaces it implements. Here is an example with another interface named Ordered which is used to order items in a class:

```
public interface Ordered
{
  public boolean precedes(Object other);  // Why use object here?
}
```

Our SportsCar class might now look like:

```
public class SportsCar implements Driveable, Ordered
{
  public String VIN; // Need a way to set this
                     // not done for this example
  public void forward()
  {
       System.out.println("Step on the accelerator");
  }

  public void stop()
  {
       System.out.println("Step on the brake");
  }
  public void turnLeft()
  {
       System.out.println("Turn steering wheel left");
  }
  public void turnRight()
  {
       System.out.println("Turn steering wheel right");
```

```
        }
        public boolean precedes(Object other)
        {
               if (other == null)
                   return false;
               else if (!(other instanceof SportsCar))
                   return false;
               else
               {
                    SportsCar otherSportsCar = (SportsCar) other;
                    return (VIN.compareTo(other.VIN));
               }
        }
      }
```

Some of our code, for example, to sort an array of Ordered things, might use the precedes method while other parts of our code dealing with driving would assume the Driveable interface.

Extending an interface

Just like a class can extend another class, an interface can extend another interface. The result is the derived interface inherits the methods of the parent interface.

The Comparable Interface

Java has many predefined interfaces that are used by many classes. One of them is the Comparable interface and it is used to impose an ordering upon the objects that implement it. The Comparable interface has only one method heading. The method **compareTo** must be written for a class to implement the Comparable interface.

```
  public int compareTo(Object other);
```

The interface allows you to specify how one object compares to another in terms of when one should "come before" or "come after" or "equal" the other. It is the programmer's responsibility to follow the semantics appropriately. For example, if you define A to come before B, and B to come before C, you shouldn't allow C to come before A.

The compareTo method should return

- a negative number if the calling object "comes before" the parameter other,
- a zero if the calling object "equals" the parameter other,
- and a positive number if the calling object "comes after" the parameter other.

As an example, consider the idiom that one can't compare apples to oranges. Let's show that this idiom doesn't quite apply to Java classes because we can define the **compareTo** method as we see fit. Our first attempt to define a Fruit class to represent apples and oranges might look like the code below. This simple class uses a String to store the name of the fruit along with methods to get and set the name. The constructor takes the name of the fruit. This initial attempt does not implement any interfaces.

```java
public class Fruit
{
    private String fruitName;
    public Fruit()
    {
          fruitName = "";
    }
    public Fruit(String name)
    {
          fruitName = name;
    }
    public void setName(String name)
    {
          fruitName = name;
    }
    public String getName()
    {
          return fruitName;
    }
}
```

A short demo program is given below that attempts to use the Fruit class. In this example we make an array that contains four Fruit objects. Then we try to sort the array using the Arrays.sort method.

```java
import java.util.Arrays;

public class FruitDemo
{
    public static void main(String[] args)
    {
          Fruit[] fruits = new Fruit[4];

          fruits[0] = new Fruit("Orange");
          fruits[1] = new Fruit("Apple");
          fruits[2] = new Fruit("Kiwi");
          fruits[3] = new Fruit("Durian");

          Arrays.sort(fruits);

          // Output the sorted array of fruits
          for (Fruit f : fruits)
          {
                System.out.println(f.getName());
          }
    }
}
```

The program will compile and run but produce the following runtime error:

```
Exception in thread "main" java.lang.ClassCastException: Fruit cannot be
cast to java.lang.Comparable
```

This error occurs because Java doesn't know how to compare two instances of the Fruit class to each other to see if one "comes after" the other when attempting to sort the array. More precisely, the Arrays.sort method has been written with the expectation that the objects passed in the array have a compareTo method in accordance with the Comparable interface. Arrays.sort attempts to invoke compareTo on objects in the array (for example, to see if fruits[0] is greater than fruits[1]) so they can be rearranged in sorted order, but since the method doesn't exist there is an error.

The solution is to make sure that the Fruit class implements the Comparable interface with a compareTo method. One way we might compare one fruit to another is to use the lexicographic ordering of the fruit name. Lexicographic ordering is the same as alphabetical ordering when both strings are either all uppercase or all lowercase letters. For example, apples would come before oranges because the word "apple" is lexicographically ordered before "orange." To accomplish this we can use the compareTo method defined for the String class. That is, if we have two strings str1 and str2 then

```
str1.compareTo(str2)
```

will return a negative number if str1 is lexicographically before str2, the value 0 if str1 is equal to str2, and a positive number if str1 is lexicographically after str2. The compareTo method we write for the Fruit class can then return the result of the compareTo method for the names of the fruits being compared. The code below contains a version of the Fruit class with these changes highlighted.

```
public class Fruit implements Comparable
{
    private String fruitName;
    public Fruit()
    {
          fruitName = "";
    }
    public Fruit(String name)
    {
          fruitName = name;
    }
    public void setName(String name)
    {
          fruitName = name;
    }
    public String getName()
    {
          return fruitName;
    }
    public int compareTo(Object o)
    {
          if ((o != null) &&
              (o instanceof Fruit))
          {
                Fruit otherFruit = (Fruit) o;
                return (fruitName.compareTo(otherFruit.fruitName));
          }
          return -1;    // Default if other object is not a Fruit
    }
}
```

Now the program will run and produce the output:

```
Apple
Durian
Kiwi
Orange
```

The Arrays.sort method is now successful because it can call the compareTo method to compare objects in the array and reorder them. To drive home the point that our compareTo method is being called from within the Arrays.sort method, we can redefine compareTo with different criteria to sort fruit. Instead of lexicographic ordering, let's use the length of the fruit name as the comparison metric. Fruit with shorter names will come before fruit with longer names. Below is an alternate definition of the compareTo method:

```
public int compareTo(Object o)
{
      if ((o != null) &&
          (o instanceof Fruit))
      {
            Fruit otherFruit = (Fruit) o;
            if (fruitName.length() > otherFruit.fruitName.length())
                  return 1;
            else if (fruitName.length() <
                         otherFruit.fruitName.length())
                  return -1;
            else
                  return 0;

      }
      return -1;  // Default if other object is not a Fruit
}
```

The program will now produce the output:

```
Kiwi
Apple
Orange
Durian
```

The fruits are now sorted in order of shortest word first.