Interfaces in Java: Introduction using the Comparable interface

Interfaces are everywhere in java. You can write your own interfaces or you can **implement** one (or more) of Java's built-in interfaces.

I think the best way to introduce the concept is to implement one of the Java API's most commonly used interfaces: the **Comparable** interface.

The Comparable interface

If we look at this "thing" in the java API we see:

Interface Comparable<T>

Type Parameters:

T - the type of objects that this object may be compared to

Method Summary

Methods	
Modifier and Type	**Method and Description**
int	**compareTo**(**T** o) Compares this object with the specified object for order.

In fact the code for the Comparable interface would look like this:

```
public interface Comparable<T>
{
   public int compareTo(T o);
}
```

The important things to note at the moment are:

1. The interface consists of one or more abstract methods (the keyword abstract is not used in interfaces but you can tell that it is abstract in the sense that there's a semi-colon instead of curly braces). That's normally all that you find in an interface – blank methods that other classes (those that decide to **implement** the interface) will fill it in a way that makes sense for those classes.
2. The <T> is part of what is called generic programming. Don't worry about the specifics yet but you've already used generics quite a bit.
   Basically, the actual type of object can be filled in later (otherwise you'd have to end doing that much more horrible stuff with `instanceof` and casting that we have to do for overriding the `equals()` method – in fact, that was also the old way with interfaces until they came up with generics).

The String Class and its use of the Comparable interface

The String is one of many classes that implements the Comparable interface. You can verify this from the API.

In fact, the String class implements a number of interfaces. **This is a powerful concept: we can implement as many interfaces as we like. Compare this to inheritance: we can only inherit from <u>one</u> superclass.**

Because the String class implements the interface we now know that it provides an implementation of the compareTo () method. Let's use it:

```
//Demonstration of Comparable interface using Strings

public class StringComparableDemo1
{
   public static void main(String[] args)
   {
      String s1 = "abc";

      String s2 = "def";

      int compareResult = s1.compareTo(s2);


      if(compareResult == 0)
      {
         System.out.println("Strings are equal");
      }
      else if(compareResult > 0)
      {
         System.out.println("s1 is greater than s2");
      }
      else if(compareResult < 0)
      {
         System.out.println("s1 is less than s2");
      }
   }
}
```

If we run this it will print the third option (because 'd' is "greater than" 'a')

Okay, that's fine but here's a more powerful example:

```
//Demonstration of Comparable interface using Strings
import java.util.*;
public class StringComparableDemo2
{
   public static void main(String[] args)
   {
      ArrayList<String> list = new ArrayList<String>();

      list.add("fff");
      list.add("ddd");
      list.add("aaa");
      list.add("eee");

      //Print list
      System.out.println("List: " + list);

      Collections.sort(list);

      //Print list
      System.out.println("List: " + list);


   }
}
```

```
List: [fff, ddd, aaa, eee]

List: [aaa, ddd, eee, fff]
```

Now, we have a nice little static method in the Collections class that we can use to sort our ArrayList (as you can clearly see above).

In fact, that sort() method **depends upon** our objects implementing the Comparable interface (the method looks at/compares adjacent objects and sees if it needs to shuffle/move/reorder them. This is the essence of interfaces: The string class implements the interface and other classes/methods can then safely "plug into" the interface (it is useful to imagine, for example, an analogy with the USB interface in computers. Different peripherals do different things but as long as they all conform to the USB interface standard then the computer can communicate with them).

Okay, so the String class already provides this facility. Can we make our own classes provide it? Let's look at how to do it in the context of the BankAccount class.

Making our BankAccount class Comparable

I'll just show the relevant code here (the full code is in a subfolder of the lecture folder)

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

   private double balance;

   //Note how the compareTo is supposed to work:
   //From the API:
   //Returns a negative integer, zero, or a positive
   //integer as this object is less than, equal to, or greater than the
specified object.
   public int compareTo(BankAccount other)
   {
      //I need to cast because i'm subtracting 2 doubles and the
      //return type is of type int
      return (int)(this.balance - other.balance);[1]
   }
}

//Tester for comparing BankAccount objects.
import java.util.*;
public class BankAccountComparableTester
{
   public static void main(String[] args)
   {
      //Let's create a list with some BankAccount objects
      ArrayList<BankAccount> list = new ArrayList<BankAccount>();
      list.add(new BankAccount(100));
      list.add(new BankAccount(70));
      list.add(new BankAccount(90));
      list.add(new BankAccount(40));
      list.add(new BankAccount(80));

      //Print out the list
      System.out.println("Original list:\n " + list);

      //Sort the list - this is only possible if the sort method
      //can compare our objects (i.e. we implement the Comparable
      //interface). Note that the sort method
      //will actually call the compareTo method to do its job.
      //You could say that it "plugs in" to the interface we've provided
      Collections.sort(list);

      //Print out the list again
      System.out.println("Sorted list:\n " + list);
```

[1] Note: Using this mechanism (subtracting doubles and then casting the result to an int) actually causes a bug (that I didn't consider when I wrote the lecture). What will happen if, for example, one balance is 100.4 and another 100.2? How could you fix this?

```
    }
}
```

One thing to note about the example above: the compareTo() method is not called anywhere in the code above – it is actually called internally by the sort() method in the Collections class. **That's kind of amazing; "Old Code" is calling our "New Code". This is one big advantage of interfaces – if we implement an interface, other code can quite easily "plug into it".**

Of course, ***we*** could also call the compareTo() method in our tester if we wanted to:

```
//Simple demonstration of calling the compareTo() method.
import java.util.*;
public class BankAccountComparableTester2
{
   public static void main(String[] args)
   {
      //Change the balances to test the if - elseif conditions
      BankAccount b1 = new BankAccount(100);
      BankAccount b2 = new BankAccount(90);

      int returnVal = b1.compareTo(b2);

      if(returnVal == 0)
      {
         System.out.println("b1 and b2 both have the same money");
      }
      else if(returnVal < 0)
      {
         System.out.println("b1 has less money than b2");
      }
      else //MUST be > 0
      {
         System.out.println("b1 has more money than b2");
      }

   }
}
```

Making our SavingsAccount class Comparable

Because SavingsAccount inherits from BankAccount it also inherits the interface. It can**, if it wishes**, override the compareTo() method to provide its own implementation.

The question is, though, how do we compare SavingsAccounts? In the code below we'll look at one possible answer, but it's important to realise that this is just one possibility (and it's probably not a great one, if you consider it).

```
public class SavingsAccount extends BankAccount //implements
Comparable<SavingsAccount>
{
   private double interestRate;

   //constructors etc. left out for brevity

   //Because BankAccount implements the Comparable interface we can
   //simply override the compareTo method in the subclass
   public int compareTo(SavingsAccount other)
   {
      //NOTE: how do we want to compare savings accounts
      //1: based on balance (handled by superclass
      //2: based on interest rate
      //3: based on a combination of both
      //Well, that's up to us. In this example we first check the balances
      //is they're equal we then use the interest rate to determine a
      //less-than/greater-than relationship
      int retVal = super.compareTo(other);

      if (retVal != 0)
     {
         return retVal;
      }
      else
      {
         return (int)(this.interestRate - other.interestRate);
      }
   }

}
```

```
public class SavingsAccountComparableTester
{
   public static void main(String[] args)
   {
      SavingsAccount s1 = new SavingsAccount(20,200);
      SavingsAccount s2 = new SavingsAccount(10,200);


      int returnVal = s1.compareTo(s2);

      if(returnVal == 0)
      {
         System.out.println("s1 and s2 are equal");
      }
      else if(returnVal < 0)
      {
        System.out.println("s1 is less than s2, but what does this mean??");
      }
      else //MUST be > 0
      {
        System.out.println("s1 is greater than s2, but what does this mean??");
      }

   }
}
```

Okay, well we've made our SavingsAccount objects comparable (and therefore, capable of being sorted by Collections.sort()), but it's hardly adequate based on the example given above.

It would probably be nicer if we could sort based on **either** balance or interestRate. If we want to do something like that (think of an email client where you want to sort based on date, sender, subject, etc.) then we use a different interface called **Comparator**.

It is important to note that the Collections.sort uses an object's Comparable interface by default but we can invoke an overloaded version to use a Comparator object.

From Collections API: `void sort(List<T> list)`

From Collections API: `void sort(List<T> list, Comparator<? super T> c)`

Here are two Comparator objects which we can use:

```
public class RateComparator implements Comparator<SavingsAccount>
{
   public int compare(SavingsAccount ob1, SavingsAccount ob2)
   {
      return (int)(ob1.getInterestRate() - ob2.getInterestRate());
   }

   //Note: we should also provide the equals() method, but the one we
```

```
  //inherit from the Object class will allow this to compile

}

//////////////////////////////////////////////////////

public class BalanceComparator implements Comparator<SavingsAccount>
{
   public int compare(SavingsAccount ob1, SavingsAccount ob2)
   {
      return (int)(ob1.getBalance() - ob2.getBalance());
   }

   //Note: we should also provide the equals() method, but the one we
//inherit from the Object class will allow this to compile

}
```

Now, we should be able to sort either by balance or by interest rate:

```
public class SavingsAccountComparatorTester
{
   public static void main(String[] args)
   {
      //Let's create a list with some BankAccount objects
      ArrayList<SavingsAccount> list = new ArrayList<SavingsAccount>();
      list.add(new SavingsAccount(3, 100));
      list.add(new SavingsAccount(1, 70));
      list.add(new SavingsAccount(5, 90));
      list.add(new SavingsAccount(4, 40));
      list.add(new SavingsAccount(5, 80));

      //Print out the list
      System.out.println("Original list:\n " + list);

       //Sort the list - this is only possible if the sort method
      //can compare our objects. In this example we can sort either
      //by interest rate or by balance
      Collections.sort(list, new RateComparator());

      //Print out the list again
      System.out.println("Sorted list:\n " + list);

      Collections.sort(list, new BalanceComparator());

      //Print out the list again
      System.out.println("Sorted list:\n " + list);


   }
}
```

Now we have the ability to sort in all kinds of different ways, without having to write brand new sort methods for each one.