Lesson 33: *Comparable* and *Comparator* Interfaces

The purpose:

The purpose of both the *Comparable* and *Comparator* interfaces is to enable us to **compare objects.**

***Comparable* Interface:**

The *Comparable* interface contains only **one** method and is specified in *java.util.Arrays*:

```
public interface Comparable
{
        int compareTo(Object otherObject);  // a.compareTo(b)…returns a neg number if
                                            //a < b; returns a pos number if a > b; returns
                                            //0 if a = b.
}
```

Comparing objects:

The most obvious standard Java class that implements the *Comparable* interface is the *String* class. You can implement *Comparable* for your own classes too. Following is an example of a *BankAccount* class in which we will implement *Comparable*. First, we must decide what it means to compare two bank accounts.

1. Do we mean to compare the dates of when the two accounts were opened?
2. Do we mean to compare the amount on deposit (the balance)?
3. Do we mean to compare a "flaky factor" (number of times the account was overdrawn)?

For our example we will compare the amount on deposit (the balance) since this seems the most natural; however, it should be emphasized that **we can define the comparison in any way we might desire**.

First, let's examine how we will call this *compareTo* method. Assume that in some *Tester* class we have the following code.

```
//Create an account called myAccount with a balance of $40.
BankAccount myAccount = new BankAccount("Hilary", 40);

//Create an account called yourAccount with a balance of $135.
BankAccount yourAccount = new BankAccount("Kallie", 135);

//Now, compare these two objects using the compareTo method
int j;
j = myAccount.compareTo(yourAccount);

// If we test j with an if-statement we should see that it's a negative number since
// the balance in myAccount, 40, is less than the balance in yourAccount, 135.
```

And now here is the *BankAccount* class in which we will implement the *Comparable* interface:

```
public class BankAccount implements Comparable
{
        public BankAccount(String nm, double bal) //Constructor
        {
                name = nm;
                balance = bal;
        }
        . . . other methods . . .

        public int compareTo(Object otherObject)
        {
                //otherObject is passed in as an Object type so let's convert it into
                //a BankAccount type object.
                BankAccount otherAccount = (BankAccount) otherObject;

                int retValue;
                if (balance < otherAccount.balance)
                {
                        retValue = –1;
                }
                else
                {
                        if (balance > otherAccount.balance)
                        {
                                retValue = 1;
                        }
                        else
                        {
                                retValue = 0;
                        }
                }
                return retValue;
        }
        public String name;
        public double balance;
}
```

You may be concerned that the following line of code in *Tester*,

```
j = myAccount.compareTo(yourAccount);
```

is incompatible with the following line of code (the signature of the *compareTo* method)

```
public int compareTo(Object otherObject),
```

since *yourAccount* is a ***BankAccount*** object and *otherObject* is of type ***Object***. This is not a problem since **any** type object **can be** stored in an *Object* type object. However, if you wish to store an *Object* type object in some other object **it must be cast**.

Using *Comparable* with wrapper class numerics:

Integer and *Double* type variables work directly with the *compareTo* method as shown in the following examples:

***Integer* example:**

```
Integer x = 5; //pre Java 5.0, Integer x = new Integer(5);
Integer y = 17; //pre Java 5.0, Integer y = new Integer(17);
System.out.println( x.compareTo(y) ); //negative number
```

***Double* example:**

```
Double x = 52.5; //pre Java 5.0, Double x = new Double(52.5);
Double y = 11.8; //pre Java 5.0, Double y = new Double (11.8);
System.out.println( x.compareTo(y) ); //positive number
```

Casting *Object* type objects to *Comparable:*

Suppose you have the following type method that receives an *Object* type parameter. The reason we receive an *Object* type is so as to make this method as **general** as possible, i.e., so it can receive **any** type object:

```
public static void theMethod(Object obj)
{       }
```

There is, however, a problem if we wish to use *obj* with a *compareTo* method in the code portion of *theMethod*. *Object* does not implement the *Comparable* class nor does it have a *compareTo* method. There are two ways to solve this problem:

Receive *obj* as a *Comparable* object :

```
public static void theMethod(Comparable obj)
```

Cast *obj* as *Comparable*:

```
public static void theMethod(Object obj1)
{
        …some code…
        //assume obj2 is also of Object type
        int c = ( (Comparable)obj1 ).compareTo( obj2 );
        //Notice the nesting of the parenthesis above
        …some code…
}
```

Using the *compareTo* method for sorting:

Recall from Lesson 19 that we used *Arrays.sort(a)* to sort a numeric array. We use exactly this same syntax to sort an array of objects **if** the class for those objects has implemented the *Comparable* interface. To sort the array named *ba_array* of type

BankAccount in which *Comparable* has been implemented, simply issue the command:

```
Arrays.sort(ba_array);
```

Using generics with Comparable:

The *Comparable* interface is generic. (For more details see page 37-5 and Appendix AF.) The following is an example of how to use generics when implementing this interface:

```
public class MyClass implements Comparable<MyClass>
{
    //… other code
    public int compareTo(MyClass obj) { //implementing code }
}
```

***Comparator* Interface:**

Occasionally, we might need a *compareTo* method in a class that we don't own or is otherwise impossible for us to modify. Or, perhaps there is already a *compareTo* method in the class of interest; however, we might want to sort objects in a way different from the standard specifications for the *compareTo* method. In these cases we need a **different** way. That alternative way is provided with the *Comparator* interface.

The *Comparator* interface also has only **one** method:

```
public interface Comparator
{
    int compare(Object firstObject, Object secondObject);
        // Returns a neg number if firstObject < secondObject;
        // Returns a pos number if firstObject > secondObject;
        // Returns 0 if firstObject = secondObject.
}
```

This *Comparator* interface is generally used to declare a *Comparator* object (let's call it *comp*) that could then be used to sort using either:

1. Arrays.sort(a, comp); //Sorts the a[] array of **objects** in ascending order. This
   //method is overloaded. There is also a single parameter
   //version presented in Lesson 19
2. Collections.sort(al, comp); //uses a merge sort and sorts the **ArrayList** al

***Comparator* example:**

As an example let's use a *BankAccount* class again, but this time **without** implementing any interface.

```
public class BankAccount
{
    public BankAccount(double bal)
    {
        balance = bal;
```

```
        }
        . . . other methods . . .
        public double balance;
}
```

We will now need to create a *BankAccount* comparator class; let's call it *BA_comparator*. Notice this one **does** implement the *Comparator* interface.

```
import java.util.*; //necessary for Comparator interface
public class BA_comparator implements Comparator
{
        public int compare(Object firstObject, Object secondObject)
        {
                BankAccount ba1 = (BankAccount) firstObject;
                BankAccount ba2 = (BankAccount) secondObject;
                int retValue;

                if (ba1.balance < ba2.balance)
                {
                        retValue = –1;
                }
                else
                {
                        if (ba1.balance > ba2.balance)
                        {
                                retValue = 1;
                        }
                        else
                        {
                                retValue = 0;
                        }
                }
                return retValue;
        }
}
```

Following is code for a *Tester* class in which we would sort an array of *BankAccount* objects:

```
//Create an array, BankAccount[ ]
BankAccount ba[ ] = new BankAccount[500];
ba[0] = new BankAccount(128);
ba[1] = new BankAccount(1200);
ba[2] = new BankAccount(621);
. . .
```

```
// Now create a comparator object using the BA_comparator class above.
Comparator comp = new BA_comparator( );

//Sort the array
Arrays.sort(ba, comp);
```

Sorting contents of a *List* object:

Similarly, to sort an *ArrayList* object (also works for *LinkedList* and *Vector* objects):

```
ArrayList recipeList = new ArrayList( );
…some code to add Recipe objects to the list…

// This assumes we have already written another class called
// RecipeComparator (in which we compare calories) that is similar to the
// BA_comparator class above.
Comparator comp = new RecipeComparator( );

//Now do the sort
Collections.sort(recipeList, comp); //Makes it possible for iterator to step through
//the list in the prescribed order.
```

This *ArrayList* can also be sorted using *Collections.sort(recipeList);* if the objects comprising the list implement the *Comparable* interface and have an appropriate *compareTo* method.

The most difficult objects to **compare** are images. For an enrichment activity, see Appendix U in which an activity is described that involves scanning a printed document and then applying OCR (optical character recognition) software.

Adapting to either *Comparable* or *Comparator*

There are occasions in which we wish create a class where there is a need to compare two objects and we have no knowledge ahead of time of whether the objects to be compared are *Comparable* or if a comparator is provided. The desire is to make our class as general as possible so as to adapt to **either** possibility. Here is how to do it:

• In the constructor for the class, receive as a parameter a *Comparator* object. A *null* may possibly be passed for this parameter in case the objects used in the class are *Comparable* rather than having a comparator. Initialize a state variable with this *Comparator* object as shown below (even if a null is passed):

```
public YourClass(…other parameters as needed…, Comparator cp)
//Constructor
{
        …
        cmptr = cp;
        …
```

```
}
//State variables
Comparator cmptr;
…
```

• Create a method in which the comparison will be done. Assuming that the objects
to be compared are of the *Object* type, the method will receive two parameters, *obj1* and *obj2*, both of *Object type*. The method will return an integer that is:

- less than 0 if *obj1* < *obj2*
- greater than 0 if *obj1* > *obj2*
- 0 if *obj1* equals *obj2*

It is assumed that the *cmptr* object adheres to these same rules if it is not *null*. The method shown below implements all these ideas.

```
private int compareObjects(Object obj1, Object obj2)
{
        if(cmptr = = null)
                { return ((Comparable)obj1).compareTo(obj2); }
        else
                { return cmptr.compare(obj1, obj2); }
}
```

• Call the *compareObjects* methods when a comparison of two objects is needed.