Comparing Objects

Often we want to sort objects. As examples, we might want to order student record objects by student numbers while we might want to order objects representing shapes by size.

To sort objects, we can start by writing a sort that is applicable to any objects that we might consider to be sortable. Let us designate such objects as being of type Sortable. Of course, if we are going to sort these objects, we must decide what it means for one object of type Sortable to "precede" another object of the same type. Suppose, for now, that we have done this by writing an instance method for Sortable objects with the following signature

public boolean precedes (Sortable other)

The method precedes returns true if and only if its implicit object precedes other. Once we have a method that defines criteria for ordering Sortable objects, we can then write a sort for such objects.

Example 1

A selection sort for Sortable objects could have the form

```
public static void selectSort (Sortable[] list)
{
  for (int top = list.length - 1; top> 0; top--)
  {
    int largeLoc = 0;
  for (int i = 1; i <= top; i++)
    if (list [largeLoc] .precedes(list[i]))
      largeLoc = i;
  Sortable temp = list [top] ;
  list[top] = list [largeLoc] ;
  list[largeLoc] = temp;
  }

}
```

Our goal is to make this method applicable to a wide variety of objects, unrelated except for the fact that each of them has some criterion for ordering the objects. To do so we must do two things:

• Specify the requirements that a class must satisfy if its objects are to be considered sortable. There is only one such requirement: any Sortable class must have a precedes method.

• Show how each class that we want to be considered Sortable should implement the precedes method.

For the first part, we use a structure that we have not yet seen in Java - an interface. Interfaces are very similar to the classes that we have been using; they can contain both fields and methods. The definition of the Sortable interface might take the following form.

```
interface Sortable
{
        public boolean precedes (Sortable other)
        ;
}
```

Notice in the definition the word interface instead of class. Notice also that the method of the interface has no body. This is characteristic of all interfaces; their methods never have bodies. All that the definition does is, in a sense, layout the framework for what would be required of any class that wants to be considered to be of type Sortable.

For the second part, for each class that we want to be designated as Sortable, we must provide a precedes method with the signature given in the interface. This method must define how to determine the ordering of objects of that type. The next example illustrates how we can do this.

Example 2

Suppose that we want to be able to sort objects of the type Fraction that we used when we introduced the idea of an object. Recall that the Fraction class had two fields, num and den, representing the numerator and denominator of a fraction. There is a natural way to decide if one Fraction object precedes another: we simply examine the values of the fractions that correspond to the Fraction objects. To implement this, we rewrite the Fraction class as follows.

```
class Fraction implements Sortable
{
  private int num;
  private int den;


  public boolean precedes (Sortable s)
  {
    Fraction f = (Fraction)s;
    double thisValue = (double)num/den;
    double otherValue = (double)f.num/f.den;
    if (thisValue < otherValue)
      return true;
    else
      return false;
  }
  // other methods of the class
}
```

Notice the phrase implements Sortable in the first line. This tells Java that there is a method in the Fraction class that provides a body for the method precedes in the Sortable interface.

Notice also that the type of the parameter in the precedes method is Sortable. The Sortable interface specifies that any precedes method must accept a parameter that is of type Sortable. Within the method, we cast the parameter to an appropriate type of object (Fraction in this case) with the statement

```
Fraction f = (Fraction)s;
```

If we were writing a precedes method for a Widget class, we would have to cast the Sortable parameter to an object of type Widget.

With this revised definition of the Fraction class, we can now use the selectSort method of Example 1 to sort an array of Fraction objects because Fraction objects can also be considered as Sortable objects. If our program contained other classes that implemented Sortable, then arrays of objects of these classes could also be sorted using exactly the same method.

This idea is such a good one that a form of it is actually implemented in Java. In the java.lang package, there is an interface called Comparable that contains the header of a single method called compareTo with the signature

public int compareTo (Object 0)

Any class implementing the Comparable interface must supply a compareTo method that accepts some kind of Object as a parameter and returns an int value as specified in the following table

Relationship Between Objects	Value Returned
implicit object precedes explicit object	some negative integer
implicit object equals explicit object	zero
implicit object follows explicit object	some positive integer

We have already encountered the compareTo method in our study of strings. The header of Java's String class contains the phrase implements Comparable and the class contains a compareTo method for String objects.

Example 3

An insertion sort that could be used for any objects that implement the Comparable interface could take the following form.

```
public static void insert Sort (Comparable[] list)
{
  for (int top = 1; top < list.length; top++)
  {
    Comparable item = list [top] ;
    int i = top;
    while (i > 0 && item.compareTo(list[i-1]) < 0)
    {
      list[i] = list[i-1];
      i--;
    }
    list [i] item;
  }

}
```

Using the Comparable interface, we have a technique that allows us to use a single sorting method to sort any type of object that we create but it cannot sort lists containing primitive types such as int or double. To include these, Java also offers a number of wrapper classes that are used, as their name suggests, to enclose a primitive type in an object. The identifiers of the wrapper types are usually the same as the corresponding primitive types but written with an initial upper case letter. For example, the wrapper class for double values is Doub'le while the wrapper class for byte is Byte. The only exceptions to this are the wrapper classes for int and char whose identifiers are Integer and Character. All the wrapper classes (except Boolean) contain appropriate compareTo methods.

If we wanted to apply this sort to an array of Fraction objects, all that we would have to do is add a compareTo method to our Fraction class and insert the phrase implements Comparable in the header of the Fraction class.

Example 4

The following code shows what must be added to the Fraction class to make it possible to sort Fraction objects using the sort in the preceding example.

```
class Fraction implements Comparable
{
  private int num;
  private int den;

  public int compareTo (Object 0)
  {
    Fraction other = (Fraction)o;
    double thisValue = (double)num/den;
    double otherValue = (double)other.num/other.den;
    if (thisValue < otherValue)
      return -1;
    else if (thisValue == otherValue)
      return 0;
    else
      return 1;
  }
//other methods of the class

}
```

Since the parameter of the compareTo method in the Comparable interface is Object, any compareTo method that we write must have a parameter of that type. Within the method, we cast the parameter to a reference to a Fraction object.

Example 5

The following program first creates three arrays: an array of Fraction objects, an array of Integer objects used as wrappers for int values, and an array of String values. After printing the contents of the original lists, it then uses a single sort method to sort all three arrays of objects and then prints the sorted results. The program assumes that the Fraction class contains the phrase implements Comparable in its header and that it contains definitions of the following methods: a compareTo method as required by the Comparable interface, a constructor that accepts a numerator and denominator as arguments, and a toString method that returns a string of the form "a/b".

Notice the use (in the println statements) of the method intValue. This method is used to recover int values from objects in the Integer wrapper class.

```
public class SortDemo
{
  public static void main (String[] args)
  {
    Fraction[] f = new Fraction [3] ;
    f[O] new Fraction(3,4);
    f[1] new Fraction(1,2);
    f[2] new Fraction(2,3);
```

```
      Integer[] n = new Integer[3];
      n[O] new Integer(7);
      n[1] = new Integer(5);
      n[2] = new Integer(2);

      String [] s = {"Jonathan","Callum","Melissa"};

    System.out.println("Original Values\n");
    for (int i = 0; i < f.length; i++)
      System.out.println (f [L] + II II + n [i].intValue 0   + II II + s [i]) ;

    selectSort(f);
    selectSort(n);
    selectSort(s);

    System.out.println("\nSorted Values\n");
    for (int i = 0; i < f.length; i++)
      System.out.println (f [i] + II II + n [i].intValue 0    + II II + s [i]) ;
    }

    public static void selectSort (Comparable[] list)
    {
      for (int-top = list.length - 1; top> 0; top--)
      {
        int largeLoc = 0;
        for (int i = 1; i <= top; i++)
          if (list[largeLoc] .compareTo(list[i]) < 0)
          largeLoc = i;

        Comparable temp = list [top];
        list[top] = list [largeLoc] ;
        list[largeLoc] = temp;
      }
    }
}
```

The output from this program would be

```
Original Values
3/4        7        Jonathan
1/2        5        Callum
2/3        2        Melissa

Sorted Values
1/2        2        Callum
2/3        5        Jonathan
3/4        7        Melissa
```

In Example 5, the compareTo method operates differently depending on the type of object associated with it. The elements of the list array can be of type String, Fraction, or Integer. At execution time, Java determines what type of object is involved in the call and picks the ap- propriate form of the compareTo method. This extremely useful feature is an example of polymorphism, a word meaning "occurring in many forms" . Java's interfaces allow us to introduce polymorphism very easily. To invoke polymorphism, an object can be associated with a method as an implicit parameter, an explicit parameter, or both (as in the case of compareTo).