6.11 Laboratory: Sorting with Comparators

Objective. To gain experience with Java’s `java.util.Comparator` interface.

Discussion. In Chapter 6, we have seen a number of different sorting techniques. Each of the techniques demonstrated was based on the fixed, natural ordering of values found in an array. In this lab, we will modify the `Vector` class so that it provides a method, `sort`, that can be used—with the help of a `Comparator`—to order the elements of the `Vector` in any of a number of different ways.

Procedure. Develop an extension of `structure.Vector`, called `MyVector`, that includes a new method, `sort`.

Here are some steps toward implementing this new class:

1. Create a new class, `MyVector`, which is declared to be an extension of the `structure.Vector` class. You should write a default constructor for this class that simply calls `super()`. This will force the `structure.Vector` constructor to be called. This, in turn, will initialize the protected fields of the `Vector` class.

2. Construct a new `Vector` method called `sort`. It should have the following declaration:

   ```java
   public void sort(Comparator<T> c)
   // pre: c is a valid comparator
   // post: sorts this vector in order determined by c
   ```

   This method uses a `Comparator` type object to actually perform a sort of the values in `MyVector`. You may use any sort that you like.

3. Write an application that reads in a data file with several fields, and, depending on the `Comparator` used, sorts and prints the data in different orders.

Thought Questions. Consider the following questions as you complete the lab:

1. Suppose we write the following `Comparator`:

   ```java
   import structure5.*;
   import java.util.Iterator;
   import java.util.Comparator;
   import java.util.Scanner;
   public class RevComparator<T> implements Comparator<T>
   {
       protected Comparator<T> base;

       public RevComparator(Comparator<T> baseCompare)
       {
           base = baseCompare;
   ```
```java
public int compare(T a, T b)
{
    return -base.compare(a,b);
}
```

What happens when we construct:

```java
MyVector<Integer> v = new MyVector<Integer>();
Scanner s = new Scanner(System.in);

while (s.hasNextInt())
{
    v.add(s.nextInt());
}

Comparator<Integer> c = new RevComparator<Integer>(new IntegerComparator());
v.sort(c);
```

2. In our examples, here, a new Comparator is necessary for each sorting order. How might it be possible to add state information (protected data) to the Comparator to allow it to sort in a variety of different ways? One might imagine, for example, a method called ascending that sets the Comparator to sort into increasing order. The descending method would set the Comparator to sort in reverse order.

Notes: