Comparators and Ordering

Introduction

Comparing objects comes naturally to us. When we consider some real numbers, we can order them easily through a set of rules; we can take a list of words and arrange them alphabetically, through a similar set of rules. It becomes more interesting, however, when we consider tangible objects, which have many properties. How would one apple be compared to another apple? By taste? Color? Mass? On a different scale, how would different fruit be compared to one another? Perhaps based on a function of how much we enjoy a particular kind of fruit, how fresh the fruit is.

In the world of digital logic, we can capture the steps used to compare things, so that they can be used repeatedly, at much faster speeds than humanly possible (Perhaps we would like to model reality to predict events or sort sets of solutions to optimize for specific criteria). For Java, TPTB have built in an easy-to-use set of interfaces for use with sorting methods that have already been implemented for us. The interface that we will explore today is java.util.Comparator, which defines a pattern for a custom class that provides a method to compare two arbitrary objects.

Getting Started

Grab the code for the project lab3.tar.gz and ungzip + untar the directory ("tar -xvzf lab3.ta" on athena). If you're using eclipse, now's the time to copy the src directory (the one just unzipped) into a project of your choice (make sure to designate the imported folder a source folder so that you can run/compile files from it).

Understanding the existing code

There are 6 source files, however, most of them have been written for you.

AntiAliasJButton.java - You can safely ignore this, it extends JButton (part of the Swing, one of Java's Graphical User Interface (GUI) libraries) to make the text look nice and anti-aliased (no jaggies).

AntiAliasJLabel.java - See above.

ComparatorExample.java - The GUI, for the address-book-like program that allows you to add people to it and delete people from it. It lets you then edit the values for anyone in the table. String values are editable by clicking on the cell and using a keyboard, while Colors are editable by clicking on them. This is the class to run in order to start the GUI. (It contains the main method).

ExtendedTableRenderer.java - A GUI-related class borrowed and adapted from a popular developer help site that renders the JTable with the Colored button instead of a color name or color values. It also passes through mouse actions so that the underlying button will respond to mouse clicks.

PeopleComparator.java - Your job. This class needs to take two objects (they are specifically passed Person objects, but you will need to check to make sure and explicitly cast to Person). Depending on the fields specified when PeopleComparator is constructed, either compare naturally the String fields, or compare the amount of red in the favorite color of the person.

Person.java - The Person data type, Person contains data that determines what fields will be drawn by

ComparatorExample's PersonTableModel. You can extend Person (or modify it) to support more attributes quite easily by following the patterns of variables/constants used to store the data, as well as adding additionally cases in the various field-modification and access methods. This is, however, not necessary for the assignment.

Your Mission: Writing the Comparator

Modify the skeleton PeopleComparator.java using constants and methods from the other classes to implement a Comparator that will allow the two sorting comboBoxes and the "sort" button to sort the list of people. (You can test this Comparator with the GUI). Note that you can do this with very little actual code and that it is not necessary to modify any of the other classes to get this to work.

In `PeopleComparitor.java`, there are only two things you need to write — the constructor and the `compare` method. You'll need the arguments passed into the constructor when you write `compare`, so it's probably a good idea to save the constructor's arguments away in private fields in the object.

The `compare` method is slightly more complicated. It must follow the `Comparator` specification. The upshot is that `compare` takes in two objects (`o1` and `o2) and returns a negative number if o1 is "less" than o2, 0 if they are equal, and a positive number if o1 is "greater" than o2.`

The code you write will deal only with `People` objects. First, you'll need to cast the Objects `o1` and `o2` into `Person` objects. Then you'll need to extract the field you're sorting by from the `Person` object. Finally, you'll need to figure out how the `Person` objects compare.

Optional: Add a Field

Following the patterns for fields in Person.java, add an "age" field that takes integers and modify your PeopleComparator so that an age comparison is possible.

Optional: Explore Swing

6.170 Problem sets and real life will eventually require you to enable human-computer interaction through the use of a pretty, graphical interface. The GUI written for this lab was in Swing, a generally platform-independent graphics library that lets you open windows, paint images, add buttons, receive input from the keyboard or mouse, etc. A good place to get started is the Swing Tutorial, and several of the commonly used components are in ComparatorExample.java. Get a feel for what the components are and how're they're used. (This visual index is rather helpful.) Try adding a few borders or adding a few buttons and adding a mouseListener (following the patterns of the sort, addPerson, RemovePerson buttons) to change the color of the button when it 'sees' different mouse actions (see MouseAdapater).

Deliverables

PeopleComparator.java and any other sources files that were created/modified.

Footnotes

Another popular interface for comparisons is java.lang.Comparable. It is implemented within a class to give it a natural ordering (the compareTo method takes another object and returns a value that gives the relation of the two objects).

Last Revision: 2006.01.25