Handout #5
Nick Parlante

HW1 DBTable

Our first homework will build a Swing frame which can display and edit a flat file database. This will homework will give you experience with the key parts of Swing apps: components, built-ins like JTable and JList, and the MVC/listener structure. This homework is due 11:59 pm. April 25th. See the MVC example from lecture. See also the table and MVC examples on the Sun site. For an additional example, see the MVC Table example in the CS108 handout directory.

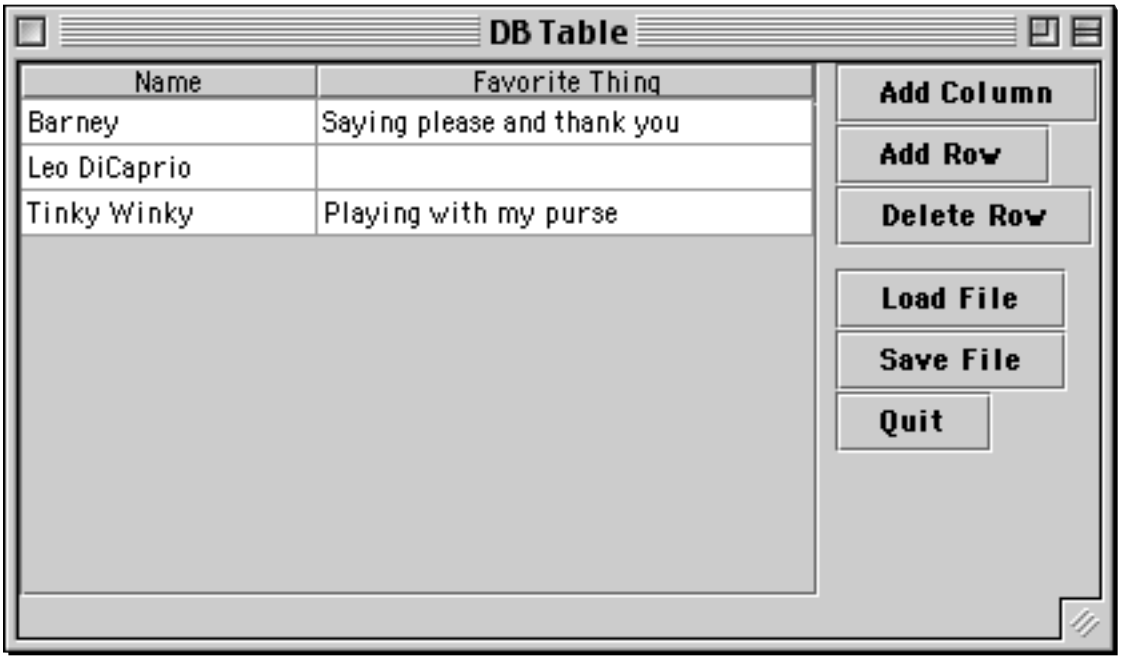
Running Java

Our default Java environment will be the Java 1.2 installation in the CS108 directory on the elaines. As a practical matter, you can build your code in any 1.2 environment, or in a 1.1 environment with the swingall.jar and collections.jar libraries added in. You should test your code a little on an elaine before you hand it in, but as a practical matter, compatibility problems between the various Java versions are rare.

The CS108 web page (http://www.stanford.edu/class/cs108/) has information on the various ways you can set up a Java environment, IDE, debugger, etc.

Part 1 — Basic DB Table

The first thing to do is build some table code that manages a flat-file data like this...



Build the frame with a Border layout. The JTable of interest is in the CENTER, and the buttons are vertically aligned in a container in the EAST. We'll be using the standard notion of "flat file database"...

- The data has an overall number of columns defined by the number of labels at the top — the two labels "Name" and "Favorite Thing" in the example above. In a new empty document, the initial number of columns is zero.

- Each row defines a number of elements fill in the column slots left to right. A row will not have more columns than the overall database, but it may have fewer. In the example above, the Leo DiCaprio row only has one element — he does not have a Favorite Thing yet.

The operations on the database are...

- Add Column Prompt the user for the label of the new column (see JOptionPane), and add the new column on the right hand side. Ideally, the user would be able to edit the column labels later, but we're not going to support that.

- Add Row Add a new, empty row at the bottom of the table, and select the new row. If there are zero columns, the add row operation should silently do nothing. (How would the view visually communicate that there is new row with zero columns? — let's just disallow it.)

- Delete Row Delete the selected row. If there is no selected row, silently do nothing.

- Load File Set the table model to show data loaded from a text file. The old contents of the table are lost (a better implementation would have a dirty bit etc. to do the right thing, but we won't bother with that). The file format will be that every row is one line of text in the file with the elements separated by tabs. The first line must be present and contains the labels. Use a JFileChooser(".") object to bring up a file chooser on the current directory. (There are implementation hints below for the boring parts of the file reading and writing.)

- Save File Write the whole table out in the format above — the labels on the first line with the rest of the rows following. Each row is one line. The elements within a row are separated by tabs.

Basic Use

Initially the table is empty. Select Add Column a couple times to make some columns. Select Add Row to make a couple rows. Use the mouse, the tab and arrow keys, to navigate and select around the table cells and type in some data. Select Add Column again and add some data for the new column for one of the rows.

DBFrame

Create a DBFrame subclass of JFrame. Its main() should create one instance of DBFrame. If there is a command line argument, main() should load the frame with that file. DBFrame constructor should build all the buttons and set up the listeners. The code is a bit long and repetitious, but that's just the way it is with current Java technology. Someday, GUI Java Beans will provide a better solution for this sort of setup.

JTable

Install a JTable in a JScrollPane. Give the scroll pane a preferred size of 300 x 200. Set the auto resize mode on the JTable so the columns auto-resize their pixel width.

Data Model

The actual data storage and logic is in the table data model. Create a DBTableModel subclass of AbstractTableModel to act as the data model for the JTable (you could subclass of DefaultTableModel, but I think it works out better if you use AbstractTableModel). Read up on TableModel and AbstractTableModel (they are like the JList examples from lecture). The data model needs to store the column labels and all the rows. Store the labels in their own ArrayList. Store the main data as an ArrayList of ArrayLists. Rows do not need to be full width. For example, the "Leo DiCaprio" row could be length-one until his Favorite Activity gets put in.

Fundamentally, all the operations described above, such as Add Column and Add Row have the following structure: (1) perform the change on the data model data structures, (2) do the appropriate fired event to notify the view, (3) on it's own eventually, the view will interrogate the data model to figure out the new state.

The initial JTable column-pixel-width scheme is lame — the user will need to manually resize them a little at first (I think this is a Swing bug).

Swing Style

To emphasize the correct passive role of your code vs. "The System" your solution should not contain any calls to the following methods: setSize(), repaint(), or revalidate() (there will be one call to revalidate() in part 3). These calls would be necessary for some Swing code, but the MVC in this assignment happens to take over 100% of the sizing and drawing logic, so your code does not need to. If you think you need to call one of those methods, ask a staffer first.

Parsing Code

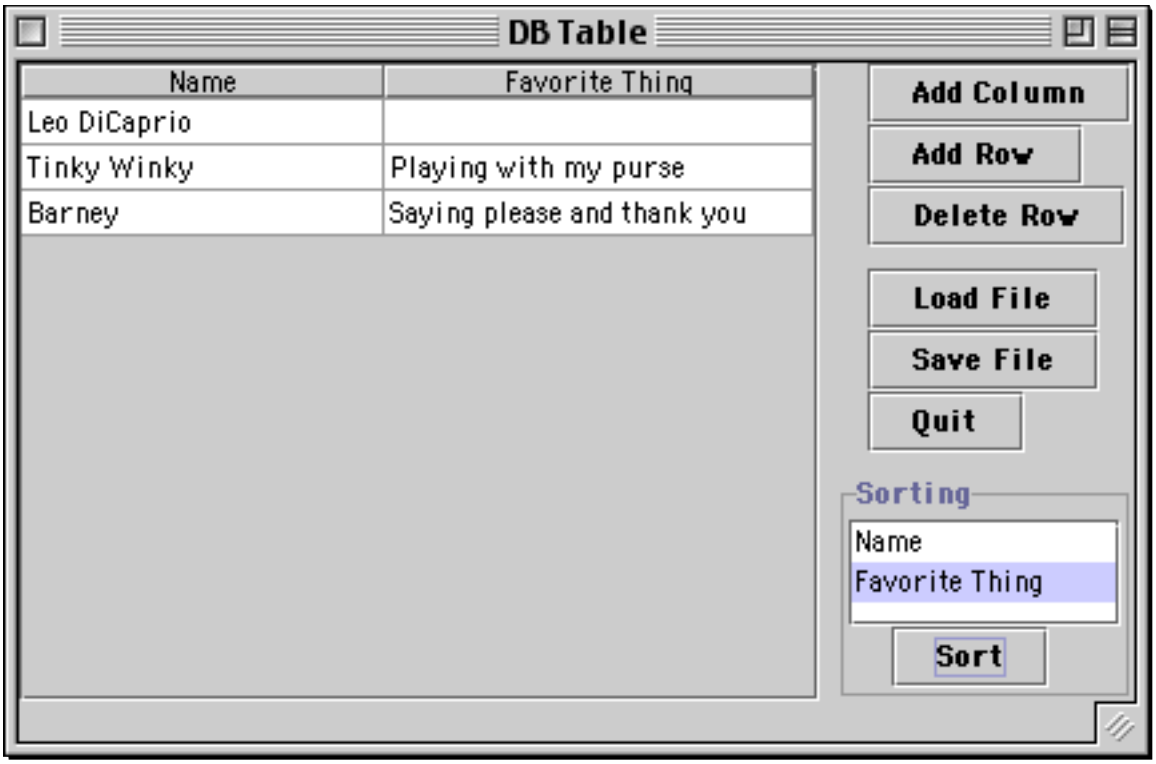
Parsing and I/O are not all that interesting. Here's some routine tab-array code you can use (available in the class directory)...

```
/*
 Contains some static utility methods for
 converting between an ArrayList and
 tab-delimited text format.
*/
```

```
class DBUtils {
   /*
    Writing Utility.
    Given an ArrayList representing a row, compute the
    String of tab-delimited text.
   */
   public static String arrayToString(ArrayList row) {
      int i;
      int len = row.size();
      StringBuffer result = new StringBuffer();

      for (i=0; i<len; i++) {
         if (i != 0) result.append('\t');
         String elem = (String) row.get(i);
         if (elem!=null) {
            result.append(elem);
         }
      }

      return(result.toString());
   }


   /*
    Reading Utility.
    Given a tab-delimited text line, compute the ArrayList
    of its elements.
    By default, the StringTokenizer wants to lump adjacent \t's
    together to count as one delimeter, so we must have it return
    every token and put the array together carefully.
    At least the docs fo StringTokenizer are decent compared to
    StreamTokenizer.
   */
   public static ArrayList stringToArray(String string) {
      StringTokenizer tokenizer = new StringTokenizer(string, "\t", true);
      ArrayList row = new ArrayList();
      String elem = null;

      while(tokenizer.hasMoreTokens()) {
         elem = tokenizer.nextToken();

         // If it's the tab, then the previous elem was empty
         if (elem.equals("\t")) row.add(null);
         else {   // otherwise it's the data
            row.add(elem);

            // discard the following tab
            if (tokenizer.hasMoreTokens()) tokenizer.nextToken();
         }
      }

      return(row);
   }
}
```

Part 2

For part 2, the goal is to add a little JList in the controls area that lists the column labels. Clicking a label in the list just selects it. Clicking the "Sort" button sorts the data in the database using a string compare of the elements in the given column.



An empty element should sort as if it were the empty string "". The HI design of this arrangement is not great. For example, it's in wrong looking state when the user selects a different column, but has not yet clicked the sort button. For good HI, the appearance should always show the current state. Using a JMenu or a JComboBox might yield a better HI. However, for purposes of experimenting the MVC, JList is the best fit, so that's what we're doing.

Implementation 2

The solution strategy will be to create a subclass of AbstractListModel to act as the data model for the list. The subclass should not store any list elements. Instead, it should have a pointer to a TableModel. It should use the labels from the table model as the elements to present as its list model. Data requests sent to the list model get forwarded behind the scenes to the real data model. The list model should listen to the table model in case there is a change in the labels (We'll use a similar strategy for part 3.)

Create a little JPanel to contain the sorting controls. Use setLayout() to give it a vertical Box layout. Use setBorder() to give it the little titled border (see BorderFactory). Put the JList in a scroller, and give the scroller a small preferred size, such as 30 x 40.

Most of the table model changes — adding a row, editing an element — do not affect the list of labels. It's acceptable to detect a change that does affect the labels by checking if the first row changed in the TableModelEvent is the constant TableModelEvent.HEADER_ROW. In that case, it's acceptable to use fireContentsChanged() to signal that the whole list has changed.

Add a method to the table model so it can be asked to sort itself. You'll need to construct a custom inner subclass of Comparator to sort by the desired column.

Part 3 — Filter Table

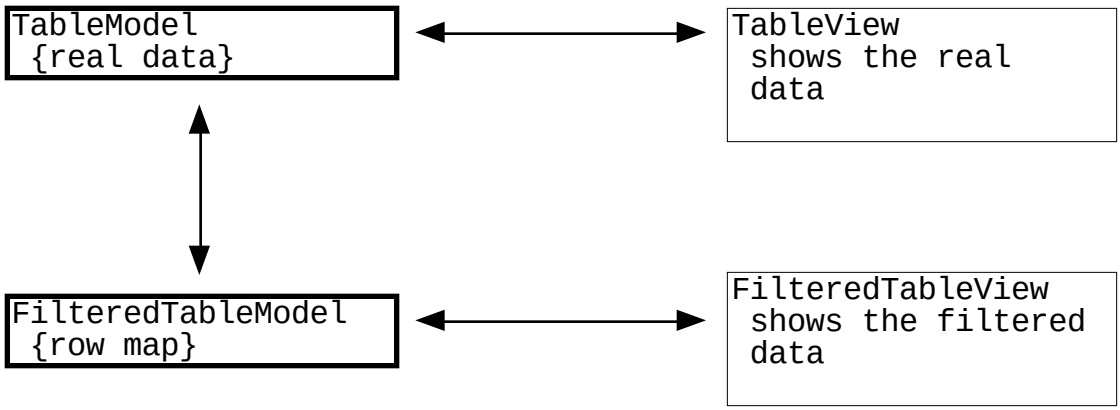
For part 3, you will add a second "filter" table in the South of the frame that shows a subset of the rows. The filter table shows the subset of the rows that contain the string in the filter text field. In this case, the string "ird" has been typed into the text field, so the filter table is showing the rows for "The Third Chimpanzee" and "To Kill a Mockingbird".

There are four elements in the filter interface which should be arranged in a vertical box in the SOUTH border of the frame — a checkbox which turns the feature on and off (initially off), a label which identifies the text field, a text field which contains the filter string, and lastly a scrollpane / table which shows the filtered rows. When the checkbox is checked, the text field should be enabled and the scrollpane / table should be added. When the checkbox is cleared, the scrollpane / table should be removed, and the text field should be disabled.

The relationships between the main table, the text field, and the filter table are all "live". Each keystroke in the text field should change the rows shown in the filter table instantly. Research JTextField to see how to listen for changes in its text.

FilteredTableModel Implementation

Create a FilteredTableModel subclass of AbstractTableModel which is the model for a table in the usual way, except it does not store any table data. FilteredTableModel looks like a regular model, but instead of storing data, it keeps a pointer to a "real" TableModel containing the data and a JTextField that contains the current filter string. The FilteredTableModel represents the subset of the rows in the real data model where one or more columns contain the filter string. The FilteredTableModel can implement this by computing a "row mapping" ArrayList that stores the indices of the rows in the real model that contain the filter string. Filtered row 0 maps to the first row in the real model that contains the filter string. Filtered row 1 maps to the second row in the real model contains the filter string, and so on. In the above example, the row map begins {2, 8, ...} since real row 2 contains "vent" as does real row 8.



The FilteredDataModel presents a subset of the real rows to the filtered view. The FilteredDataModel does not store any data, behind the scenes it gets it from the real table model. In the drawing, the arrows go in both directions —data flows one way while requests flow the other.

Using its row mapping, the FilteredDataModel can forward the getDataAt() and setDataAt() messages to the real data model. The FilteredTableModel should not store any table data itself — everything to do with real data is forwarded to the real model.

As every product of ResEd knows, the FilteredTableModel also needs to be a good listener. It needs to listen for changes in the real table model and in the

filter string, either of which could change the row mapping. In both cases, the FilteredTableModel needs to re-compute its row mapping.

- Use the standard TableModel messages such as getDataAt() and setDataAt(), for data exchange with the real data model and use the standard TableModelListener interface to listen to it. In this way, the FilteredTableModel can be layered on top of any object that implements TableModel.

- The FilteredTableModel could be very clever when responding to changes in the real data model. For example, if row 8 in the real data model is changed, then the FilteredTableModel could be clever about doing the minimum survey of the real data model to figure the new row mapping. For this assignment, we are officially not bothering with such clever optimizations. If there is a change to the real model or the filter string, we will just re-compute the whole row-mapping. This seems to work fine in practice. Changes to the real model are not that frequent when you are using the filter feature.

- Most changes to the filter table model are large — an additional character is typed in the filter text field and half the rows disappear. For this reason, we won't worry about doing fine-grained optimized notifications for small changes. The FilteredTableModel can divide change notifications for its own listeners into two categories: (a) changes to the rows or their elements — fireTableDataChanged(). (b) changes to the number of columns — fireTableStructureChanged(). It's worthwhile to keep these separate, because fireTableStructureChanged() essentially causes the table to be built from scratch from the model which causes the table to forget state such as the column widths. It's irritating if every time you edit a cell, the filtered columns revert to their default width.

Miscellaneous Hints

- A row should appear in the filter table if any of its elements contain the filter string anywhere using case-insensitive string search (see String for help). A string in a row which is null or the empty string is considered to not contain the filter string.

- When the filter button adds and removes the table / scrollpane, it needs to explicitly force a re-layout of the container (since the container's population has changed). Add() and remove() do not get this right automatically. Therefore, your code will need to call revalidate() or container.pack() to re-layout the container. Revalidate() will adjust everything within the current window size. Alternately, pack() will adjust the window size to fit the new contents. Try them both to see how they work. The revalidate() solution is probably better so that's the official solution, although having the button jump around is a bit disturbing.

- It's a reasonable design for the FilteredTableModel to take its real model and text field as arguments to its constructor. It's hard for FilteredTableModel to do anything with out those two.

The FilteredTableModel is a nice example of programming within an OOP system. The class introduces its own feature, in this case filtering. But the new feature is fit in to the built-in structures in such a way that the built-ins do most of the work.

Once you have a basic solution working, try it on the following stress cases..

- Have at least a few rows in the main table. Search for the string "foo". Now add a new column. Now add the string "foo" in the new column in a row which did not contain foo. When you hit return, the new row should show up in the filter table.

- Now in the filter table, double click the "foo" and edit it to "oo". When you hit return, the row should remove itself from the filter table, and the change should automatically be reflected back in the main table.

When you get this all working, it demonstrates classic OOP-library behavior. There is real effort to fit your code into the library structure. However, once you fit in, all the weird cases tend to automatically work, since the library structure has a lot of design and testing in it for all cases.

Swing Bug Workaround Policy

Our official course policy is that we will not go out of our way to write code to work around Swing bugs or omissions. We will try to write code in the most straightforward correct way. Ideally, that code will work the best in time as Swing is improved and debugged.

Swing bugs you may observe on this assignment that you are not responsible for....

- Table cell focus — it's possible to do an edit in a table cell and then click in a separate text field causing the table cell to loose focus and forget the edit. You may need to hit return or tab to explicitly "synch" the edit into the cell.

- The blinking cursor does not necessarily appear when it should when you have typing focus into a table cell. Double click should explicitly begin a cell edit.

- Even if you set table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS); columns in the table do not necessarily adjust their width to fill the table right away. Often they wait until a resize situation, and then they do.

Deliverables

We will put your solution through its paces — make sure you have tested it thoroughly. It should never crash or throw an exception.

Your submission directory should contain...

1) All your .java files to build your solution. We will build with...

```
% javac *.java
```

2) Put your main() in a class called xxxxx. Once we have run javac on your .java files, we shold be able to run your program from the command line with...

```
% java xxxxxx
```

3) Have a Readme text file in the directory which contains on the first line, your name last, first. On the second line, your email address. And in the balance of the file, you may put any notes for the grader.

```
Gruber, Hans
hans@cs

I just can't tell you how much I enjoyed that.
Really, I can't.
```

4) Remove any extraneous files (.class files, ~ files) from the directory.

5) Once the directory is ready, read the instructions in the directory /usr/class/cs193k/submit/ — they will direct you to run the submit program which will copy the contents of your homework directory up into submit space. You may submit multiple times, but we only grade the last one.