THE JTABLE CLASS

The JTable class is much like the JList class, in that you can program it very easily to do simple things. Similarly, in order to do sophisticated things, you need to create a class derived from the AbtractTableModel class to hold your data.

A Simple JTable Program

In the simplest program, you just create a rectangular array of objects and use it in the constructor for the Jtable. You can also include an array of strings to be used a column labels for the table.

```
public SimpleTable()
   {
      super("Simple table");
      JPanel jp = new JPanel();
      getContentPane().add(jp);
      Object[] [] musicData = {
         {"Tschaikovsky", "1812 Overture", new Boolean(true)},
         {"Stravinsky", "Le Sacre", new Boolean(true)},
         {"Lennon","Eleanor Rigby", new Boolean(false)},
         {"Wagner", "Gotterdammerung", new Boolean(true)}
      };
      String[] columnNames = {"Composer", "Title",
                              "Orchestral"};
      JTable table = new JTable(musicData, columnNames);
      JScrollPane sp = new JScrollPane(table);
      table.setPreferredScrollableViewportSize(
                     new Dimension(250,170));
      jp.add(sp);

      setSize(300,200);
      setVisible(true);
   }
```

This produces the simple table display below:

This table has all cells editable and displays all the cells using the *toString* method of each object. Of course, like the JList interface, this simple interface to JTable creates a data model object under the covers. In order to produce a more flexible display you need to create that data model yourself.

You can create a TableModel by extending the AbstractTableModel class. All of the methods have default vales and operations except the following 3 which you must provide:

```
public int getRowCount();
public int getColumnCount();
public Object getValueAt(int row, int column);
```

However, you can gain a good deal more control by adding a couple of other methods. You can use the method

```
public boolean isCellEditable(int row, int col)
```

to protect some cells from being edited. If you want to allow editing of some cells, you must provide the implementation for the method

```
public void setValueAt(Object obj, int row, int col)
```

Further, by adding a method which returns the data class of each object to be displayed, you can make use of some default cell formatting behavior. The JTable's default cell renderer displays

- Numbers as right-aligned labels
- ImageIcons as centered labels
- Booleans as checkboxes
- Objects using their *toString* method

You simply need to return the class of the objects in each column:

```
   public Class getColumnClass( int col)        {
      return getValueAt(0, col).getClass();
   }
```

Our complete table model class creates exactly the same array and table column captions as before and implements the methods we just mentioned:

```
class MusicModel extends AbstractTableModel
{
   String[] columnNames = {"Composer", "Title", "Orchestral"};

   Object[] [] musicData = {
   {"Tschaikovsky", "1812 Overture", new Boolean(true)},
   {"Stravinsky", "Le Sacre", new Boolean(true)},
   {"Lennon","Eleanor Rigby", new Boolean(false)},
   {"Wagner", "Gotterdammerung", new Boolean(true)}
   };

   int rowCount, columnCount;
   //------------------------------------------
   public MusicModel()    {
      rowCount = 4;
      columnCount =3;
   }
   //------------------------------------------
   public String getColumnName(int col)    {
      return columnNames[col];
   }
   //------------------------------------------
   public int getRowCount(){return rowCount;}
   public int getColumnCount(){return columnCount;}
   //------------------------------------------
   public Class getColumnClass( int col)    {
      return getValueAt(0, col).getClass();
   }
   //------------------------------------------
   public boolean isCellEditable(int row, int col)    {
      return (col > 1);
   }
   //------------------------------------------
   public void setValueAt(Object obj, int row, int col)    {
      musicData[row][col] = obj;
        fireTableCellUpdated(row, col);
   }
   //------------------------------------------
   public Object getValueAt(int row, int col)       {
         return musicData[row][col];
      }
```

The main program simply becomes:

```
public ModelTable()
{
   super("Simple table");
   JPanel jp = new JPanel();
   getContentPane().add(jp);
   JTable table = new JTable(new MusicModel());
   JScrollPane sp = new JScrollPane(table);
   table.setPreferredScrollableViewportSize(
                 new Dimension(250,170));
   jp.add(sp);

   setSize(300,200);
   setVisible(true);
}
```

As you can see in our revised program display, the boolean column is now rendered as check boxes. We have also only allowed editing of the right-most column y using the *isCellEditable* method to disallow it for columns 0 and 1.

As we noted in the JList section, the TableModel class is a class which holds and manipulates the data and notifies the Jtable whenever it changes. Thus, the Jtable is an Observer pattern, operating on the TableModel data.

Cell Renderers

Each cell in a table is rendered by a cell renderer. The default renderer is a JLabel, and it may be used for all the data in several columns. Thus, these cell renderers can be thought of as Flyweight pattern implementations. The JTable class chooses the renderer accoring to the object's type as we outlined above. However, you can change to a different

rendered, such as one that uses another color, or another visual interface quite easily.

Cell renderers are registered by type of data:

```
table.setDefaultRenderer(String.class, new ourRenderer());
```

and each renderer is passed the object, selected mode, row and column using the only required public method:

```
public Component getTableCellRendererComponent(JTable jt,
       Object value, boolean isSelected,
       boolean hasFocus, int row, int column)
```

One common way to implement a cell renderer is to extend the JLabel type and catch each rendering request within the renderer and return a properly configured JLabel object, usually the renderer itself. The renderer below displays cell (1,1) in boldface red type and the remaining cells in plain, black type:

```
public class ourRenderer extends JLabel
       implements TableCellRenderer
{
   Font bold, plain;
  public ourRenderer()   {
     super();
     setOpaque(true);
     setBackground(Color.white);
     bold = new Font("SansSerif", Font.BOLD, 12);
     plain = new Font("SansSerif", Font.PLAIN, 12);
     setFont(plain);
  }
//-------------------------------------------------------
  public Component getTableCellRendererComponent(JTable jt,
       Object value, boolean isSelected,
       boolean hasFocus, int row, int column)
        {
          setText((String)value);
          if(row ==1 && column==1)           {
             setFont(bold);
             setForeground(Color.red);
          }
          else           {
            setFont(plain);
            setForeground(Color.black);
          }
        return this;
       }
  }
```

The results of this rendering are shown below:

In the simple cell renderer shown above the renderer is itself a JLabel which returns a different font, but the same object, depending on the row and column. More complex renderers are also possible where one of several already-instantiated objects is returned, making the renderer a Component Factory.

THE JTREE CLASS

Much like the JTable and JList, the JTree class consists of a data model and an observer. One of the easiest ways to build up the tree you want to display is to create a root node and then add child notes to it and to each of them as needed. The *DefaultMutableTreeNode* class is provided as an implementation of the *TreeNode* interface.

You create the JTree with a root node as its argument

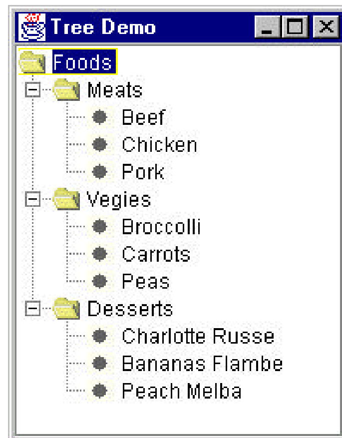
```
   root = new DefaultMutableTreeNode("Foods");
   JTree tree = new JTree(root);
```

and then add each node to the root, and additional nodes to those to any depth. The following simple program produces a food tree list by category:

```
public class TreeDemo extends JxFrame
{
   DefaultMutableTreeNode root;
   public TreeDemo()
   {
   super("Tree Demo");
   JPanel jp = new JPanel();   // create interior panel
   jp.setLayout(new BorderLayout());
   getContentPane().add(jp);

   //create scroll pane
   JScrollPane sp =  new JScrollPane();
   jp.add("Center", sp);

   //create root node
   root = new DefaultMutableTreeNode("Foods");
   JTree tree = new JTree(root);    //create tree
   sp.getViewport().add(tree);      //add to scroller

   //create 3 nodes, each with three sub nodes
   addNodes("Meats", "Beef", "Chicken", "Pork");
   addNodes("Vegies", "Broccolli", "Carrots", "Peas");
   addNodes("Desserts","Charlotte Russe",
              "Bananas Flambe","Peach Melba");

   setSize(200, 300);
   setVisible(true);
   }
   //-----------------------------------------
   private void addNodes(String b, String n1, String n2,
                                              String n3)
   {
```

```
        DefaultMutableTreeNode base =
              new DefaultMutableTreeNode(b);
        root.add(base);
        base.add(new DefaultMutableTreeNode(n1));
        base.add(new DefaultMutableTreeNode(n2));
        base.add(new DefaultMutableTreeNode(n3));
}
```

The tree it generates is shown below.



If you want to know if a user has clicked on a particular line of this tree, you can add a TreeSelectionListener and catch the *valueChanegd* event. The *TreePath* you can obtain from the *getPath* method of the TreeSelectionEvent is the complete path back to the top of the tree. However the *getLastPathComponent* method will return the string of the line the user actually selected. You will see that we use this method and display in the Composite pattern example.

```
public void valueChanged(TreeSelectionEvent evt)     {
       TreePath path = evt.getPath();
       String selectedTerm =
             path.getLastPathComponent().toString();
```

The TreeModel Interface

The simple tree we build above is based on adding a set of nodes to make up a tree. This is an implementation of the DefaultTreeModel class which handles this structure. However, there might well be many other sorts of data structure that you'd like to display using this tree display. To do so, you create a class of your own to hold these data which implements the TreeModel interface. This interface is very simple indeed, consisting only of

```
void addTreeModelListener(TreeModelListener l);
Object getChils(Object parent, int index);
int getChildCount(Object parent);
int getIndexOf Child(Object parent, Object child);
Object getRoot();
boolean isLeaf(Object);
void removeTreeModelListener(TreeModelListener l);
void value ForPathChanges(TreePath path, Object newValue);
```

Note that this general interface model does not specify anything about how you add new nodes, or add nodes to nodes. You can implement that in any way that is appropriate for your data.

Summary

In this brief chapter, we've touched on some of the more common JFC controls, and noted how frequently the design patterns we're discussing in this books are represented. Now, we can go on and use these Swing controls in our programs as we develop code for the rest of the patterns.