MVC Table

Thanks to Nick Parlante for much of this handout

GUI Model and View

How Does a GUI Work?

- You see pixels on screen representing your data.
- You click and make gestures, and this appears to edit the data.
- How does this really work?
- e.g. In MS word, if position the cursor to the right of some text and type an "m" -- what happens so that I see an "m" on screen? The change is not done to the screen/pixels directly, although that's what it looks like to my dad, who only knows about the pixels.

See: Model ->View -> Pixels

- How does the user get to see the pixels of the data?
- Have a "view" object -- say a `JPanel` or `JComponent` subclass
- The view holds a pointer to the "data model" -- ints, strings ... the actual data
- The "render" code looks at the model (int, String, ...) and renders that state out as pixels
  - e.g. `paintComponent()` in our `JComponent` looks at the model data and calls *g.drawXXX* to draw a representation of that data.

Edit: User Wants To Make a Change -> Model

- To make a change/edit ... the change first must go the model
- After the model change, we do something like `view.repaint()` to trigger a re-draw

Model

- In simple cases, the data model is just ivars in the view object. The view "owns" or "has" the data model. Changes happen to the data model, and then the view redraws. (e.g. the Dots example)
- The model can be a totally separate object from the view ... that's the full MVC pattern described below.

e.g. Dots

- The model in the Dots example is a list of `DotModel` (x, y, color) objects
- See: `paintComponent()` looks at all the x/y/color and draws circles
- Edit: click/drag in the view is translated to `setX`, etc. on the model, followed by `repaint()`

Model / View / Controller

- A decomposition strategy/pattern where "presentation" is separated from data storage
- "Model" -- data storage
- "View" -- display of data, aka "presentation"
- "Controller" -- change propagation. Typically a change is made on the model and then somehow the view is notified that the data has changed.
- Web Version
  - Shopping cart model is on the server -- the actual items you have
  - The view is the HTML in front of you

- Form submit = send transaction to the model, it computes the new state, sends you back a new view

Pluggable

- Since the model and view are nicely separated roles, can plug in different code for either.
- The model presents a standard interface, so any view can work with it.
- e.g. model is stock market data, can attach a graphing view, or a compute-statistics "view".

Modularity

- Writing larger programs in teams, we're always looking for a natural dividing line to help use separate our 1000-line program into two 500-line parts that are as independent as possible.
- Separating the "Data Model" and "View" ideas works well and is a very common modularity strategy.

Model -- aka Data Model

- "data model"
- Storage, not presentation
- Knows data, not pixels
- Support operations on the data (not its presentation)
  - cut/copy/paste, File Saving, undo, networked data -- these can be expressed just on the model which simplifies things
  - e.g. can get the logic for file save or undo working, without worrying about pixels.

Canonical Data

- Within a large system, you want to pick a single, "canonical" repository of the true data. Aka "the source of truth."
- In general, we try to avoid having multiple copies of things, but it's natural to have a few temporary copies of data passed around as parameters and whatnot during computations. Do not confuse the temporary copies with the canonical data.
- In MVC, the model contains the one, canonical version of the data.

View

- Presentation -- pixels
- Gets all data from model and draws or otherwise renders it for the user
- Does not store data itself (asks the model for data as needed)

Controller

- The controller is the logic that glues the model and the view together for data change.
- Manage the relationship between the model and view -- typically this involves sending updates around as the data changes.
- 1. Most data changes are initiated by user events (keyboard, mouse gestures) that tend to initiate on the view side. These are translated to setter/mutator messages to the model which does the actual data maintenance
- 2. When a change happens in the model data, the view needs to hear about it so it can draw differently if appropriate. In Swing, this is done with the Listener paradigm.

Model Role

- Store the canonical copy of the data
- Respond to getters methods to provide data

- Respond to setters to change data
- In Java, it is typical to use a "listener" strategy to tell the views about data changes.
  - Java uses the Model/Listener structure, and it's a good design, although there are other ways to do it.
  - Also known as the "observer/observable" pattern
- Model manages a list of listeners
- When receiving a *`setXXX`*`()` to change the data, the model makes the change and then notifies the listeners of the change (*`fireXXXChanged`*)
  - Iterate through the listeners and notify each about the change.
  - Change notification messages can include more specific information about the change (cell edited, row deleted, ...)

View Role

- Have a pointer to model
- Don't store any data
- Send *`getXXX`*`()` to model to get data as needed
- User edit operations (clicking, typing) in the UI map to *`setXXX`*`()` messages sent to model
- Register as a listener to the model and respond to change notifications
- On change notification, consider doing a *`model.getXXX()`* to get the new values to make the pixels up-to-date with the real data. Or perhaps nothing needs to be done, if for example that data is currently scrolled off screen or not shown.

Swing Table Classes

JTable -- View Component

- Has a pointer to a `TableModel` that does its storage
- Has all sorts of built-in features to display tabular data.

TableModel -- Interface

- The messages that define a table model -- the abstraction is a rectangular area of cells.
- `getValueAt()`, `setValueAt()`, `getRowCount()`, `getColumnCount()`, ...
- The table model establishes a co-ordinate system: 0..`getRowCount()-1`, 0..`getColumnCount()-1`. The model and the view(s) all use this model coordinate system to identify rows and columns.

TableModelListener -- Interface

- Defines the one method `tableChanged(TableModelEvent e)`
- The `TableModelEvent` object indicates specifically what sort of change it was -- row add, row delete, cell updated, ...
- If you want to listen to a `TableModel` to hear about its changes, implement this interface.

```
public interface TableModelListener extends java.util.EventListener
{
    /**
     * This fine grain notification tells listeners the exact range
     * of cells, rows, or columns that changed.
     */
    public void tableChanged(TableModelEvent e);
}
```

AbstractTableModel

- Implements some `TableModel` utility behavior.

- Provides helper utilities for things not directly concerned with storage
  - `addTableModelListener()`, `removeTableModelListener()`, ...
- *`fireXXXChanged()`* convenience methods
  - These iterate over the listeners and send the appropriate notification
  - `fireTableCellUpdated(row, col)  // this cell was changed`
  - `fireTableRowDeleted(row)  // this row index was deleted`
  - etc.
- `getRowCount()`, `getColumnCount()`, and `getValueAt()` are **abstract** -- they must be provided by a subclass table model that actually stores data.
- This is similar to the situation we had subclassing `ChunkList` off `AbstractCollection`.

DefaultTableModel

- A built-in Swing implementation of `TableModel`
- Extends `AbstractTableModel`
- Uses `Vector` for its implementation (`Vector` is what Java had before `ArrayList`), so it's a little old.

BasicTableModel Code Points

- A complete implementation of `TableModel` using `ArrayList`
- `getValueAt()`
  - Pulls data out of the `ArrayList` implementation
- setValueAt()
  - Changes the data model and uses *`fireTableXXX`* (below) to notify the listeners
- AbstractTableModel
  - Has routine code in it to manage listeners -- add and remove.
  - Has *`fireTableXXX()`* methods that notify the listeners -- `BasicTableModel` uses these to tell the listeners about changes.

1. Passive Example

- 1. Table `View` points to model
- 2. `View` does *`model.getXXX`* to get data to display

2. Edit Example / Two Views

- 1. Table `View1` points to model for its data and listens for changes
- 2. Table `View2` also points to the model and listens for changes
- 3. User clicks/edits data in `View1`
- 4. `View1` does a *`model.setXXX`* to make the change
- 5. Model does a `fireDataChanged()` -- notifies both listeners
- 6. Both views get the notification of change, update their display (*`getXXX`*) if necessary

`View2` can be smart if `View1` changes a row that `View2` is not currently scrolled to see -- in that case `View2` can compute that the changed row is scrolled off and just do nothing.

- 

- In this case, *Elvis* has been entered in the top table, but the return key has not yet been hit for the *Hair creme* entry

BasicTableModel.java

- Demonstrates a complete implementation of `TableModel` -- stores the data and generates *`fireXXXChanged()`* notifications where necessary.

```
// BasicTableModel.java

/*
 Demonstrate a basic table model implementation
 using ArrayList of rows, where each row is itself an ArrayList
 of the data in that row.
 This code is free for any purpose -- Nick Parlante.

 A row may be shorter than the number of columns
 which complicates the data handling a bit.
*/
import javax.swing.table.*;
import java.util.*;
import java.io.*;


class BasicTableModel extends AbstractTableModel {
    private ArrayList<String> colNames;   // defines the number of cols
    private ArrayList<ArrayList> data;    // arraylist of arraylists


    public BasicTableModel() {
        colNames = new ArrayList<String>();
        data = new ArrayList<ArrayList>();
    }


    /*
     Basic getXXX methods required by an class implementing TableModel
    */
```

```
// Returns the name of each col, numbered 0..columns-1
public String getColumnName(int col) {
    return colNames.get(col);
}

// Returns the number of columns
public int getColumnCount() {
    return(colNames.size());
}

// Returns the number of rows
public int getRowCount() {
    return(data.size());
}

// Returns the data for each cell, identified by its
// row, col index.
public Object getValueAt(int row, int col) {
    ArrayList rowList = data.get(row);
    Object result = null;
    if (col<rowList.size()) {
        result = rowList.get(col);
    }

    // _apparently_ it's ok to return null for a "blank" cell
    return(result);
}


// Returns true if a cell should be editable in the table
public boolean isCellEditable(int row, int col) {
    return true;
}

// Changes the value of a cell
public void setValueAt(Object value, int row, int col) {
    ArrayList rowList = data.get(row);

    // make this row long enough
    if (col>=rowList.size()) {
        while (col>=rowList.size()) rowList.add(null);
    }

    // install the data
    rowList.set(col, value);

    // notify model listeners of cell change
    fireTableCellUpdated(row, col);
}


/*
 Convenience methods of BasicTable
*/


// Adds the given column to the right hand side of the model
public void addColumn(String name) {
    colNames.add(name);
    fireTableStructureChanged();
    /*
     At present, TableModelListener does not have a more specific
     notification for changing the number of columns.
    */
}

// Adds the given row, returns the new row index
public int addRow(ArrayList row) {
    data.add(row);
    fireTableRowsInserted(data.size()-1, data.size()-1);
    return(data.size() -1);
}

// Adds an empty row, returns the new row index
public int addRow() {
```

```
        // Create a new row with nothing in it
        ArrayList row = new ArrayList();
        return(addRow(row));
    }


    // Deletes the given row
    public void deleteRow(int row) {
        if (row == -1) return;

        data.remove(row);
        fireTableRowsDeleted(row, row);
    }...
}
```

TableFrame.java

```
// TableFrame.java
/*
 Demonstrate a couple tables using one table model.
*/

import java.awt.*;
import javax.swing.*;
import java.awt.event.*;
import java.io.*;

class TableFrame extends JFrame {
    private BasicTableModel model;

    private JTable table;
    private JTable table2;

    JComponent content;

    public TableFrame(String title, File file) {
        super(title);
        content = (JComponent)getContentPane();
        content.setLayout(new BorderLayout(6,6));

        // Create a table model
        model = new BasicTableModel();

        // Create a table using that model
        table = new JTable(model);

        // there are many options for col resize strategy
        //table.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
        // JTable.AUTO_RESIZE_OFF

        // Create a scroll pane in the center, and put
        // the table in it
        JScrollPane scrollpane = new JScrollPane(table);
        scrollpane.setPreferredSize(new Dimension(300,200));
        content.add(scrollpane, BorderLayout.CENTER);

        // Create a second table using the same model, and put in the south
        table2 = new JTable(model);
        scrollpane = new JScrollPane(table2);
        scrollpane.setPreferredSize(new Dimension(300,200));
        content.add(scrollpane, BorderLayout.SOUTH);

        ...
```

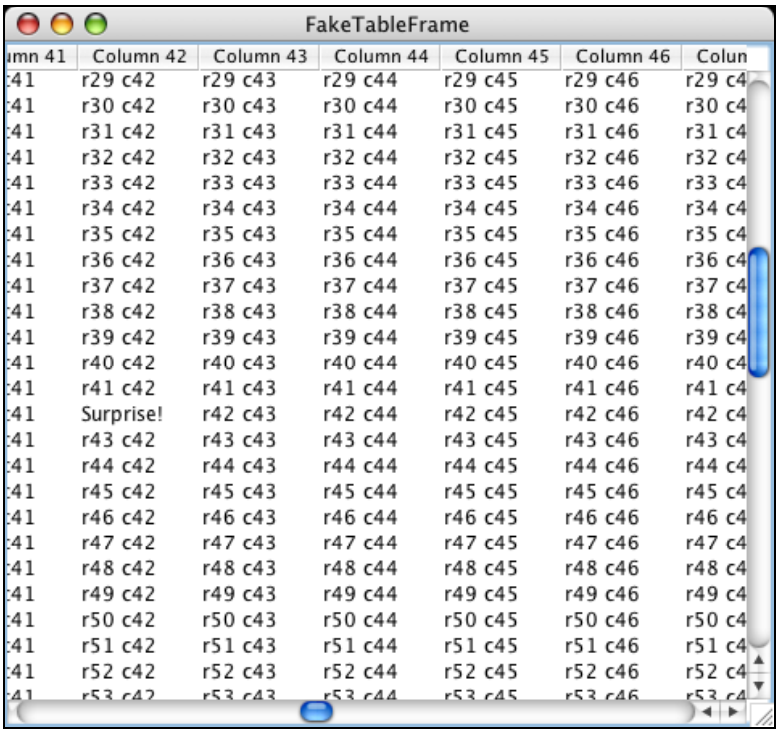
TableModel Interface Flexibility

- Any object that responds to the `TableModel` messages can play the role of table model, and `JTable` will display/scroll its data
- i.e. respond to `getRowCount()`, `getValueAt()`, etc.

- This is a good example of the use of Java interfaces -- allows any class to play a role, so long as it responds to the right messages

e.g. FakeTableModel

- Claims to have 100 x 100 data
- In reality, stores nothing -- just makes the strings up in `getValueAt()`
- Except the "surprise" location, which has the value "Surprise!"




```
/*
FakeTableModel
A little example of a TableModel that appears to be a 100 x 100
table of strings like "r2 c3". Except one location,
which is the string "Surprise!".
In reality, there is no 2-d data, we just make it up
in getValueAt.
Anything that can respond to the TableModel messages appears
to be a table model.  Neat example of interfaces in use for modularity.
*/
class FakeTableModel extends AbstractTableModel {
public static final int SIZE = 100;
    private int surprise;

    public FakeTableModel(int surprise) {
        this.surprise = surprise;
    }
```

```
    // Basic Model overrides
    public String getColumnName(int col) {
        return("Column " + col);
    }
    public int getColumnCount() { return(SIZE); }
    public int getRowCount() { return(SIZE); }

    // Returns a string like "r3c4" for each cell, except the
    // rwo==col==surprise cell which is "Surprise!"
    public Object getValueAt(int row, int col) {
        if (row==surprise && col==surprise) return "Surprise!";
        else return("r" + row + " c" + col);
    }
}
```

e.g. SQL Results in JTable

- Could have an object that has a connection to your back end SQL database.
- Suppose there is a table with fields *name* and *phone number*
- Could implement `TableModel`, and making it look like a table with one row for each record in the database, and 2 columns. In `getValueAt()` etc. get the data from the database and return it.
- This is the delegate/adapter pattern -- we are a `TableModel` object, but we don't store anything. We have a pointer to a delegate object to provide the data as needed.