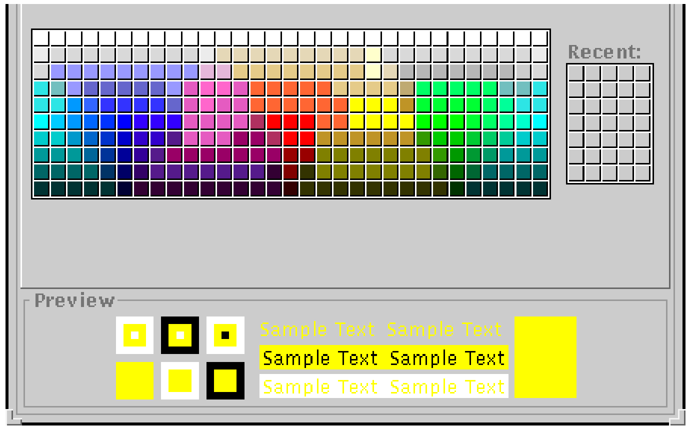
Other Components

JColorChooser

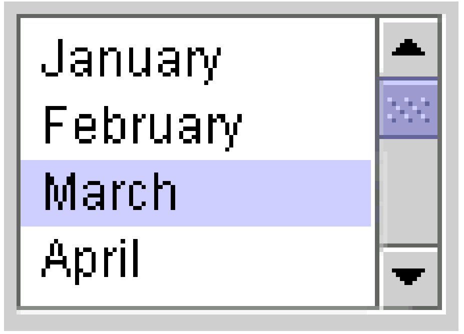
a dialog box that allows the user to choose a color from a palette



- `public JColorChooser()`
- `public JColorChooser(Color initial)`
- `public Color showDialog(Component parent, String title, Color initialColor)`
  - returns `null` if user chooses the Cancel button

Jlist - revisit

a list of selectable pre-defined text items



- `public JList()`
  creates an empty list.
- `public JList(String[] entries)`
  creates a list which contains the strings of array entries. The first string is the first, i.e. top-most, list entry.
- `JList(ListModel lModel)`
- `setModel(ListModel lModel)`

-
  ```
  public void addListSelectionListener(
              ListSelectionListener lsl)
  ```
  Adds the given listener to be informed when the selected index changes for this list.

Jlist - revisit

- `void setListData(String[] entries)`
  - sets the list entries to the strings in array entries. The first string is the first, i.e. top-most, list entry.
- `void setSelectionMode(int selectionMode)`
  - sets the way in which the user is allowed to select list entries.
    - SINGLE_SELECTION – only one list entry can be selected, if a new one is selected the old selection is dropped;
    - SINGLE_INTERVAL_SELECTION – one group of adjacent entries can be selected;
    - MULTIPLE_INTERVAL_SELECTION – arbitrary combinations of entries can be selected. (The default value)
  - The three constants are defined in class ListSelectionModel.
- `void getSelectionMode()`
  - determines which selection mode is currently used. The returned integer value is one of those described for method setSelectionMode above.
- `void setVisibleRowCount(int rowNo)`
  - sets the number of rows visible of the list. This adjusts the height of the graphical component such that rowNo shows how many rows are displayed.
- `int getSelectedIndex()`
  - returns the index of the (first) selected entry or −1 if no entry is selected.
- `int[] getSelectedIndices()`
  - returns the indices of the selected entries in an integer array.
- `Object getSelectedValue()`
  - returns the (first) selected entry (not the index but the entry itself) or null if nothing is selected.
- `Object[] getSelectedValues()`
  - returns the selected entries (not the indices but the entries themselves), or null if nothing is selected.

List Model - revisit

- DefaultListModel() DefaultListModel() is the constructor which creates a list model without any entries.
- int size() size() returns the number of entries currently in the list model.
- boolean contains(Object entry) contains(Object entry) returns true if object entry is in the list model, and false otherwise.
- Object get(int position) get(int position) returns the list entry at position position. Throws an exception if position is negative or greater than or equal to the current number of list entries. As the return type is Object, an explicit cast to the correct type might be needed.
- void insertElementAt(Object entry, int position) insertElementAt(Object entry, int position) adds entry entry at position position to the list model. Throws an exception if position is an invalid index.
- void addElement(Object entry) addElement(Object entry) adds an entry at the end of the list.
- void removeElementAt(int position) removeElementAt(int position) removes the entry at position position. Throws an exception if position is an invalid index. The remaining list entries are renumbered to maintain a consecutive indexing.
- boolean removeElement(Object entry) removeElement(Object entry) removes the entry entry. If successful true is returned, otherwise false is returned.
- void removeAllElements() removeAllElements() removes all entries from the list model.

ListSelectionListener Interface

- **public void** `valueChanged(ListSelectionEvent evt)`
- This method is automatically called by the runtime system every time the selection is changed. It is not executed if the selection is unchanged, for example if the user clicks in an item that is already selected
- The body of this method has to contain the code that should be executed in response to the selection.

`ListSelectionEvent`

- `getFirstIndex()` returns the position of the first (top-most) entry, the selection mode of which might have changed (from selected to non-selected or vice versa). 'Might' refers to the fact that the section state might be the same as before. For example if entries 5 to 8 were selected and the user now selects 5 to 10 then the first selected index remains unchanged. Which entries are selected can be found by using method `getSelectedIndices` of class JList.
- `getLastIndex()` returns the position of the last (bottom-most) entry, the selection mode of which might have changed. See also `getFirstIndex`().
- `getIsAdjusting` – changing the selection in a list triggers a series of actions, each of which generates a list selection event. For example previously selected entries become unselected before new ones become selected. Usually, one would prefer this series of events to be completed before the application reacts to the change. For such a rapid series of events all but the last one return true as a result of `getIsAdjusting`. Only the last one returns false as a result of `getIsAdjusting`. This can be used to wait for the end of the series of adjustments before taking further action.

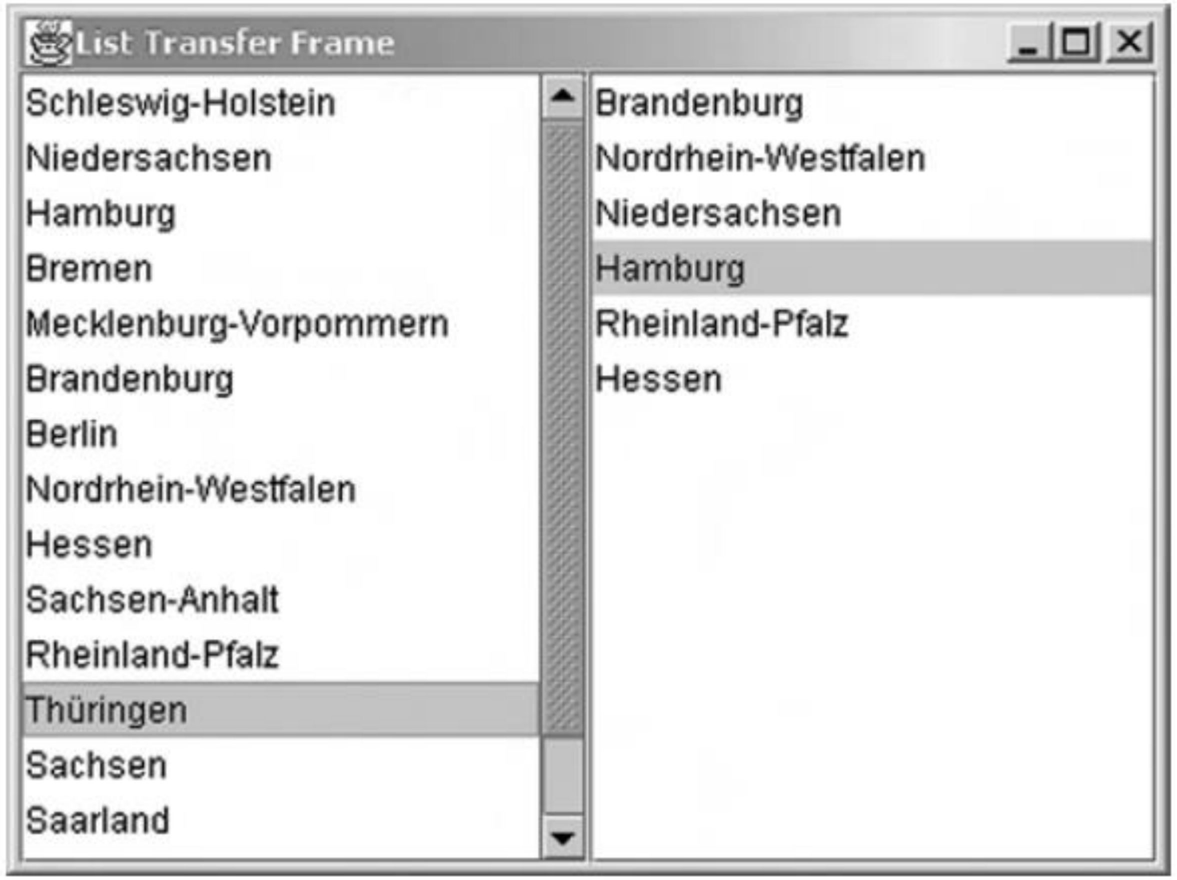
- At the moment, `ListSelectionEvent` supports only the selection modes `SINGLE_SELECTION` and `SINGLE_INTERVAL_SELECTION`.

JList example

```
Import SimpleFrame.SimpleFrame;
import java.awt.*;
import javax.swing.*;
import javax.swing.event.*;

public class ListTransferFrame extends SimpleFrame
{
  private JList leftList, rightList;
  private DefaultListModel rightListModel;
  JButton transferButton;
  String[] entries = {"Schleswig-Holstein", "Niedersachsen", "Hamburg",
                      "Bremen", "Mecklenburg-Vorpommern", "Brandenburg", "Berlin",
                      "Nordrhein-Westfalen","Hessen","Sachsen-Anhalt",
                      "Rheinland-Pfalz","Th/*@�@*/ringen","Sachsen","Saarland",
                      "Bayern","Baden-W/*@�@*/rttemberg"};

  public ListTransferFrame()
  {
    this.setSize(400,300);
    leftList  = new JList(entries);

    rightListModel = new DefaultListModel();
    rightList = new JList(rightListModel);
    leftList.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);

    TransferListener selLis = new TransferListener();
    leftList.addListSelectionListener(selLis);

    JPanel listPanel = new JPanel();
    listPanel.setLayout(new GridLayout(1,2));
    JScrollPane leftScrollPane  = new JScrollPane(leftList);
    JScrollPane rightScrollPane = new JScrollPane(rightList);
    listPanel.add(leftScrollPane);
    listPanel.add(rightScrollPane);
    this.getContentPane().add(listPanel,BorderLayout.CENTER);
  }
```

JList example cont.

```
public static void main(String[] args)
  {
    ListTransferFrame LLF = new ListTransferFrame();
    LLF.showIt("List Transfer Frame");
  }

//Listener as internal class
  class TransferListener implements ListSelectionListener{

    public TransferListener(){}

    public void valueChanged(ListSelectionEvent evt){
     if(!evt.getValueIsAdjusting())
     {
     String sel = (String)leftList.getSelectedValue();
     if(rightListModel.contains(sel))
      {
         rightListModel.removeElement(sel);
      }
     else
      {
         rightListModel.addElement(sel);
      }//ifelse
     }//if
   }//valueChanged

  }//internal class
}
```



Tables

Tables are used to display data with a two-dimensional structure

- `JTable` is used to visualize tables. As for lists, a model is internally used to handle the data.
  - If the data are complex or change while the program is running, it advisable to explicitly define such a table model.
  - A table consists of cells arranged in a rectangular grid. It is possible to add headings to every column
- `JTables` are column-oriented.
  - This means that there is support for the manipulation of columns (for example swapping two columns or rendering the column content) but only limited support for row manipulation.
  - When designing a table one should therefore try to arrange the data in such a way that a column contains data of the same type (strings, integers, images, etc.).

Tables

- `JTable(Object[][] content,Object[] columnNames)`
- The data are organized in a two-dimensional array content[][].
    - It is interpreted as an array consisting of the rows, i.e. content[i][] is the one dimensional array containing the entries of the ith row of the table. In a second,
- one-dimensional array columnNames the headers of the columns are passed.
    - The rows and the array with the column names all have to have the same length

- Note that the arrays are of type Object.
- If ‘non-standard’ types are used, one has to ensure that the program knows how to render them by supplying a user defined renderer.
- This is done by implementing the interface `TableCellRenderer`.
- By default, `JTable` uses a `DefaultTableCellRenderer` which can display the ‘standard’ types such as strings, images and boolean values. For most purposes this is sufficient.
- `setAutoResizeMode(JTable.AUTO_RESIZE_OFF)`
    - To switch the automatic width adjustment off
- `setPreferredScrollableViewportSize(Dimension size)` sets the preferred size of the viewport for this table.

- `void setFillsViewportHeight(boolean fillsViewportHeight)` sets whether or not this table is always made large enough to fill the height of an enclosing viewport. If the preferred height of the table is smaller than the viewport, then the table will be stretched to fill the viewport. In other words, this ensures the table is never smaller than the viewport. The default for this property is false.

JTable

- `int  getSelectedColumn()` Returns the index of the first selected column, -1 if no column is selected.
- `int  getSelectedColumnCount()` Returns the number of selected columns.
- `int[]      getSelectedColumns()` Returns the indices of all selected columns.
- `int  getSelectedRow()` Returns the index of the first selected row, -1 if no row is selected.
- `int  getSelectedRowCount()` Returns the number of selected rows.
- `int[]      getSelectedRows()` Returns the indices of all selected rows.
- `void addColumn(TableColumn aColumn)` Appends aColumn to the end of the array of columns held by this JTable's column model.
- `int getRowCount()` Returns the number of rows that can be shown in the JTable, given unlimited space.
- `TableModel getModel()` Returns the TableModel that provides the data displayed by this JTable.
- `TableColumnModel getColumnModel()` Returns the TableColumnModel that contains all column information of this table.

Table Interfaces

- `TableCellEditor:` This interface defines the method any object that would like to be an editor of values for components such as JListBox, JComboBox, JTree, or JTable needs to implement.
- `TableCellRenderer:` This interface defines the method required by any object that would like to be a renderer for cells in a JTable.
- `TableColumnModel:` Defines the requirements for a table column model object suitable for use with JTable.
- `TableModel:` The TableModel interface specifies the methods the JTable will use to interrogate a tabular data model.

Table Classes

- `AbstractTableModel` This abstract class provides default implementations for most of the methods in the `TableModel` interface.
- `DefaultTableModel` This is an implementation of `TableModel` that uses a Vector of Vectors to store the cell value objects.
- `DefaultTableCellRenderer` The standard class for rendering (displaying) individual cells in a `JTable`. Provides implementation for `TableCellRenderer` interface
- Class `DefaultCellEditor` The default editor for table cells. Provides implementation for `TableCellEditor` interface. An object of `DefaultTableCellEditor` can be constructed using check box, combo box, or text field.
    - DefaultCellEditor(JCheckBox checkBox)
    - DefaultCellEditor(JComboBox comboBox)
    - DefaultCellEditor(JTextField textField)
- `DefaultTableColumnModel` The standard column-handler for a `JTable`.
- `TableRowSorter` An implementation of `RowSorter` that provides sorting and filtering using a `TableModel`.

TableColumnModel Interface

- Implemented by Class `DefaultTableColumnModel`
- Defines the requirements for a table column model object suitable for use with `JTable`.
- `void addColumn(TableColumn aColumn)` Appends aColumn to the end of the tableColumns array.
- `void addColumnModelListener(TableColumnModelListener x)` Adds a listener for table column model events.
- `TableColumn getColumn(int columnIndex)` Returns the TableColumn object for the column at columnIndex.
- `int  getColumnCount()` Returns the number of columns in the model.
- `int  getColumnIndex(Object columnIdentifier)` Returns the index of the first column in the table whose identifier is equal to identifier, when compared using equals.
- `int  getColumnIndexAtX(int xPosition)` Returns the index of the column that lies on the horizontal point, xPosition; or -1 if it lies outside the any of the column's bounds.
- `void removeColumn(TableColumn column)` Deletes the `TableColumn` column from the tableColumns array.
- `void moveColumn(int columnIndex, int newIndex)` Moves the column and its header at `columnIndex` to `newIndex`.
- `int  getSelectedColumnCount()` Returns the number of selected columns.
- `int[]      getSelectedColumns()` Returns an array of indicies of all selected columns.

TableCellRenderer Interface

- `Component getTableCellRendererComponent(JTable table, Object value, boolean isSelected, boolean hasFocus, int row, int column)`
  - Returns the component used for drawing the cell .
  - This method is used to configure the renderer appropriately before drawing.
- Parameters:
  - table - the JTable that is asking the renderer to draw; can be null
  - value - the value of the cell to be rendered. It is up to the specific renderer to interpret and draw the value. For example, if value is the string "true", it could be rendered as a string or it could be rendered as a check box that is checked. null is a valid value
  - isSelected - true if the cell is to be rendered with the selection highlighted; otherwise false
  - hasFocus - if true, render cell appropriately. For example, put a special border on the cell, if the cell can be edited, render in the color used to indicate editing
  - row - the row index of the cell being drawn. When drawing the header, the value of row is -1
  - column - the column index of the cell being drawn

TableCellEditor Interface

- `Component getTableCellEditorComponent(JTable table, Object value, boolean isSelected, int row, int column)`
  - Sets an initial value for the editor.
  - Returns the component for editing
- Parameters:
  - table - the JTable that is asking the editor to edit; can be null
  - value - the value of the cell to be edited; it is up to the specific editor to interpret and draw the value. For example, if value is the string "true", it could be rendered as a string or it could be rendered as a check box that is checked. null is a valid value
  - isSelected - true if the cell is to be rendered with highlighting
  - row - the row of the cell being edited
  - column - the column of the cell being edited

TableModelListener Interface

- `TableModelListener` defines the interface for an object that listens to changes in a `TableModel`.
- `void    tableChanged(TableModelEvent e)`
  - This fine grain notification tells listeners the exact range of cells, rows, or columns that changed.
- Should be added into a `TableModel` or `AbstractTableModel` using the method:
  - `void addTableModelListener(TableModelListener l)`
    - Adds a listener to the list that is notified each time a change to the data model occurs.

TableModelListener Interface

- Class `TableModelEvent`
- Fields
  - `int ALL_COLUMNS` Specifies all columns in a row or rows.
  - `int column`
  - `int DELETE` Identifies the removal of rows or columns.
  - `int firstRow`
  - `int HEADER_ROW` Identifies the header row.
  - `int INSERT` Identifies the addition of new rows or columns.
  - `int lastRow`
  - `int type`
  - `int UPDATE` Identifies a change to existing data.
- Methods
- `int getColumn()` Returns the column for the event. If the return value is ALL_`COLUMNS`; it means every column in the specified rows changed.
- `int getFirstRow()` Returns the first row that changed. HEADER_`ROW` means the meta data, ie. names, types and order of the columns.
- `int getLastRow()` Returns the last row that changed.
- `int getType()` Returns the type of event - one of: INSERT, UPDATE and DELETE.

AbstractTableModel Class

- `AbstractTableModel` defined in the library `javax.swing.table.`
- Class `AbstractTableModel` requires the implementation of the following abstract methods:
  - `int getRowCount()`
    - has to be implemented to return the number of rows in the table.
  - `int getColumnCount()`
    - has to be implemented to return the number of columns in the table.
  - `Object getValueAt(int r, int c)`
    - has to be implemented to return the content of the table cell in column c of row r.
- `AbstractTableModel` contains some nonabstract methods one would often like to override:
  - `String getColumnName(int c)`
    - returns the name of column c as a string. By default, an AbstractTableModel assigns the names A, B,C, . . . , Z, AA, AB, AC, . . . to the columns. Overriding getColumnName by your own implementation allows you to specify other names.
  - `Class getColumnClass(int c)`
    - returns the class of the objects in column c. By default, an AbstractTableModel returns class Object. One should override getColumn-Class(c) to return the correct class of the objects in column c. This way one ensures that the correct renderer is used to display the entries. As mentioned above, tables are column-oriented and in every column the objects should be of the same class. To determine the appropriate class for column c one can look at the entry in the first row: getValueAt(0,c).getClass().
  - `boolean isCellEditable(int r, int c)`
    - the return value is true if the cell is editable and false otherwise. The default implementation returns false, i.e. the user cannot change the content of the cell. The programmer can override this method to make some cells editable by the user.
  - `void setValueAt(Object val, int r, int c)`
    - can be used to set the value of the cell in row r and column c to val. The default implementation has an empty body, i.e. it does not change the value of the cell. The programmer can override this method to change the values of editable cells.

AbstractTableModel Class

- fireTableCellUpdated(int row, int column) Notifies all listeners that the value of the cell at [row, column] has been updated.
- fireTableDataChanged() Notifies all listeners that all cell values in the table's rows may have changed
- fireTableRowsDeleted(int firstRow, int lastRow) Notifies all listeners that rows in the range [firstRow, lastRow], inclusive, have been deleted
- fireTableRowsInserted(int firstRow, int lastRow) Notifies all listeners that rows in the range [firstRow, lastRow], inclusive, have been inserted.
- fireTableRowsUpdated(int firstRow, int lastRow) Notifies all listeners that rows in the range [firstRow, lastRow], inclusive, have been updated
- fireTableStructureChanged() Notifies all listeners that the table's structure has changed.

JTable example

```
public class StaticTableFrame extends SimpleFrame
{

  private String[][] entries  = {{"Cell 0.0","Cell 0.1","Cell 0.2"},
                                 {"Cell 1.0","Cell 1.1","Cell 1.2"}};
  private String[] columnNames = {"Column 0","Column 1","Column 2"};


  public StaticTableFrame()
  {
    JTable table = new JTable(entries,columnNames);
    JScrollPane scrollpane = new JScrollPane(table);
    this.getContentPane().add(scrollpane);
  }

  public static void main(String[] args)
  {
    StaticTableFrame STF = new StaticTableFrame();
    STF.showIt("Static Table Frame");
  }
}
```

JTable example 2

```
import javax.swing.table.AbstractTableModel;

public class MultiplicationTableModel extends AbstractTableModel {

  private int noOfRows, noOfCols;
  public MultiplicationTableModel(int r, int c) {
    noOfRows = r;
    noOfCols = c;
  }

// Implementing the three abstract methods:
  public int getColumnCount() {
    return(noOfRows);
  }

  public Object getValueAt(int r,int c) {
   return(new Integer(r*c));
  }

  public int getRowCount() {
    return(noOfRows);
  }

 // Overriding some methods:
  public Class getColumnClass(int c){
    return( getValueAt(0,c).getClass() );
  }

  public String getColumnName(int c){
    return("Col "+c);
  }
}
```

```
Import SimpleFrame.SimpleFrame;
import javax.swing.*;

public class MultiplicationTableFrame extends SimpleFrame {

  public MultiplicationTableFrame(int r, int c) {
    MultiplicationTableModel multModel = new MultiplicationTableModel(r,c);
    JTable multTable = new JTable(multModel);
    multTable.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
    JScrollPane scrollPane = new JScrollPane(multTable);
    this.setSize(350,250);
    this.getContentPane().add(scrollPane);
  }

  public static void main(String[] args) {
    MultiplicationTableFrame mtf30 =
        new MultiplicationTableFrame(30,30);
    mtf30.showIt("Multiplication Table");
  }
}
```

JTable example 3

```java
import SimpleFrame.SimpleFrame;
import javax.swing.*;
public class OrderTableFrame extends SimpleFrame
{

  public OrderTableFrame()
  {
    this.setSize(300,250);
    OrderTableModel otmodel = new OrderTableModel();
    JTable JTab = new JTable(otmodel);
    JTab.setRowHeight(50);

    JScrollPane SP = new JScrollPane(JTab);
    this.getContentPane().add(SP);

  }

  public static void main(String[] args)
  {
    OrderTableFrame otframe = new OrderTableFrame();
    otframe.showIt();
  }
}
```

```java
import javax.swing.ImageIcon;
import javax.swing.table.AbstractTableModel;

public class OrderTableModel extends AbstractTableModel
{
  private String[] columnNames = {"Product","Picture","Price",
                                  "Quantity","Total"};
  private String[] products    = {"Circle","Triangle","Rectangle"};
  private String[] imageName   = {"circ.png","tria.png","rect.png"}
  private int[] quantities     = {0,0,0};
  private double[] prices      = {10.00, 12.00, 12.50};
  public boolean[] bv= {true,false,true};

  public OrderTableModel()
  {
  }
  public int getColumnCount()
  {
    return(columnNames.length);
  }

  public int getRowCount()
  {
    return(products.length+1);
  }

   public String getColumnName(int c) {
       return(columnNames[c]);
    }

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

JTable example 3 cont.

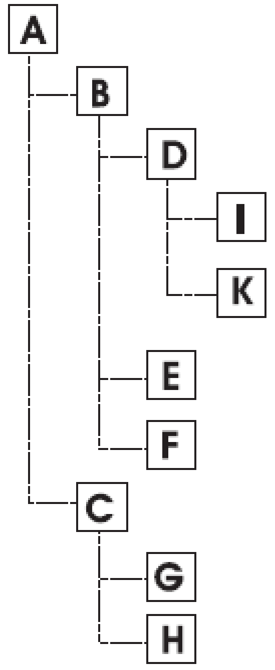
```
public Object getValueAt(int r, int c)
{
 Object result = new Object();
  if( r < products.length) {
   switch(c){
    case 0: result = products[r]; break;
    case 1: result = new ImageIcon(imageName[r]);  break;
    case 2: result = new Double(prices[r]); break;
    case 3: result = new Integer(quantities[r]);   break;
    case 4: int quantity = ((Integer)getValueAt(r,3)).intValue();
            double price = ((Double)getValueAt(r,2)).doubleValue();
            result = new Double(quantity * price);  break;
   }//switch
  }
  else{
   switch(c){
    case 0: result = new String("SUM"); break;
    case 1: result = new Object();    break;
    case 2: result = new Double(0.0); break;
    case 3: result = new Integer(0);  break;
    case 4: double sum = 0.0;
            double ee;
            for (int i = 0; i < products.length; i++) {
              sum += ((Double)getValueAt(i,4)).doubleValue();
            }
            result = new Double(sum);  break;
   }
  }
  return(result);
}
```

```
//cells in column 3 can be edited
   public boolean isCellEditable(int r, int c) { return(c == 3);
}

   public void setValueAt(Object obj, int r, int c)
   {
     if(c == 3){
       quantities[r] = ((Integer)obj).intValue();
     }
        this.fireTableDataChanged();
   }

}
```

Trees

- Hierarchical structures can often be represented by *trees*.
- Examples of hierarchical structures are
  - Family trees,
  - The organization structure of a company,
  - The directory structure on a computer hard disk, or
  - The embedding structure of Swing components

- For the abstract representation of the tree structure we use the non-graphical classes `DefaultTreeModel` and `DefaultMutableTreeNode`. The classes are locatec `javax.swing.tree`



DefaultMutableTreeNode

Implementation of the interface `TreeNode`

- `DefaultMutableTreeNode(Object label)` defines a mutable tree node. The node receives object label as label.
  - When a node is displayed one sees a symbol and the label to the right of the symbol. The label is often a string. The term ‘mutable’ refers to the fact that clicking on such a node in the display makes the subtree under this node appear or disappear. This function is automatically supplied by `DefaultMutableTreeNode`.
- `add(DefaultMutableTreeNode node)` adds node as the right-most child to this node.

When nodes are displayed, they receive a default symbol.

- For DefaultMutableTreeNode this is the ‘folder’ symbol of Java if the node is not a leaf.
- If the node is a leaf the ‘file’ symbol of Java is displayed.

TreeModel

- Define the abstract structure of the tree.
- This is done by specifying the root and, for every node, specifying its children and their order. We shall use the following methods.
- `DefaultTreeModel(TreeNode root)` creates a tree model with root node root.
    - We may use a `DefaultMutableTreeNode` instead of `TreeNode`. There is no parameterless constructor; you must always specify a root node!
- `int getChildCount(TreeNode parent)` returns the number of children of parent.
    - If parent has n children they are numbered 0, 1, . . . , n− 1.
- `boolean isLeaf(TreeNode node)` returns true if node is a leaf, i.e. has no children, and false otherwise.
- `insertNodeInto(MutableTreeNode newChild, MutableTreeNode parent, int k)` makes `newChild` the kth child of `parent`. If `parent` has n children then k has to be one of 0, 1, 2, . . . , n otherwise a runtime error will occur. If k is strictly less than n then the former children k through n− 1 become children k+ 1 through n and `newChild` is squeezed in between children k− 1 and k+ 1.
- `Object getChild(TreeNode parent, int k)` returns the kth child of `parent` as an Object and null if the `parent` does not have a kth child.
- `removeNodeFromParent(TreeNode child)` removes `child`from its parent. Note

that the whole subtree with root `child` is also removed.

JTree

- `JTree(TreeModel treeModel)` receives a `TreeModel` (or a `DefaultTreeModel`) as parameter.

`setTreeModel(TreeModel treeModel)` is used to set the tree model

`putClientProperty("JTree.lineStyle","Angled")` is used to display the angled lines corresponding to tree’s edges.

Alternative to Angled’

- ‘Horizontal’, where horizontal lines between the nodes indicate the structure,
- ‘None’ where only indentation is used.

JTree example

```
import javax.swing.*;
import javax.swing.tree.*;

public class BiologyTree extends JTree {

  private DefaultTreeModel biologyTreeModel;

  public BiologyTree(){
     makeModel();
     this.setModel(biologyTreeModel);
     this.putClientProperty("JTree.lineStyle","Angled");
  }

  private void makeModel(){
    DefaultMutableTreeNode root    = new DefaultMutableTreeNode("Trees");
    DefaultMutableTreeNode leaved  = new DefaultMutableTreeNode("Leaved Trees");
    DefaultMutableTreeNode conifer = new DefaultMutableTreeNode("Conifers");
    DefaultMutableTreeNode beech   = new DefaultMutableTreeNode("Beech");
    DefaultMutableTreeNode oak     = new DefaultMutableTreeNode("Oak");
    DefaultMutableTreeNode birch   = new DefaultMutableTreeNode("Birch");
    DefaultMutableTreeNode pine    = new DefaultMutableTreeNode("Pine");
    DefaultMutableTreeNode fir     = new DefaultMutableTreeNode("Fir");
    DefaultMutableTreeNode beechR  = new DefaultMutableTreeNode("Red Leaved");
    DefaultMutableTreeNode beechG  = new DefaultMutableTreeNode("Green Leaved");

    biologyTreeModel = new  DefaultTreeModel(root);
    biologyTreeModel.insertNodeInto(leaved ,root   ,0);
    biologyTreeModel.insertNodeInto(conifer,root   ,1);
    biologyTreeModel.insertNodeInto(beech  ,leaved ,0);
    biologyTreeModel.insertNodeInto(oak    ,leaved ,1);
    biologyTreeModel.insertNodeInto(birch  ,leaved ,1);
    biologyTreeModel.insertNodeInto(pine   ,conifer,0);
    biologyTreeModel.insertNodeInto(fir    ,conifer,1);
    biologyTreeModel.insertNodeInto(beechR ,beech  ,0);
    biologyTreeModel.insertNodeInto(beechG ,beech  ,1);
  }
}
```

```
import SimpleFrame.SimpleFrame;
import java.awt.*;

class TreeFrame extends SimpleFrame {

  public TreeFrame() {
    this.setSize(300,500);
    BiologyTree bioTree = new BiologyTree();
    this.getContentPane().add(bioTree,BorderLayout.
  }

  public static void main(String[] args) {
      TreeFrame treeFrame = new TreeFrame();
      treeFrame.showIt("Tree Frame");
  }
}
```

JTree example 2

```
import javax.swing.*;
import javax.swing.tree.*;
import java.io.File;

public class DirectoryTree extends JTree {

   private DefaultTreeModel directoryTreeModel;
   //The following path might have to be changed.
   private  String startDir = "./";
   private File startFile;

  public DirectoryTree() {
    startFile = new File(startDir);
    recursion(startFile);
    this.setModel(directoryTreeModel);
  }

  public DefaultMutableTreeNode recursion(File currentFile)
   {
    DefaultMutableTreeNode curNode =
         new DefaultMutableTreeNode(currentFile.getName());
    if(currentFile == startFile)
    {
      directoryTreeModel = new DefaultTreeModel(curNode);
     }
    if(currentFile.isDirectory())
     {
      File[] files = currentFile.listFiles();
       for (int i = 0; i < files.length; i++) {
          directoryTreeModel.insertNodeInto(recursion(files[i]),curNode,0);

       }//for i
      }
      return(curNode);
     }
}
```

```
import SimpleFrame.SimpleFrame;
import javax.swing.*;
import java.awt.*;

public class DirectoryFrame extends SimpleFrame {

  public DirectoryFrame() {
    this.setSize(300,500);
    DirectoryTree dirTree = new DirectoryTree();
    JScrollPane scrollpane = new JScrollPane(dirTree);
    this.getContentPane().add(scrollpane,BorderLayout.CEN
  }

  public static void main(String[] args) {
     DirectoryFrame dirTreeFrame = new DirectoryFrame();
     dirTreeFrame.showIt("Directory Tree Frame");
    }

}
```

JSplitPane

- Split panes can be seen as a pair of panels inside a rectangular area.
- The two components are separated by a vertical or horizontal bar, called a divider.
- The divider bar can be moved by using the mouse, thus adjusting the visible area of the two components.
- If the divider is moved to the left or right margin of the split pane (in case of a vertical separation) then only the right (or left), component is visible.
- Split panes are used, for example, if information is displayed in different formats.

JSplitPane

- `JSplitPane(int division)` is the constructor.
- Parameter division specifies whether the division is horizontal or vertical.
- There are predefined constants HORIZONTAL_SPLIT and VERTICAL_SPLIT in class JSplitPane.
    - Observe that choosing HORIZONTAL_SPLIT determines that the two components are placed horizontally next to each other; the divider bar is the vertical.
    - Similarly VERTICAL_SPLIT results in placing one component on top of the other and a horizontal divider bar.
- `setXXXComponent(Component comp)` embeds component comp into one of the two areas.
    - For XXX = Left, comp becomes the left component if HORIZONTAL_SPLIT is used.
    - If XXX = Left is used with VERTICAL_SPLIT then comp becomes the top component. The other choices for XXX then have the obvious meanings.
    - XXX can be Left, Right, Top, Bottom
- `setDividerSize(int width)` determines the width of the divider bar in pixels.
- `setDividerLocation(int pos)` determines the position of the divider bar.
    - Parameter pos is the distance of the divider bar from the left margin of the splitpane when HORIZONTAL_SPLIT is used, and from the upper margin when VERTICAL_ SPLIT is used.
- `setDividerLocation(double fraction)` determines the position of the divider bar. Parameter fraction is the fraction of the JSplitPane's size horizontal or vertical size, depending on the way it is split.
    - Therefore fraction has to be in the interval [0, 1], otherwise an IllegalArgumentException is thrown.
    - The fraction refers to the current width of the splitpane. If this method is called before the component is visible then the width is zero and the divider will be at the left (or top) margin.
- setOneTouchExpandable(boolean b) – if parameter b is true then two small arrows appear on the divider bar.
    - They point towards the two areas. When clicking on one arrow the divider bar jumps in that direction. If the divider bar is not at a margin of the split pane, it jumps to that margin to which the arrow points. If it is at the margin it jumps back to the previous position.
    - If parameter b is false the arrows are not shown and the described functions are not available.

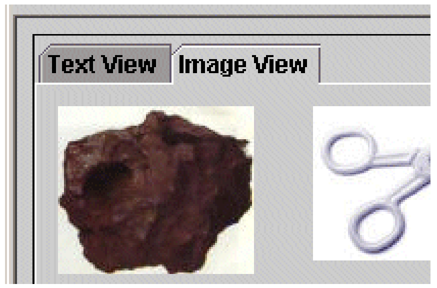
JSplitPane Example

```
Import SimpleFrame.SimpleFrame;
import java.awt.*;
import java.awt.event.*;
import java.io.*;
import javax.swing.*;

public class SplitPaneFrame extends SimpleFrame {

  private JButton okButton;
  private JList list;
  private JLabel picLabel1, picLabel2;
  private ImageIcon picture1, picture2;
  private static String picturePath = ".";


  public SplitPaneFrame()  {

   this.setSize(800,400);
   JSplitPane splitPane1 = new JSplitPane(JSplitPane.HORIZONTAL_SPLIT);
   JSplitPane splitPane2 = new JSplitPane(JSplitPane.VERTICAL_SPLIT);
   splitPane1.setOneTouchExpandable(true);
   splitPane1.setDividerSize(20);
   this.getContentPane().add(splitPane1,BorderLayout.CENTER);
   splitPane1.setRightComponent(splitPane2);
   picture1 = new ImageIcon(picturePath+"/Orange.png");
   picture2 = new ImageIcon(picturePath+"/Banana.png");
   picLabel1 = new JLabel(picture1);
   JScrollPane sp2a = new JScrollPane(picLabel1);
   splitPane2.setTopComponent(sp2a);
   picLabel2 = new JLabel(picture2);
   JScrollPane sp2b = new JScrollPane(picLabel2);
   splitPane2.setBottomComponent(sp2b);
   splitPane2.setDividerLocation(150);
   splitPane1.setDividerLocation(100);
```

```
File startFile = new File(picturePath);
   list = new JList(startFile.listFiles());
   JScrollPane sp = new JScrollPane(list);
   splitPane1.setLeftComponent(sp);

   okButton = new JButton("OK");
   okButton.setBackground(Color.cyan);
   this.getContentPane().add(okButton,BorderLayout.SOUTH);
   okButton.addActionListener(new ActionListener()
   {
     public void actionPerformed(ActionEvent e)
     {
        String filename = ((File)(list.getSelectedValue())).getPath();
       if((filename.endsWith(".png")) || (filename.endsWith(".jpg")) ||
          (filename.endsWith(".PNG")) || (filename.endsWith(".JPG")))
        {
          picLabel2.setIcon(new ImageIcon(filename));
        }
     }
   });
  }

  public static void main(String[] args)
  {
    SplitPaneFrame spf = new SplitPaneFrame();
    spf.showIt("Split-Panes",200,200);
  }
}
```

JTabbedPane

a container that holds subcontainers,
each with a "tab" label and content



- *A tabbed pane is initially empty, i.e. it contains no pages. Then components are added and each of them becomes a new page.*
- *When adding a component, the text for the tab should be provided.*
- *The tabs are arranged from left to right (or from top to bottom) in the order in which the components are added.*
- *Selecting a component as the topmost one does not change the order of the tabs. This order is reflected in the so-called indices. Every page has an index (index).*
- *A tabbed pane with n pages has indices 0, 1, . . . , n− 1. The index of the component with the left most (topmost) tab is 0.*
- *The indices can, for example, be used to remove a component or to insert a new one at a certain position.*

JTabbedPane

- `public JTabbedPane()`
  - The tabs are placed on the upper edge.
- `public JTabbedPane(int tabPosition)`
  Constructs a new tabbed pane. Defaults to having the tabs on `TOP`; can be set to `JTabbedPane.BOTTOM`, `LEFT`, `RIGHT`, etc.

- `Public void add(Component comp)` adds `comp` as last page. 'Last' means that the tab is the right-most (or bottom-most) one. The tab does not have any text.
- `public void add(String tabTitle, Component comp)`
  - adds `comp` as last page. 'Last' means that the tab is the right-most (or bottom-most) one. The tab is labelled by `tabTitle`.
- `public void addTab(String tabTitle, Component comp)`
  - adds `comp` as last page. 'Last' means that the tab is the right-most (or bottom-most) one. The tab is labelled by `tabTitle`.
- `Public void setTitleAt(int pos, String tabTitle)` assigns a new text `tabTitle` to the tab at position `pos`.
- `public void setSelectedComponent(Component comp)` selects the page containing component `comp`, i.e. this page is made visible.
- `public void setSelectedIndex(int pos)` selects the page whose tab is at position `pos`, i.e. this page is made visible.

JTabbedPane

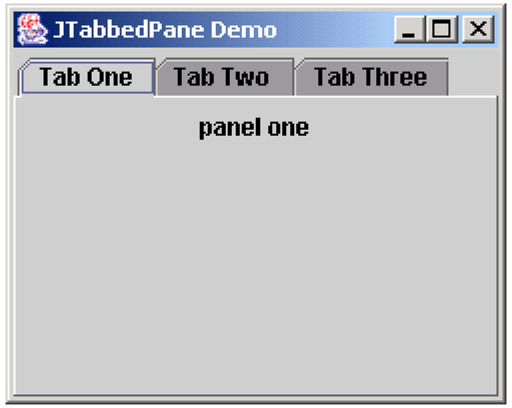
- `public void insertTab (String title, Icon icon, Component component, String tip, int index)` Inserts a new tab for the given component, at the given index, represented by the given title and/or icon, either of which may be null.
- `public void remove(Component comp)` Removes the specified Component from the JTabbedPane.
- `public void remove(int index)` Removes the tab and component which corresponds to the specified index
- `public void removeAll()` Removes all the tabs and their corresponding components from the tabbedpane.
- `Component getSelectedComponent()` Returns the currently selected component for this tabbedpane.
- `int  getSelectedIndex()` Returns the currently selected index for this tabbedpane.
- `Component   getTabComponentAt(int index)` Returns the tab component at index.
- `int  getTabCount()` Returns the number of tabs in this tabbedpane.
- `public void setTabLayoutPolicy(int tabLayoutPolicy)` Sets the policy which the tabbedpane will use in laying out the tabs when all the tabs will not fit within a single run.
  - Possible values are:
    - `JTabbedPane.WRAP_TAB_LAYOUT`
    - `JTabbedPane.SCROLL_TAB_LAYOUT`
  - The default value, if not set by the UI, is `JTabbedPane.WRAP_TAB_LAYOUT`.
- `setComponentAt(int index, Component component)` Sets the component at index to component.
- `addChangeListener(ChangeListener l)` Adds a ChangeListener to this tabbedpane.

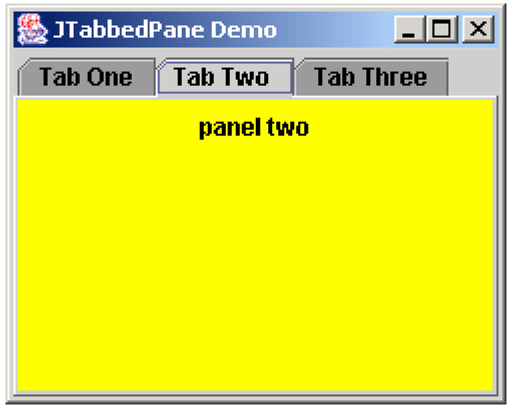
JTabbedPane Example

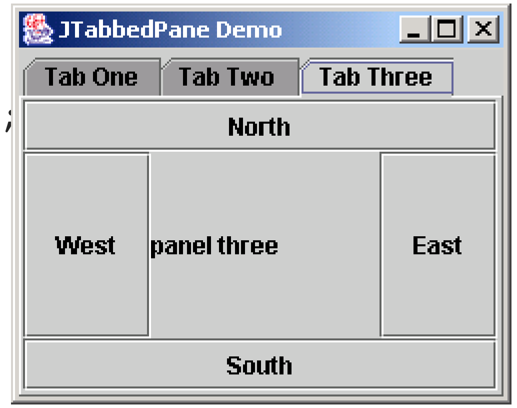
```
import java.awt.*;
import javax.swing.*;
public class JTabbedPaneDemo extends JFrame {
   // set up GUI
    public JTabbedPaneDemo() {
        super("JTabbedPane Demo ");
        JTabbedPane tabbedPane = new JTabbedPane();
         JLabel label1 = new JLabel("panel one", SwingConstants.CENTER);
        JPanel panel1 = new JPanel();
        panel1.add(label1);
        tabbedPane.addTab("Tab One", null, panel1, "First Panel");

        JLabel label2 = new JLabel("panel two", SwingConstants.CENTER);
        JPanel panel2 = new JPanel();
        panel2.setBackground(Color.YELLOW);
        panel2.add(label2);
        tabbedPane.addTab("Tab Two", null, panel2, "Second Panel");
```

JTabbedPane Example Cont.

```
// set up panel3 and add it to JTabbedPane
       JLabel label3 = new JLabel("panel three");
       JPanel panel3 = new JPanel();
       panel3.setLayout(new BorderLayout());
       panel3.add(new JButton("North"), BorderLayout.NORTH);
       panel3.add(new JButton("West"), BorderLayout.WEST);
       panel3.add(new JButton("East"), BorderLayout.EAST);
       panel3.add(new JButton("South"), BorderLayout.SOUTH);
       panel3.add(label3, BorderLayout.CENTER);
       tabbedPane.addTab("Tab Three", null, panel3, "Third Panel");

       // add JTabbedPane to container
       getContentPane().add(tabbedPane);

       setSize(250, 200);
       setVisible(true);

   } // end constructor

   public static void main(String args[]) {
       JTabbedPaneDemo tabbedPaneDemo = new JTabbedPaneDemo();
       tabbedPaneDemo.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
   }

}
```

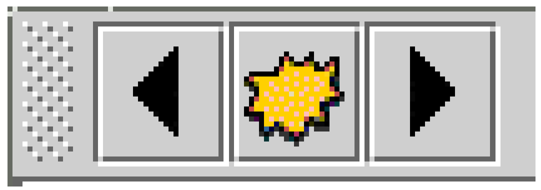






JToolbar

a movable dock container to hold common app buttons and commands



- `public JToolBar()` creates a new toolbar with horizontal orientation
- `public JToolBar(int orientation)` creates a new toolbar with specified orientation
- `public JToolBar(String title)` creates a new toolbar with specified name
- `public JToolBar(String title, int orientation)` Constructs a new tool bar, with optional title and orientation; can be `JToolBar.HORIZONTAL` or `VERTICAL`, default horizontal

- `public void add(Component comp)` Adds the given component to this tool bar.
  - Note: If using JToolbar, don't put other components in N/E/S/W.

JToolbar

addSeparator() : adds the separator to the end of toolbar

addSeparator(Dimension size) :Appends a separator of a specified dimension.

addSeparator() :Appends a separator of default size.

add(Action a) : adds a new JButton that follows the specified action.

setFloatable(boolean b) : if true is passed then the toolbar can be dragged to other locations or else not .

setLayout(LayoutManager m) : set layout of the toolbar

setOrientation(int o) : sets the orientation of toolbar

getComponentIndex(Component c) : Returns the index of the specified component.

getComponentAtIndex(int i) :Returns the component at the specified index

.