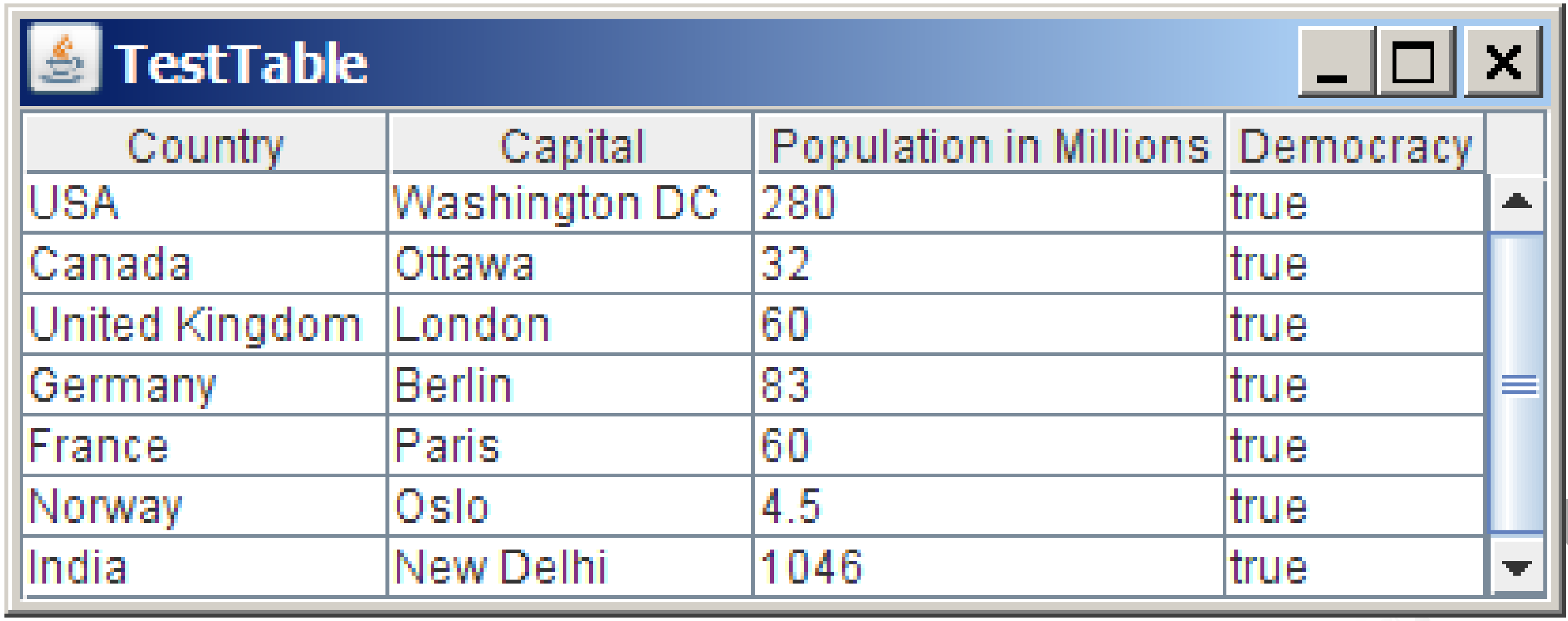
23 JTable and JTree

Chapter 36@8e

JTable

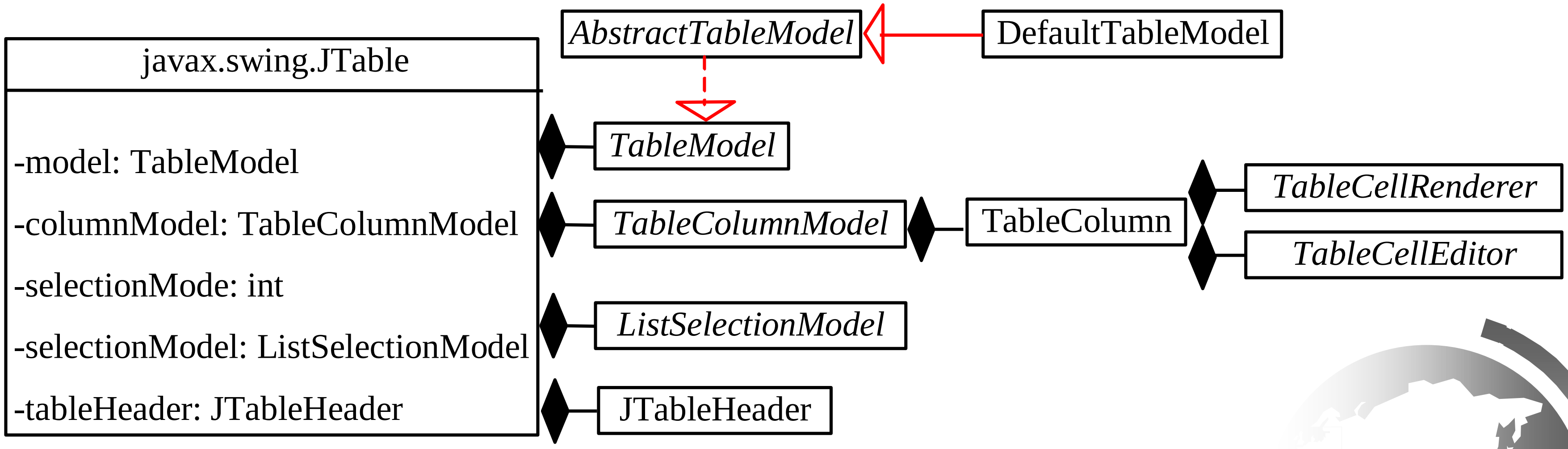
JTable is a Swing component that displays data in rows and columns in a two-dimensional grid.



Country	Capital	Population in Millions	Democracy
USA	Washington DC	280	true
Canada	Ottawa	32	true
United Kingdom	London	60	true
Germany	Berlin	83	true
France	Paris	60	true
Norway	Oslo	4.5	true
India	New Delhi	1046	true

JTable and Its Supporting Models

NOTE: All the supporting interfaces and classes for JTable are grouped in the javax.swing.table package.

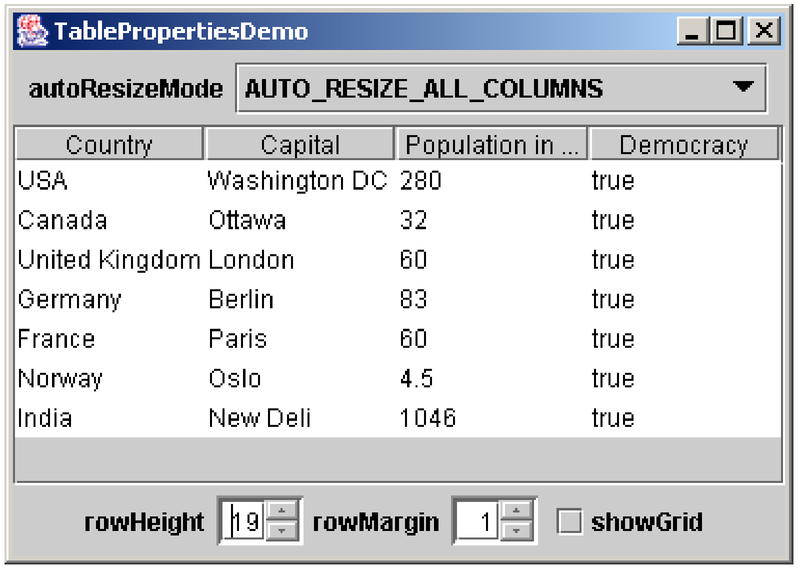


The JTable Class

javax.swing.JTable	
-autoCreateColumnsFromModel: boolean	Indicates whether the columns are created in the table (default: true).
-autoResizeMode: int	Specifies how columns are resized (default: SUBSEQUENT_COLUMNS).
-cellEditor: TableCellEditor	Specifies a cell editor.
-cellSelectionEnabled: boolean	Specifies whether individual cells can be selected (Obsolete since JDK 1.3).
-columnModel: TableColumnModel	Maintains the table column data.
-columnSelectionAllowed: boolean	Specifies whether the rows can be selected (default: false).
-editingColumn: int	Specifies the column of the cell that is currently being edited.
-editingRow: int	Specifies the row of the cell that is currently being edited.
-gridColor: java.awt.Color	The color used to draw grid lines ((default: GRAY).
-intercellSpacing: Dimension	Specifies the horizontal and vertical margins between cells (default: 1, 1).
-model: TableModel	Maintains the table model.
-rowCount: int	Read-only property that counts the number of rows in the table.
-rowHeight: int	Specifies the row height of the table (default: 16 pixels).
-rowMargin: int	Specifies the vertical margin between rows (default: 1 pixel).
-rowSelectionAllowed: boolean	Specifies whether the rows can be selected (default: true).
-selectionBackground: java.awt.Color	The background color of selected cells.
-selectionForeground: java.awt.Color	The foreground color of selected cells.
-showGrid: boolean	Specify whether the grid lines are displayed (write-only, default: true).
-selectionMode: int	Specifies a selection mode (write-only).
-selectionModel: ListSelectionModel	Specifies a selection model.
-showHorizontalLines: boolean	Specifies whether the horizontal grid lines are displayed (default: true).
-showVerticalLines: boolean	Specifies whether the vertical grid lines are displayed (default: true).
-tableHeader: JTableHeader	Specifies a table header.
+JTable()	Creates a default JTable with all default models.
+JTable(numRows: int, numColumns: int)	Creates a JTable with the specified number of empty rows and columns.
+JTable(rowData: Object[][], columnData: Object[])	Creates a JTable with the specified row data and column header names.
+JTable(dm: TableModel)	Creates a JTable with the specified table model.
+JTable(dm: TableModel, cm: TableColumnModel)	Creates a JTable with the specified table model and table column model.
+JTable(dm: TableModel, cm: TableColumnModel, sm: ListSelectionModel)	Creates a JTable with the specified table model, table column model, and selection model.
+JTable(rowData: Vector, columnNames: Vector)	Creates a JTable with the specified row data and column data in vectors.
+addColumn(aColumn: TableColumn): void	Adds a new column to the table.
+clearSelection(): void	Deselects all selected columns and rows.
+editCellAt(row: int, column: int): void	Edits the cell if it is editable.
+getDefaultEditor(column: Class): TableCellEditor	Returns the default editor for the column.
+getDefaultRenderer(col: Class): TableCellRenderer	Returns the default renderer for the column.
+setDefaultEditor(column: Class, editor: TableCellEditor): void	Sets the default editor for the column.
+setDefaultRenderer(column: Class, editor:	Sets the default renderer for the column.

Example: Table Properties Demo

Problem: This example demonstrates the use of several JTable properties. The example creates a table and allows the user to choose an Auto Resize Mode, specify the row height and margin, and indicate whether the grid is shown.

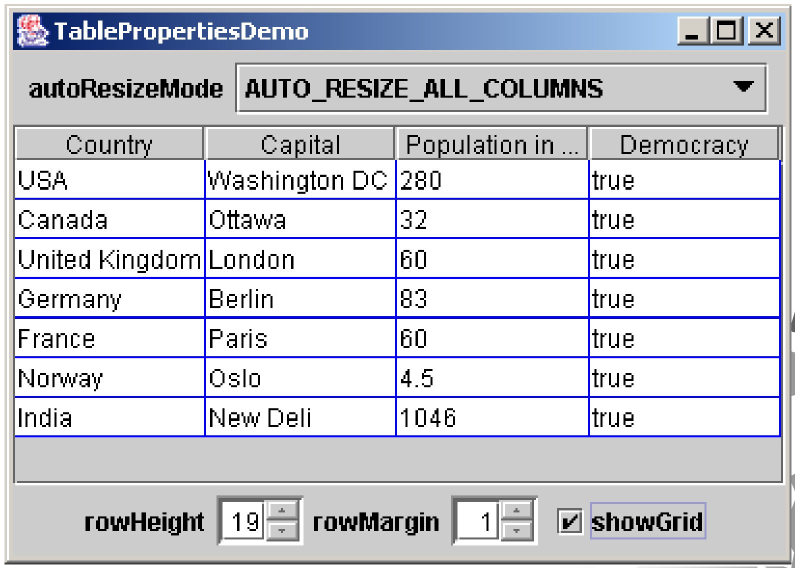


TablePropertiesDemo

autoResizeMode AUTO_RESIZE_ALL_COLUMNS

Country	Capital	Population in ...	Democracy
USA	Washington DC	280	true
Canada	Ottawa	32	true
United Kingdom	London	60	true
Germany	Berlin	83	true
France	Paris	60	true
Norway	Oslo	4.5	true
India	New Deli	1046	true

rowHeight 19 rowMargin 1 ☐ showGrid



TablePropertiesDemo

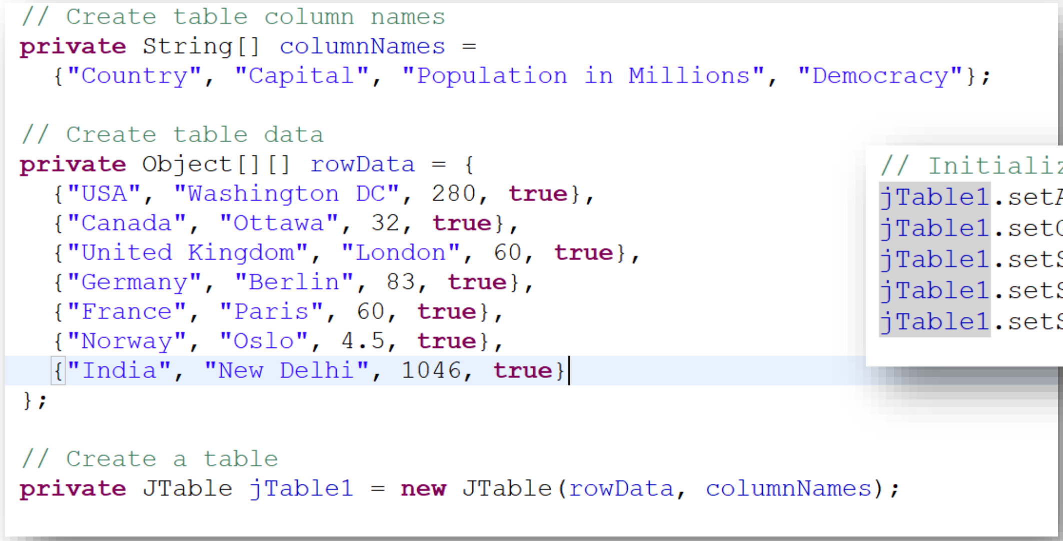
autoResizeMode AUTO_RESIZE_ALL_COLUMNS

Country	Capital	Population in ...	Democracy
USA	Washington DC	280	true
Canada	Ottawa	32	true
United Kingdom	London	60	true
Germany	Berlin	83	true
France	Paris	60	true
Norway	Oslo	4.5	true
India	New Deli	1046	true

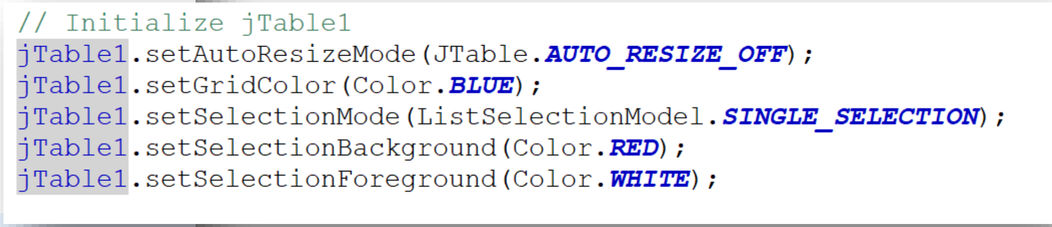
rowHeight 19 rowMargin 1 ☑ showGrid

TablePropertiesDemo

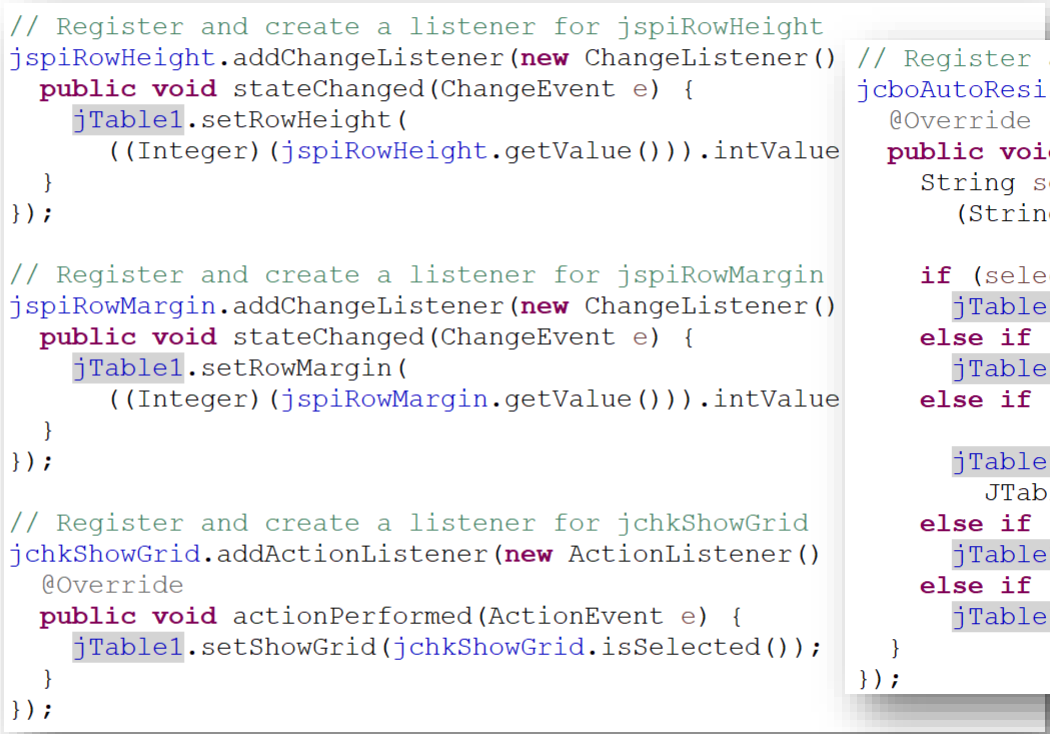
Run



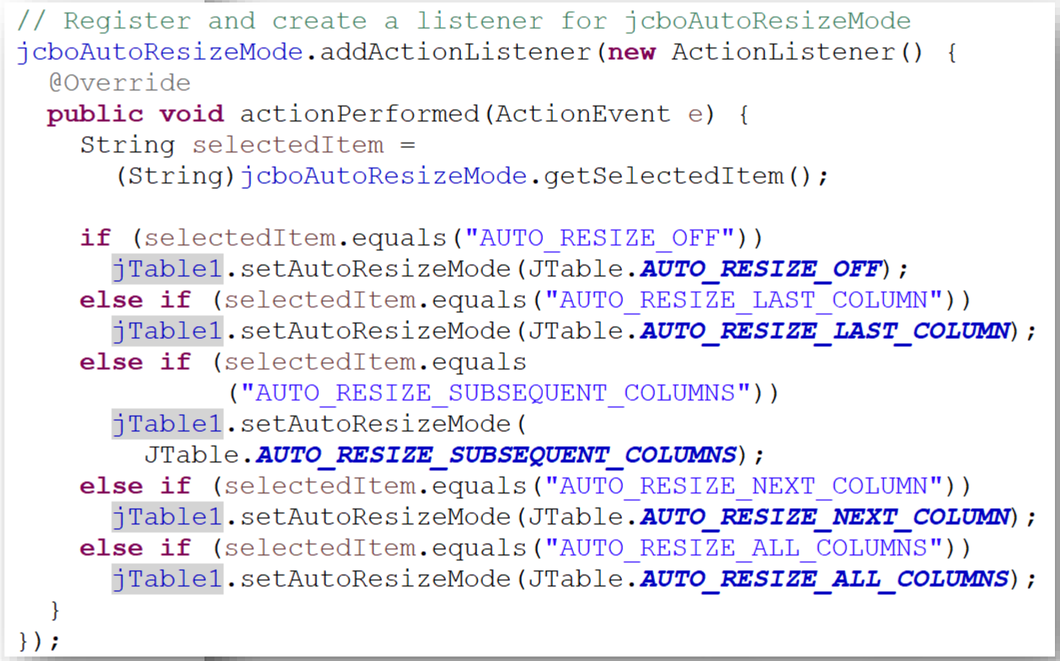
```java
// Create table column names
private String[] columnNames =
  {"Country", "Capital", "Population in Millions", "Democracy"};

// Create table data
private Object[][] rowData = {
  {"USA", "Washington DC", 280, true},
  {"Canada", "Ottawa", 32, true},
  {"United Kingdom", "London", 60, true},
  {"Germany", "Berlin", 83, true},
  {"France", "Paris", 60, true},
  {"Norway", "Oslo", 4.5, true},
  {"India", "New Delhi", 1046, true}
};

// Create a table
private JTable jTable1 = new JTable(rowData, columnNames);
```



```java
// Initialize jTable1
jTable1.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
jTable1.setGridColor(Color.BLUE);
jTable1.setSelectionMode(ListSelectionModel.SINGLE_SELECTION);
jTable1.setSelectionBackground(Color.RED);
jTable1.setSelectionForeground(Color.WHITE);
```



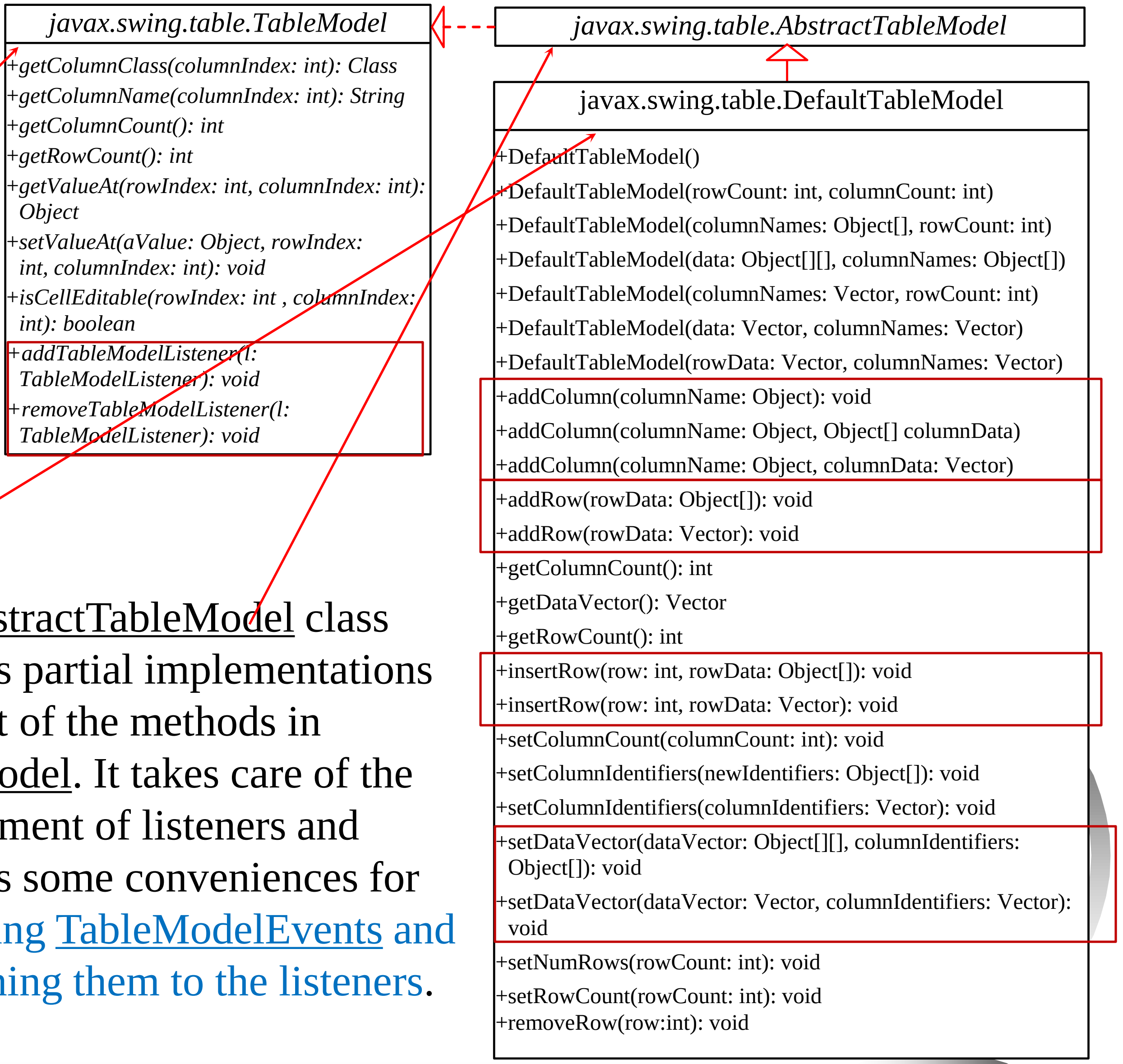
```java
// Register and create a listener for jspiRowHeight
jspiRowHeight.addChangeListener(new ChangeListener()
  public void stateChanged(ChangeEvent e) {
    jTable1.setRowHeight(
      ((Integer)(jspiRowHeight.getValue())).intValue
  }
});

// Register and create a listener for jspiRowMargin
jspiRowMargin.addChangeListener(new ChangeListener()
  public void stateChanged(ChangeEvent e) {
    jTable1.setRowMargin(
      ((Integer)(jspiRowMargin.getValue())).intValue
  }
});

// Register and create a listener for jchkShowGrid
jchkShowGrid.addActionListener(new ActionListener()
  @Override
  public void actionPerformed(ActionEvent e) {
    jTable1.setShowGrid(jchkShowGrid.isSelected());
  }
});
```



```java
// Register and create a listener for jcboAutoResizeMode
jcboAutoResizeMode.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    String selectedItem =
      (String)jcboAutoResizeMode.getSelectedItem();

    if (selectedItem.equals("AUTO_RESIZE_OFF"))
      jTable1.setAutoResizeMode(JTable.AUTO_RESIZE_OFF);
    else if (selectedItem.equals("AUTO_RESIZE_LAST_COLUMN"))
      jTable1.setAutoResizeMode(JTable.AUTO_RESIZE_LAST_COLUMN);
    else if (selectedItem.equals
              ("AUTO_RESIZE_SUBSEQUENT_COLUMNS"))
      jTable1.setAutoResizeMode(
        JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS);
    else if (selectedItem.equals("AUTO_RESIZE_NEXT_COLUMN"))
      jTable1.setAutoResizeMode(JTable.AUTO_RESIZE_NEXT_COLUMN);
    else if (selectedItem.equals("AUTO_RESIZE_ALL_COLUMNS"))
      jTable1.setAutoResizeMode(JTable.AUTO_RESIZE_ALL_COLUMNS);
  }
});
```

Table Models

JTable delegates data storing and processing to its table data model. A table data model must implement the TableModel interface, which defines the methods for registering table model listeners, manipulating cells, and obtaining row count, column count, column class, and column name.


javax.swing.table.TableModel
+getColumnClass(columnIndex: int): Class
+getColumnName(columnIndex: int): String
+getColumnCount(): int
+getRowCount(): int
+getValueAt(rowIndex: int, columnIndex: int): Object
+setValueAt(aValue: Object, rowIndex: int, columnIndex: int): void
+isCellEditable(rowIndex: int , columnIndex: int): boolean
+addTableModelListener(l: TableModelListener): void
+removeTableModelListener(l: TableModelListener): void
javax.swing.table.AbstractTableModel
javax.swing.table.DefaultTableModel
+DefaultTableModel()
+DefaultTableModel(rowCount: int, columnCount: int)
+DefaultTableModel(columnNames: Object[], rowCount: int)
+DefaultTableModel(data: Object[][], columnNames: Object[])
+DefaultTableModel(columnNames: Vector, rowCount: int)
+DefaultTableModel(data: Vector, columnNames: Vector)
+DefaultTableModel(rowData: Vector, columnNames: Vector)
+addColumn(columnName: Object): void
+addColumn(columnName: Object, Object[] columnData)
+addColumn(columnName: Object, columnData: Vector)
+addRow(rowData: Object[]): void
+addRow(rowData: Vector): void
+getColumnCount(): int
+getDataVector(): Vector
+getRowCount(): int
+insertRow(row: int, rowData: Object[]): void
+insertRow(row: int, rowData: Vector): void
+setColumnCount(columnCount: int): void
+setColumnIdentifiers(newIdentifiers: Object[]): void
+setColumnIdentifiers(columnIdentifiers: Vector): void
+setDataVector(dataVector: Object[][], columnIdentifiers: Object[]): void
+setDataVector(dataVector: Vector, columnIdentifiers: Vector): void
+setNumRows(rowCount: int): void
+setRowCount(rowCount: int): void
+removeRow(row:int): void

The AbstractTableModel class provides partial implementations for most of the methods in TableModel. It takes care of the management of listeners and provides some conveniences for generating TableModelEvents and dispatching them to the listeners.

The DefaultTableModel provides concrete storage for data using a vector.

Table Column Model

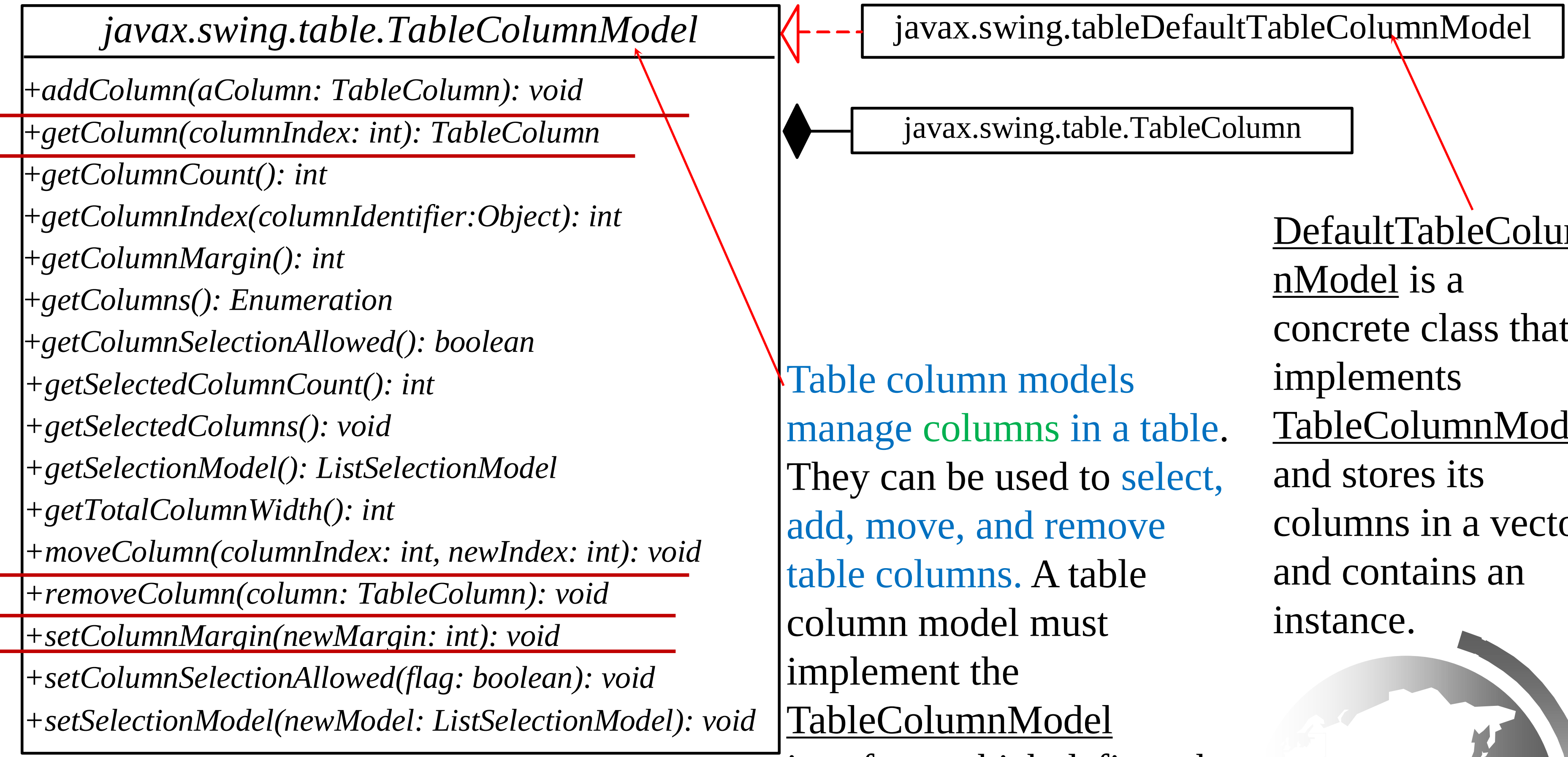

javax.swing.table.TableColumnModel
+addColumn(aColumn: TableColumn): void
+getColumn(columnIndex: int): TableColumn
+getColumnCount(): int
+getColumnIndex(columnIdentifier:Object): int
+getColumnMargin(): int
+getColumns(): Enumeration
+getColumnSelectionAllowed(): boolean
+getSelectedColumnCount(): int
+getSelectedColumns(): void
+getSelectionModel(): ListSelectionModel
+getTotalColumnWidth(): int
+moveColumn(columnIndex: int, newIndex: int): void
+removeColumn(column: TableColumn): void
+setColumnMargin(newMargin: int): void
+setColumnSelectionAllowed(flag: boolean): void
+setSelectionModel(newModel: ListSelectionModel): void
javax.swing.tableDefaultTableColumnModel
javax.swing.table.TableColumn

Table column models manage columns in a table. They can be used to select, add, move, and remove table columns. A table column model must implement the TableColumnModel interface, which defines the methods for registering table column model listeners, and for accessing and manipulating columns.

DefaultTableColumnModel is a concrete class that implements TableColumnModel and stores its columns in a vector and contains an instance.

The TableColumn Class

The column model deals with all the columns in a table. The TableColumn class is used to model an individual column in the table. An instance of TableColumn for a specified column can be obtained using the getColumn(index) method in TableColumnModel or the getColumn(columnIdentifier) method in JTable.

javax.swing.table.TableColumn	
+cellEditor: TableCellEditor	The editor for editing a cell inf this column.
+cellRenderer: TableCellRenderer	The renderer for displaying a cell in this column.
+headerRenderer: TableCellRenderer	The renderer for displaying the header of this column.
+headerValue: Object	The header value of this column.
+identifier: Object	The identifier for this column.
+maxWidth: int	The maximum width of this column.
+minWidth: int	The minimum width of this column (default: 15 pixels).
+modelIndex: int	The index of the column in the table model (default: 0).
+preferredWidth: int	The preferred width of this column (default: 75 pixels).
+resizable: boolean	Indicates whether this column can be resized (default: true).
+width: int	Specifies the width of this column (default: 75 pixels).
+TableColumn()	Constructs a default table column.
+TableColumn(modelIndex: int)	Constructs a table column for the specified column.
+TableColumn(modelIndex: int, width: int)	Constructs a table column with the specified column and width.
+TableColumn(modelIndex: int, width: int, cellRenderer: TableCellRendere)	Constructs a table column with the specified column, width, and cell renderer.
+sizeWidthToFit(): void	Resizes the column to fit the width of its header cell.

The JTableHeader Class

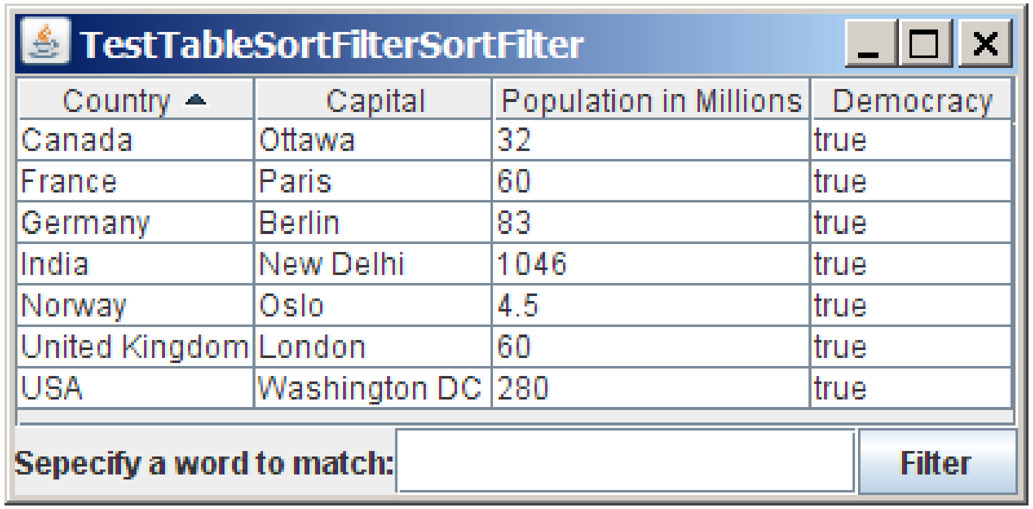
JTableHeader is a GUI component that manages the header of the JTable (see Figure 36.29). When you create a JTable, an instance of JTableHeader is automatically created and stored in the tableHeader property.

javax.swing.table.JTableHeader	
+columnModel: TableColumnModel	The TableColumnModel of the table header.
+draggedColumn: TableColumn	The column being dragged.
+draggedDistance: TableCellRenderer	The distance from its original position to the dragged position.
+reorderingAllowed: boolean	Whether reordering of columns is allowed (default: true).
+resizingAllowed: boolean	Whether resizing of columns is allowed (default: true).
+resizingColumn: TableColumn	The column being resized.
+table: JTable	The table for which this object is the header.
+JTableHeader()	Constructs a JTableHeader with a default TableColumnModel.
+JTableHeader(TableColumnModel cm)	Constructs a JTableHeader with with a TableColumnModel.

Auto Sort and Filtering

Auto sort and filtering are two useful new features in JDK 1.6. To enable auto sort on any column in a JTable, create an instance of TableRowSet with a table model and set JTable's rowSorter with this TableRowSet instance, as follows:

```
TableRowSorter<TableModel> sorter =
  new TableRowSorter<TableModel>(tableModel);
jTable.setRowSorter(sorter);
```



Country ▲	Capital	Population in Millions	Democracy
Canada	Ottawa	32	true
France	Paris	60	true
Germany	Berlin	83	true
India	New Delhi	1046	true
Norway	Oslo	4.5	true
United Kingdom	London	60	true
USA	Washington DC	280	true

Sepecify a word to match:

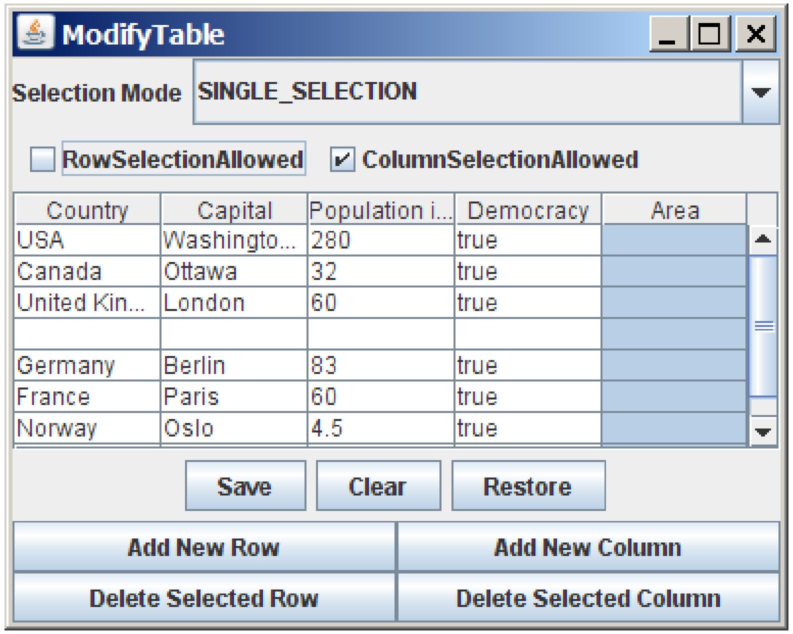
TestTableSortFilter

Run

```
// Enable auto sorter
jTable1.setRowSorter(sorter);

JPanel panel = new JPanel(new java.awt.BorderLayout());
panel.add(new JLabel("Specify a word to match:"),
  BorderLayout.WEST);
panel.add(jtfFilter, BorderLayout.CENTER);
panel.add(btFilter, BorderLayout.EAST);

add(panel, BorderLayout.SOUTH);
add(new JScrollPane(jTable1), BorderLayout.CENTER);

btFilter.addActionListener(new java.awt.event.ActionListener() {
  @Override
  public void actionPerformed(java.awt.event.ActionEvent e) {
    String text = jtfFilter.getText();
    if (text.trim().length() == 0)
      sorter.setRowFilter(null);
    else
      sorter.setRowFilter(RowFilter.regexFilter(text));
  }
});
```

Example:
Modifying Rows and Columns

Problem: This example demonstrates the use of table models, table column models, list-selection models, and the TableColumn class. The program allows the user to choose selection mode and selection type, to add or remove rows and columns, and to save, clear, and restore table.

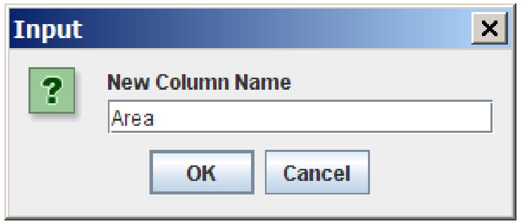


ModifyTable

Selection Mode: SINGLE_SELECTION

☐ RowSelectionAllowed ☑ ColumnSelectionAllowed

Country	Capital	Population i...	Democracy	Area
USA	Washingto...	280	true	
Canada	Ottawa	32	true	
United Kin...	London	60	true	
Germany	Berlin	83	true	
France	Paris	60	true	
Norway	Oslo	4.5	true	



Input

New Column Name: Area

ModifyTable

Run

```java
jbtAddRow.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    if (jTable1.getSelectedRow() >= 0)
      tableModel.insertRow(jTable1.getSelectedRow(),
        new java.util.Vector<String>());
    else
      tableModel.addRow(new java.util.Vector<String>());
  }
});

jbtAddColumn.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    String name = JOptionPane.showInputDialog("New Colum
    tableModel.addColumn(name, new java.util.Vector());
  }
});

jbtDeleteRow.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    if (jTable1.getSelectedRow() >= 0)
      tableModel.removeRow(jTable1.getSelectedRow(
  }
});
```

```java
jbtDeleteColumn.addActionListener(new ActionListener(
  @Override
  public void actionPerformed(ActionEvent e) {
    if (jTable1.getSelectedColumn() >= 0) {
      TableColumnModel columnModel = jTable1.getColum
      TableColumn tableColumn =
          columnModel.getColumn(jTable1.getSelectedCo
      columnModel.removeColumn(tableColumn);
    }
  }
});
```

```java
jbtClear.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    tableModel.setRowCount(0);
  }
});
```

```java
jcboSelectionMode.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    String selectedItem =
      (String)jcboSelectionMode.getSelectedItem();

    if (selectedItem.equals("SINGLE_SELECTION"))
      jTable1.setSelectionMode(
        ListSelectionModel.SINGLE_SELECTION);
    else if (selectedItem.equals("SINGLE_INTERVAL_SELECTION"))
      jTable1.setSelectionMode(
        ListSelectionModel.SINGLE_INTERVAL_SELECTION);
    else if (selectedItem.equals("MULTIPLE_INTERVAL_SELECTION"))
      jTable1.setSelectionMode(
        ListSelectionModel.MULTIPLE_INTERVAL_SELECTION);
  }
});
```

```java
jchkRowSelectionAllowed.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    jTable1.setRowSelectionAllowed(
      jchkRowSelectionAllowed.isSelected());
  }
});

jchkColumnSelectionAllowed.addActionListener(
    new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    jTable1.setColumnSelectionAllowed(
      jchkColumnSelectionAllowed.isSelected());
  }
});
```

Table Renderers and Editors

Table cells are painted by cell renderers. **By default, a cell object's string representation (toString()) is displayed and the string can be edited as it was in a text field**. **JTable maintains a set of predefined renderers and editors**, listed in Table 36.1, which can be specified to replace default string renderers and editors.

The predefined renderers and editors are automatically located and loaded to match the class returned from the getColumnClass() method in the table model. To use a predefined renderer or editor for a class other than String, you need to create your own table model **by extending a subclass of TableModel**. In your table model class, you need to override the getColumnClass() method to return the class of the column, as follows:

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

By default, all cells are editable. To prohibit a cell from being edited, override the isCellEditable(int rowIndex, int columnIndx) method in TableModel to return false. By default, this method returns true in AbstractTableModel.

Example: Using Predefined Table Renderers and Editors

Problem: Write a program that displays a table for the books. The table consists of three rows with column names Title, Copies Needed, Publisher, Date Published, In-Stock, and Book Photo, as shown in Figure 36.32. Display all the columns using the predefined renderers and editors. Assume dates and icons are not editable; prohibit users from editing of these two columns.

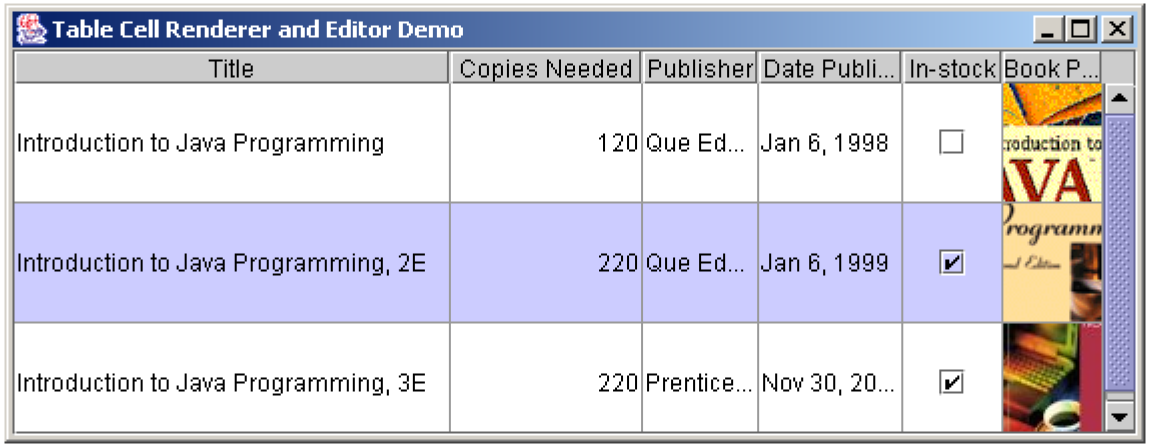
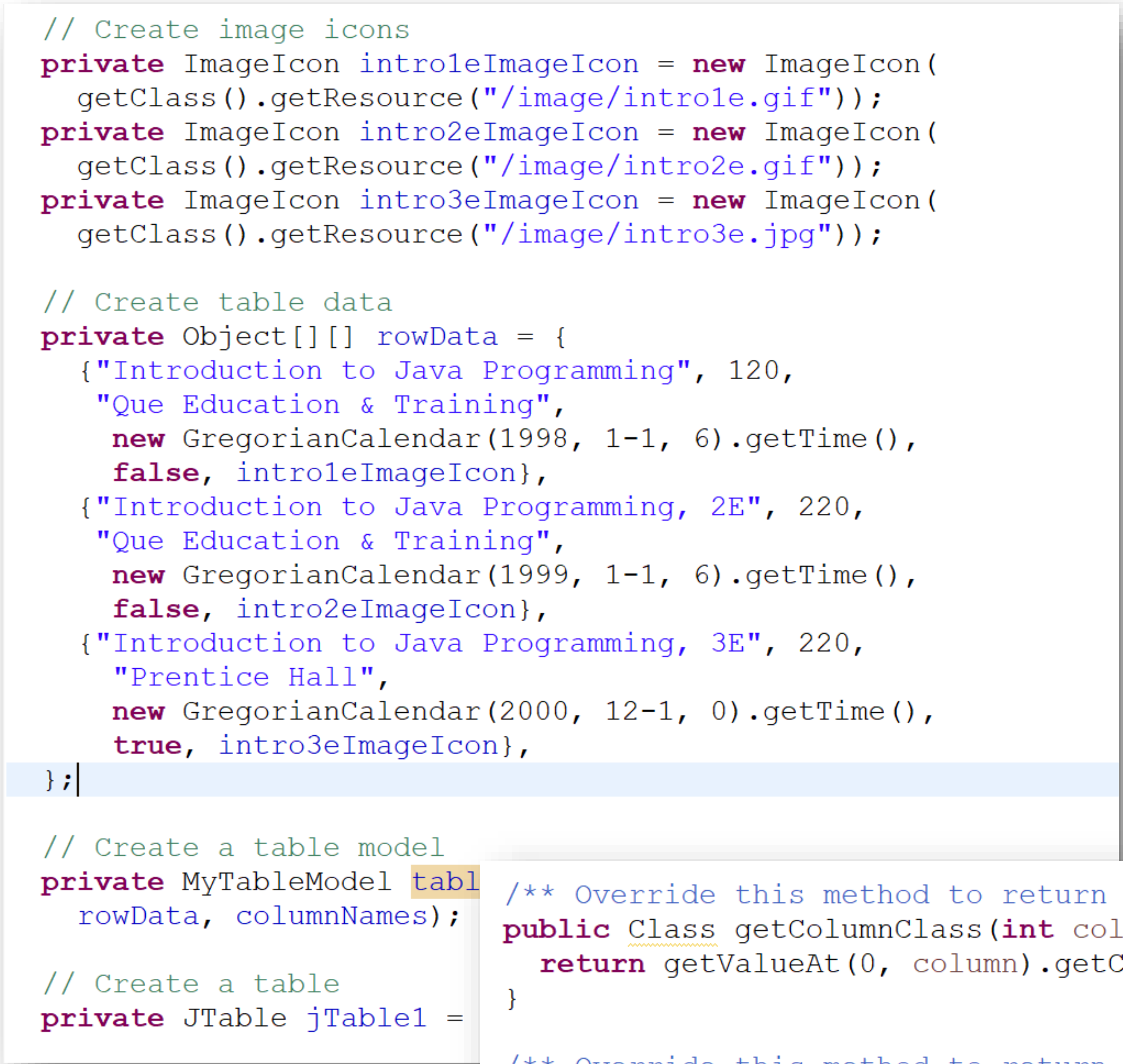


Table Cell Renderer and Editor Demo

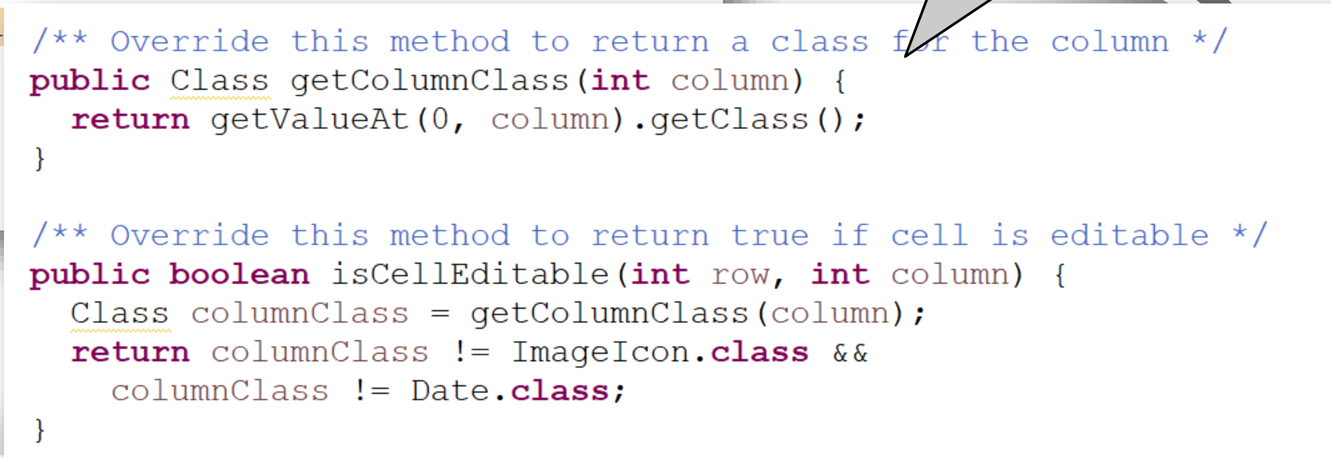
Title	Copies Needed	Publisher	Date Publi...	In-stock	Book P...
Introduction to Java Programming	120	Que Ed...	Jan 6, 1998	☐	
Introduction to Java Programming, 2E	220	Que Ed...	Jan 6, 1999	☑	
Introduction to Java Programming, 3E	220	Prentice...	Nov 30, 20...	☑	

MyTableModel TableCellRendererEditorDemo Run



```java
// Create image icons
private ImageIcon intro1eImageIcon = new ImageIcon(
  getClass().getResource("/image/intro1e.gif"));
private ImageIcon intro2eImageIcon = new ImageIcon(
  getClass().getResource("/image/intro2e.gif"));
private ImageIcon intro3eImageIcon = new ImageIcon(
  getClass().getResource("/image/intro3e.jpg"));

// Create table data
private Object[][] rowData = {
  {"Introduction to Java Programming", 120,
   "Que Education & Training",
    new GregorianCalendar(1998, 1-1, 6).getTime(),
    false, intro1eImageIcon},
  {"Introduction to Java Programming, 2E", 220,
   "Que Education & Training",
    new GregorianCalendar(1999, 1-1, 6).getTime(),
    false, intro2eImageIcon},
  {"Introduction to Java Programming, 3E", 220,
    "Prentice Hall",
    new GregorianCalendar(2000, 12-1, 0).getTime(),
    true, intro3eImageIcon},
};

// Create a table model
private MyTableModel tabl
  rowData, columnNames);

// Create a table
private JTable jTable1 =
```

MyTableModel



```java
/** Override this method to return a class for the column */
public Class getColumnClass(int column) {
  return getValueAt(0, column).getClass();
}

/** Override this method to return true if cell is editable */
public boolean isCellEditable(int row, int column) {
  Class columnClass = getColumnClass(column);
  return columnClass != ImageIcon.class &&
    columnClass != Date.class;
}
```

Custom Table Renderers and Editors

Predefined renderers and editors are convenient and easy to use, but their functions are limited. The predefined image icon renderer displays the image icon in a label. The image icon cannot be scaled. If you want the whole image to fit in a cell, you need to create a custom renderer.

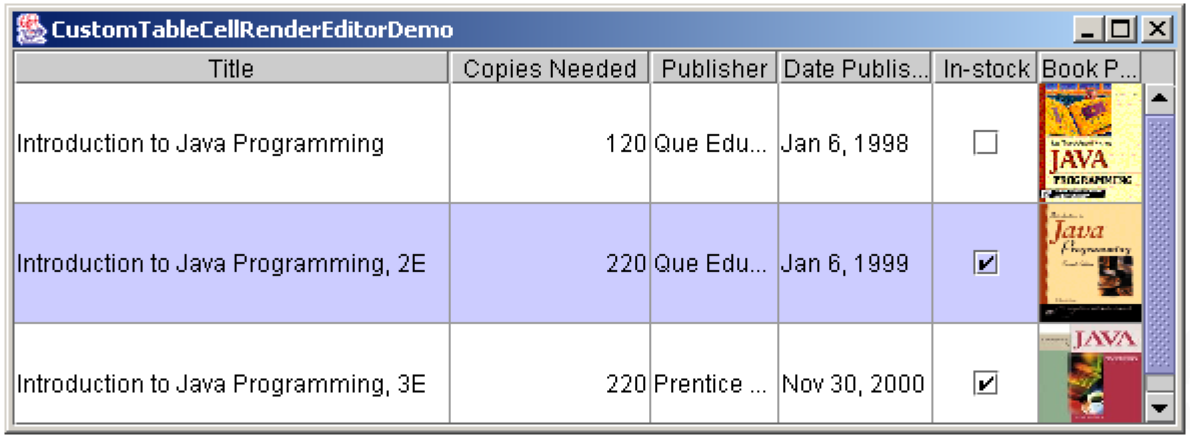
A custom renderer can be created by extending the DefaultTableCellRenderer, which is a default implementation for the TableCellRender interface. The custom renderer must **override** the getTableCellRendererComponent() to return a component for rendering the table cell. The getTableCellRendererComponent() is defined as follows:

```
  public Component getTableCellRendererComponent
    (JTable table, Object value, boolean isSelected,
     boolean isFocused, int row, int column)
```

This method signature is very similar to the getListCellRendererComponent() method used to create custom list cell renderers.

Example: Using Custom Table Renderers and Editors

Problem: Revise Example 36.9, "Using Predefined Table Renderers and Editors," to display scaled image icons and to use a custom combo editor to edit the cells in the Publisher column.



Title	Copies Needed	Publisher	Date Publis...	In-stock	Book P...
Introduction to Java Programming	120	Que Edu...	Jan 6, 1998	☐	
Introduction to Java Programming, 2E	220	Que Edu...	Jan 6, 1999	☑	
Introduction to Java Programming, 3E	220	Prentice ...	Nov 30, 2000	☑	

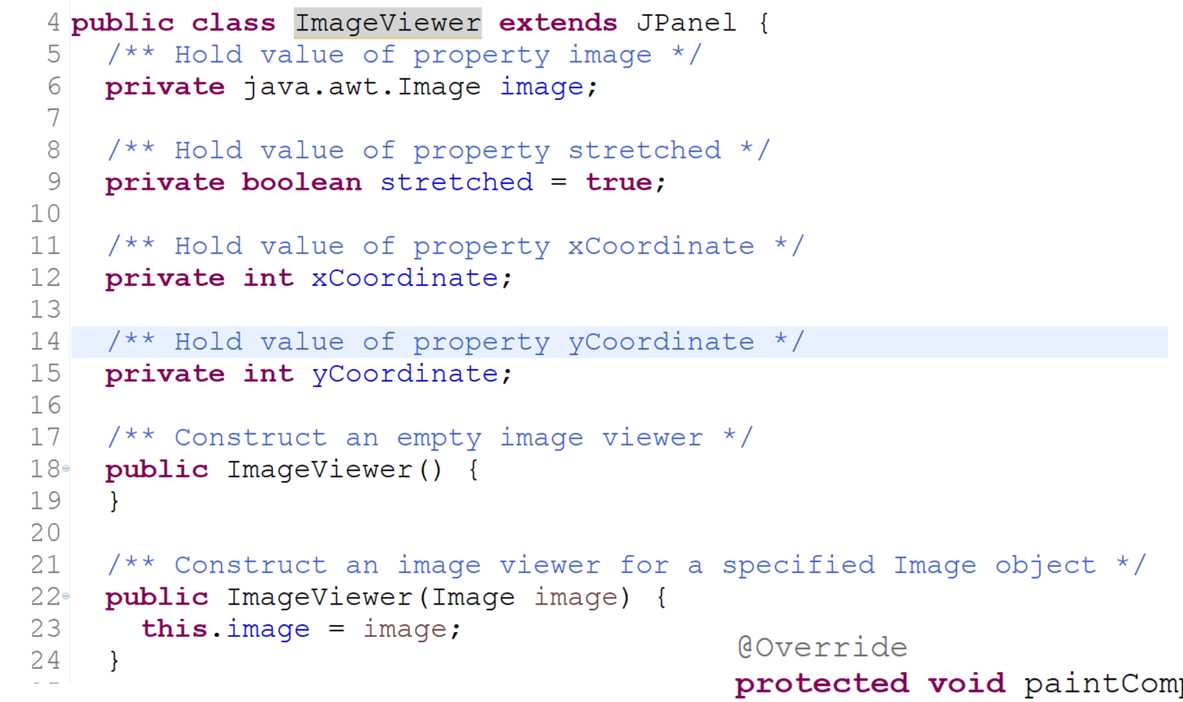
CustomTableCellRenderEditorDemo

Run

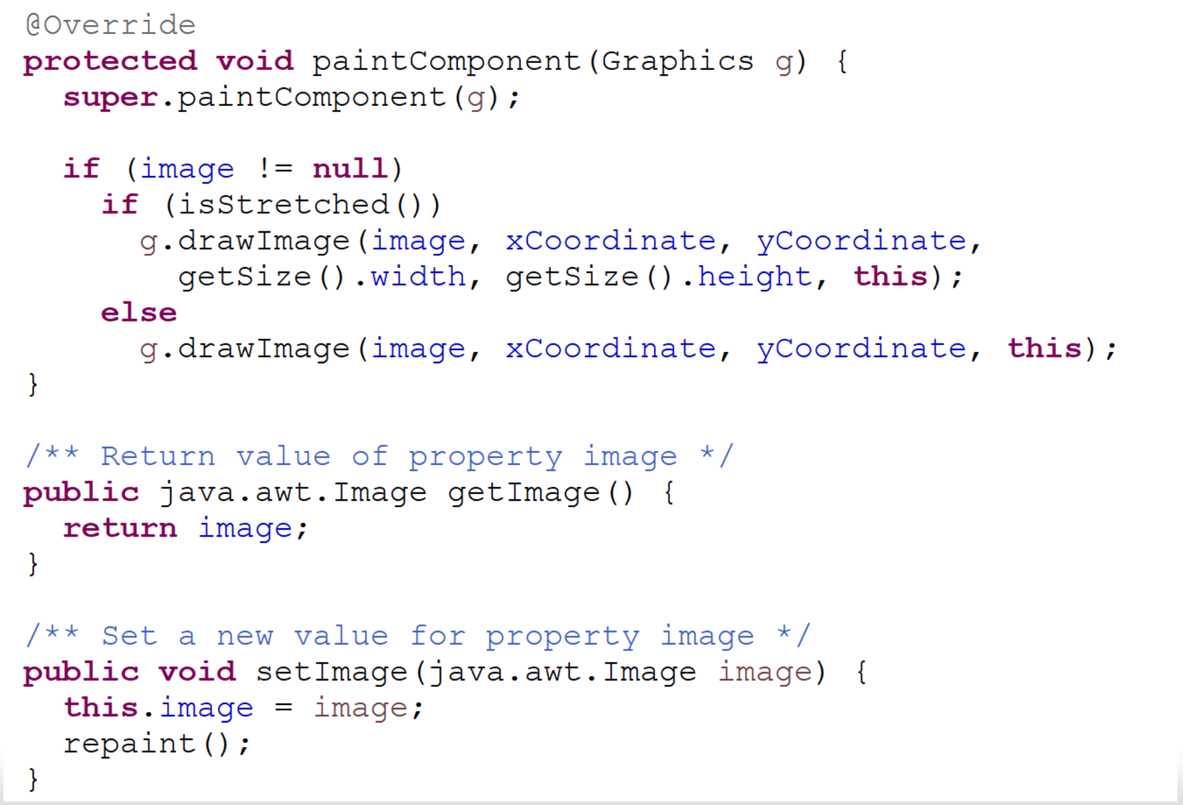
```java
// Set custom renderer for displaying images
TableColumn bookCover = jTable1.getColumn("Book Photo");
bookCover.setCellRenderer(new MyImageCellRenderer());
```

```java
// Create a combo box for publishers
JComboBox jcboPublishers = new JComboBox();
jcboPublishers.addItem("Prentice Hall");
jcboPublishers.addItem("Que Education & Training");
jcboPublishers.addItem("McGraw-Hill");

// Set combo box as the editor for the publisher column
TableColumn publisherColumn = jTable1.getColumn("Publisher");
publisherColumn.setCellEditor(
  new DefaultCellEditor(jcboPublishers));
```

```java
public class MyImageCellRenderer extends DefaultTableCellRenderer {
  /** Override this method in DefaultTableCellRenderer */
  public Component getTableCellRendererComponent
      (JTable table, Object value, boolean isSelected,
       boolean isFocused, int row, int column) {
    Image image = ((ImageIcon)value).getImage();
    ImageViewer imageViewer = new ImageViewer(image);

    return imageViewer;
  }
}
```



```
public class ImageViewer extends JPanel {
  /** Hold value of property image */
  private java.awt.Image image;

  /** Hold value of property stretched */
  private boolean stretched = true;

  /** Hold value of property xCoordinate */
  private int xCoordinate;

  /** Hold value of property yCoordinate */
  private int yCoordinate;

  /** Construct an empty image viewer */
  public ImageViewer() {
  }

  /** Construct an image viewer for a specified Image object */
  public ImageViewer(Image image) {
    this.image = image;
  }
```



```
@Override
protected void paintComponent(Graphics g) {
  super.paintComponent(g);

  if (image != null)
    if (isStretched())
      g.drawImage(image, xCoordinate, yCoordinate,
        getSize().width, getSize().height, this);
    else
      g.drawImage(image, xCoordinate, yCoordinate, this);
}

/** Return value of property image */
public java.awt.Image getImage() {
  return image;
}

/** Set a new value for property image */
public void setImage(java.awt.Image image) {
  this.image = image;
  repaint();
}
```

Table Events

JTable does not fire table events.

It fires the events such as MouseEvent, KeyEvent, and ComponentEvent inherited from its superclass JComponent.

Table events are fired by table models, table column models, and table-selection models whenever changes are made to these models.

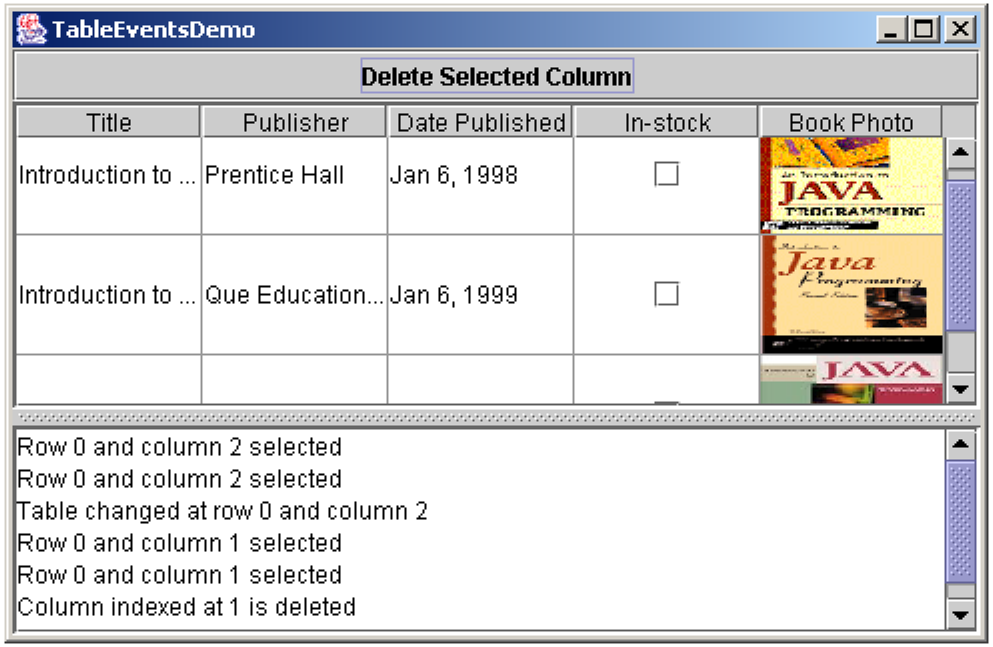
Table models fire TableModelEvent when table data are changed.

Table column models fire TableColumnModelEvent when columns are added, removed, or moved, or when the column selection changes.

Table-selection models fire ListSelectionEvent when the selection changes.

Example: Using Table Events

Problem: This example demonstrates handling table events. The program displays messages on a text area when a row or a column is selected, when a cell is edited, or when a column is removed.

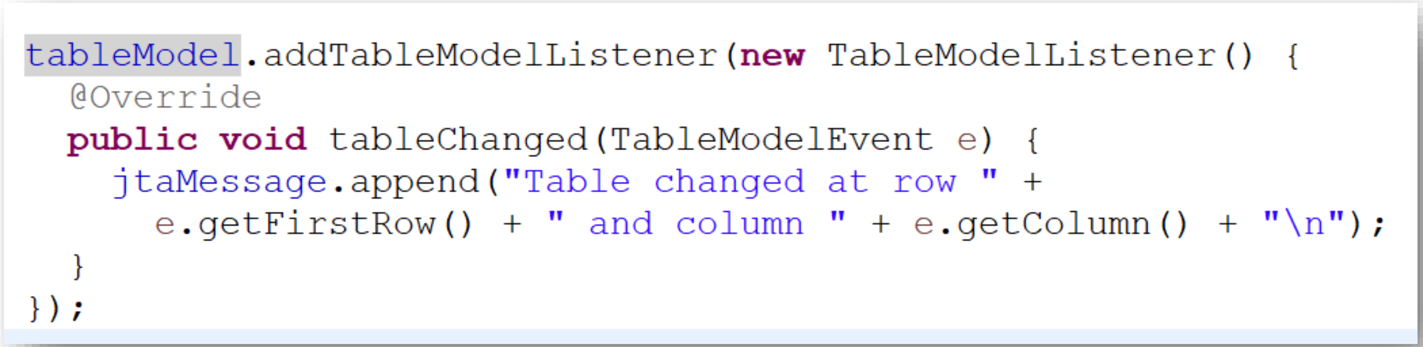


Title	Publisher	Date Published	In-stock	Book Photo
Introduction to ...	Prentice Hall	Jan 6, 1998	☐	
Introduction to ...	Que Education...	Jan 6, 1999	☐	

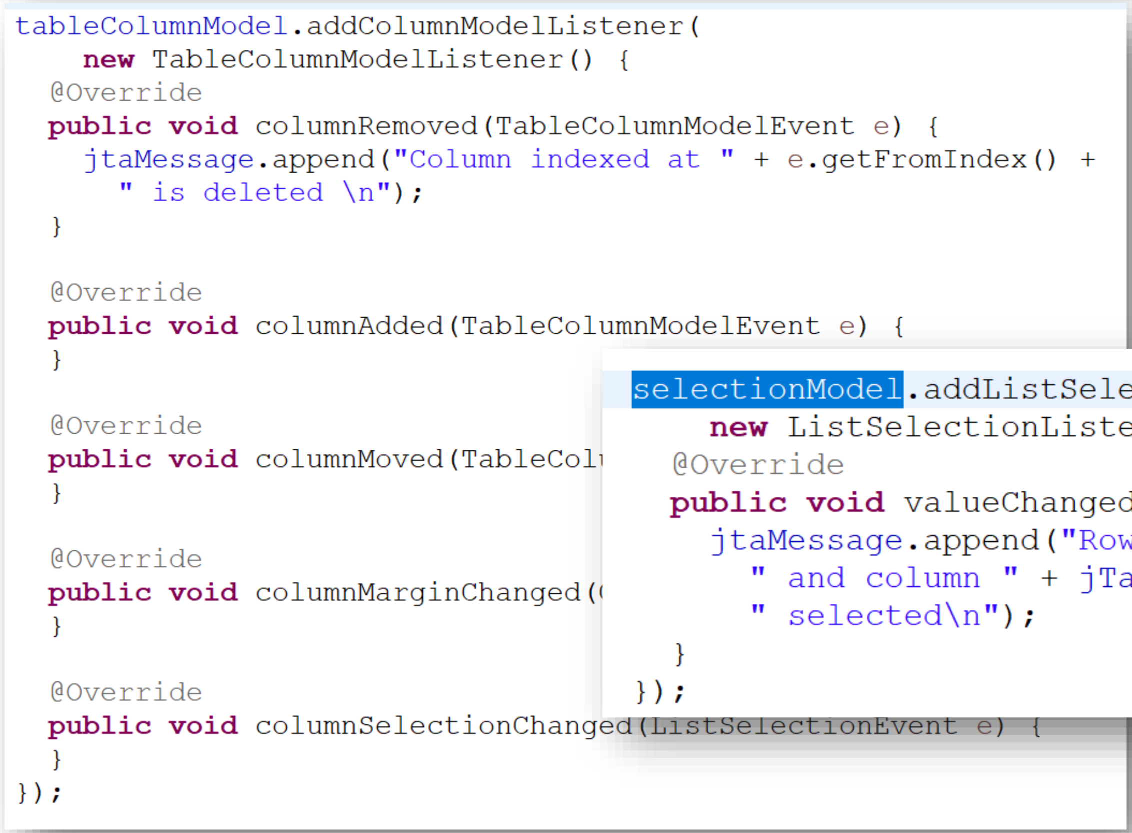
```
Row 0 and column 2 selected
Row 0 and column 2 selected
Table changed at row 0 and column 2
Row 0 and column 1 selected
Row 0 and column 1 selected
Column indexed at 1 is deleted
```

TableEventDemo

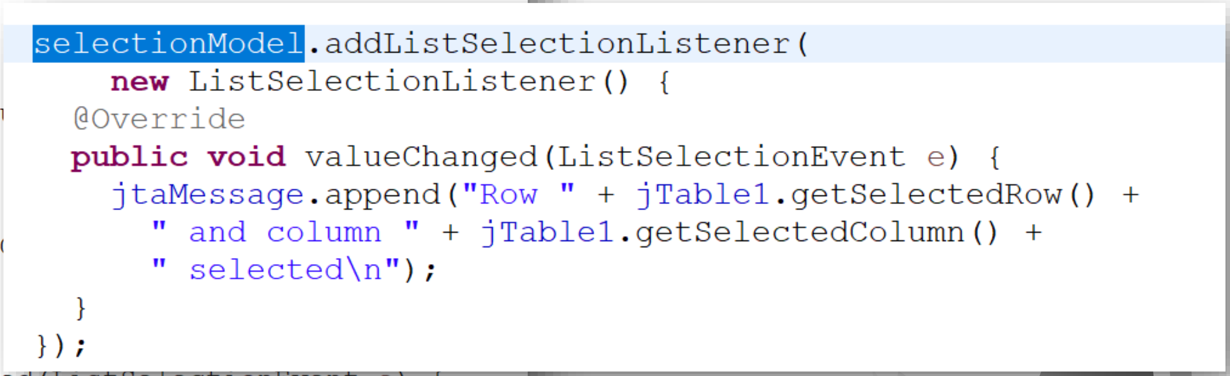
Run



```
tableModel.addTableModelListener(new TableModelListener() {
  @Override
  public void tableChanged(TableModelEvent e) {
    jtaMessage.append("Table changed at row " +
      e.getFirstRow() + " and column " + e.getColumn() + "\n");
  }
});
```



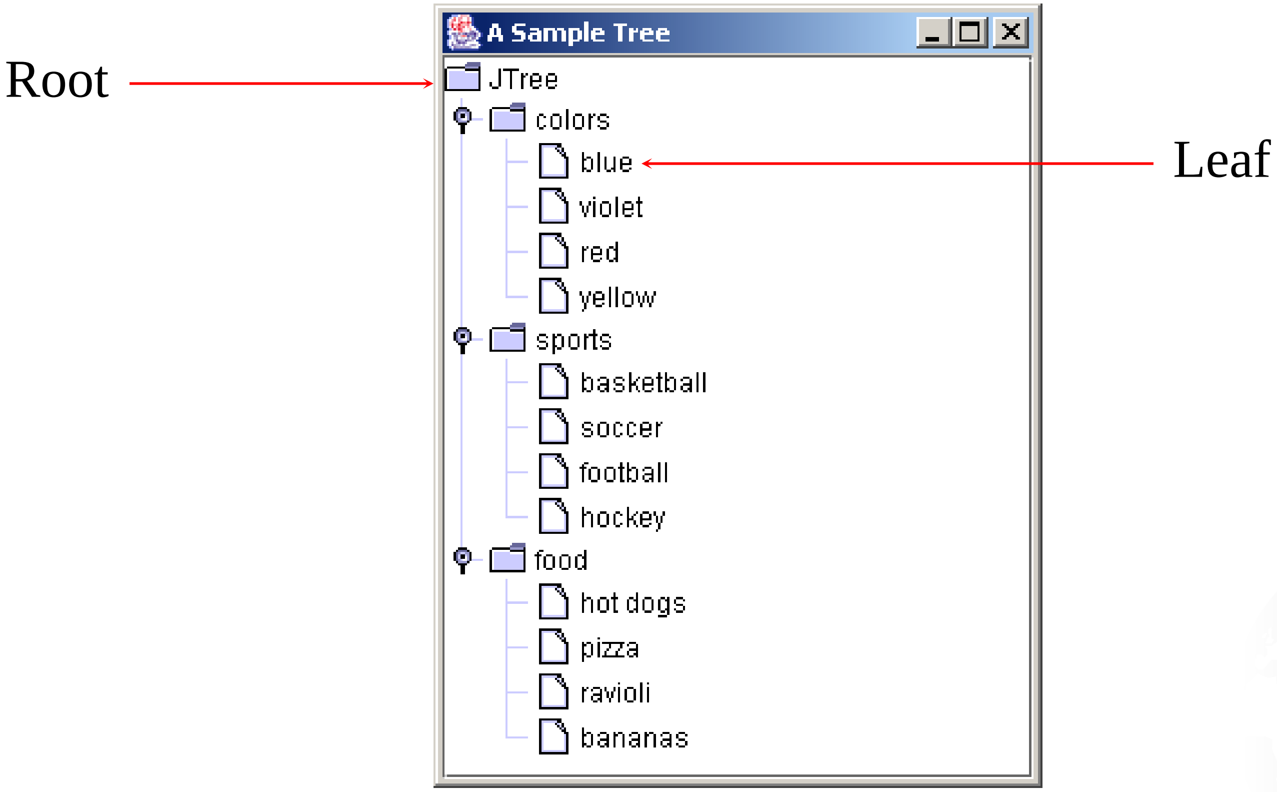
```
tableColumnModel.addColumnModelListener(
    new TableColumnModelListener() {
  @Override
  public void columnRemoved(TableColumnModelEvent e) {
    jtaMessage.append("Column indexed at " + e.getFromIndex() +
      " is deleted \n");
  }

  @Override
  public void columnAdded(TableColumnModelEvent e) {
  }

  @Override
  public void columnMoved(TableColu
  }

  @Override
  public void columnMarginChanged((
  }

  @Override
  public void columnSelectionChanged(ListSelectionEvent e) {
  }
});
```



```
selectionModel.addListSelectionListener(
    new ListSelectionListener() {
  @Override
  public void valueChanged(ListSelectionEvent e) {
    jtaMessage.append("Row " + jTable1.getSelectedRow() +
      " and column " + jTable1.getSelectedColumn() +
      " selected\n");
  }
});
```

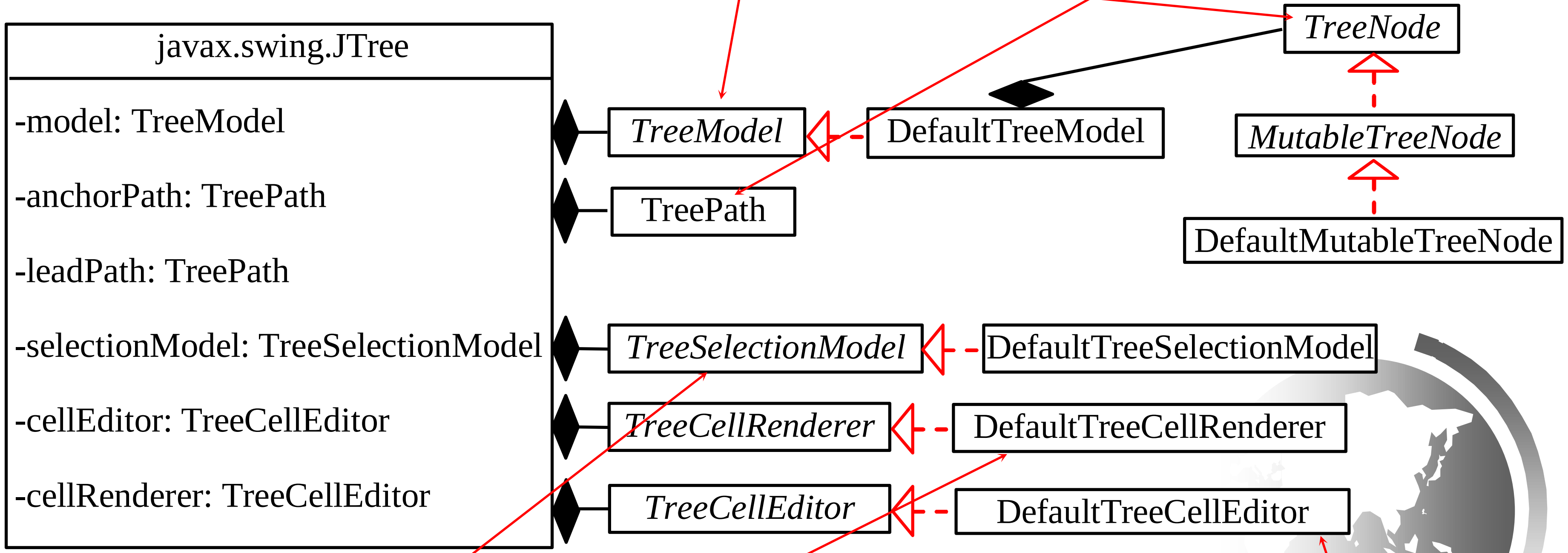
JTree

JTree is a Swing component that displays data in a treelike hierarchy.



Tree Models

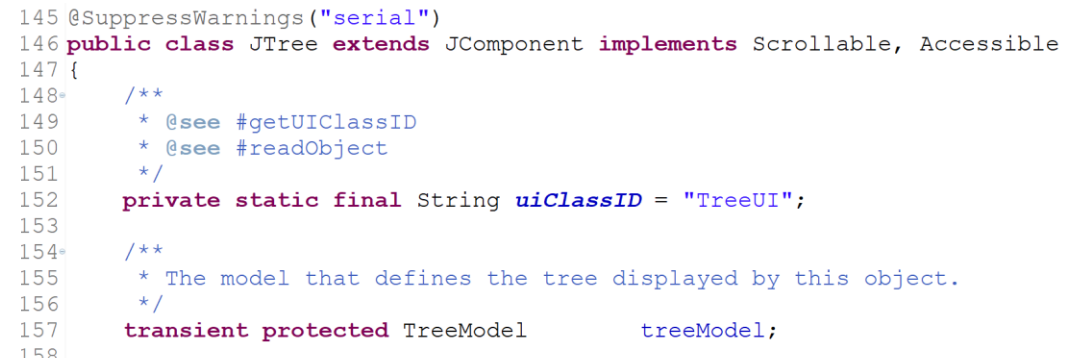
While JTree displays the tree, the data representation of the tree is handled by TreeModel, TreeNode, and TreePath. TreeModel represents the entire tree, TreeNode represents a node, and TreePath represents a path to a node. Unlike the ListModel or TableModel, **the tree model does not directly store or manage tree data**. Tree data are stored and managed in TreeNode and TreePath.



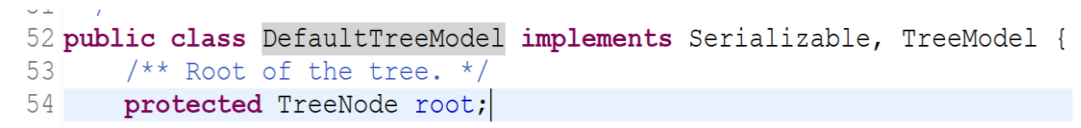
The TreeSelectionModel interface handles tree node selection.

The DefaultTreeCellRenderer class provides a default tree node renderer that can display a label and/or an icon in a node.

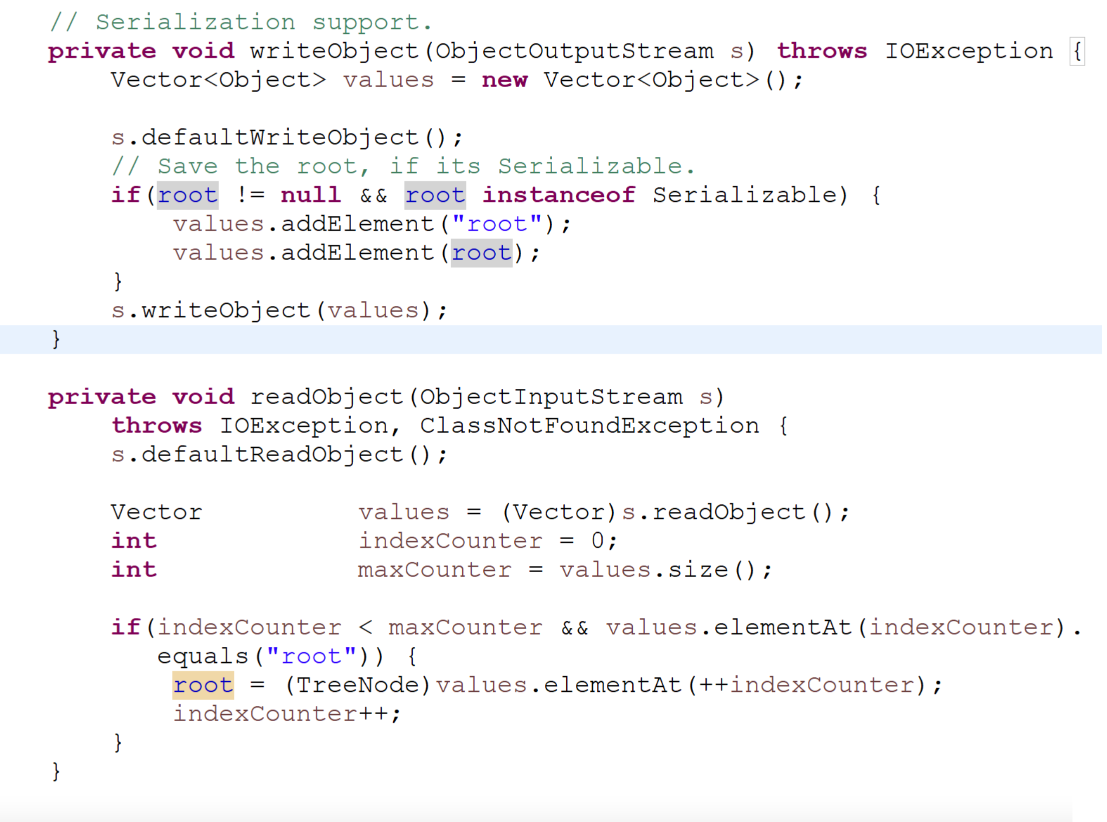
The DefaultTreeCellEditor can be used to edit the cells in a text field.



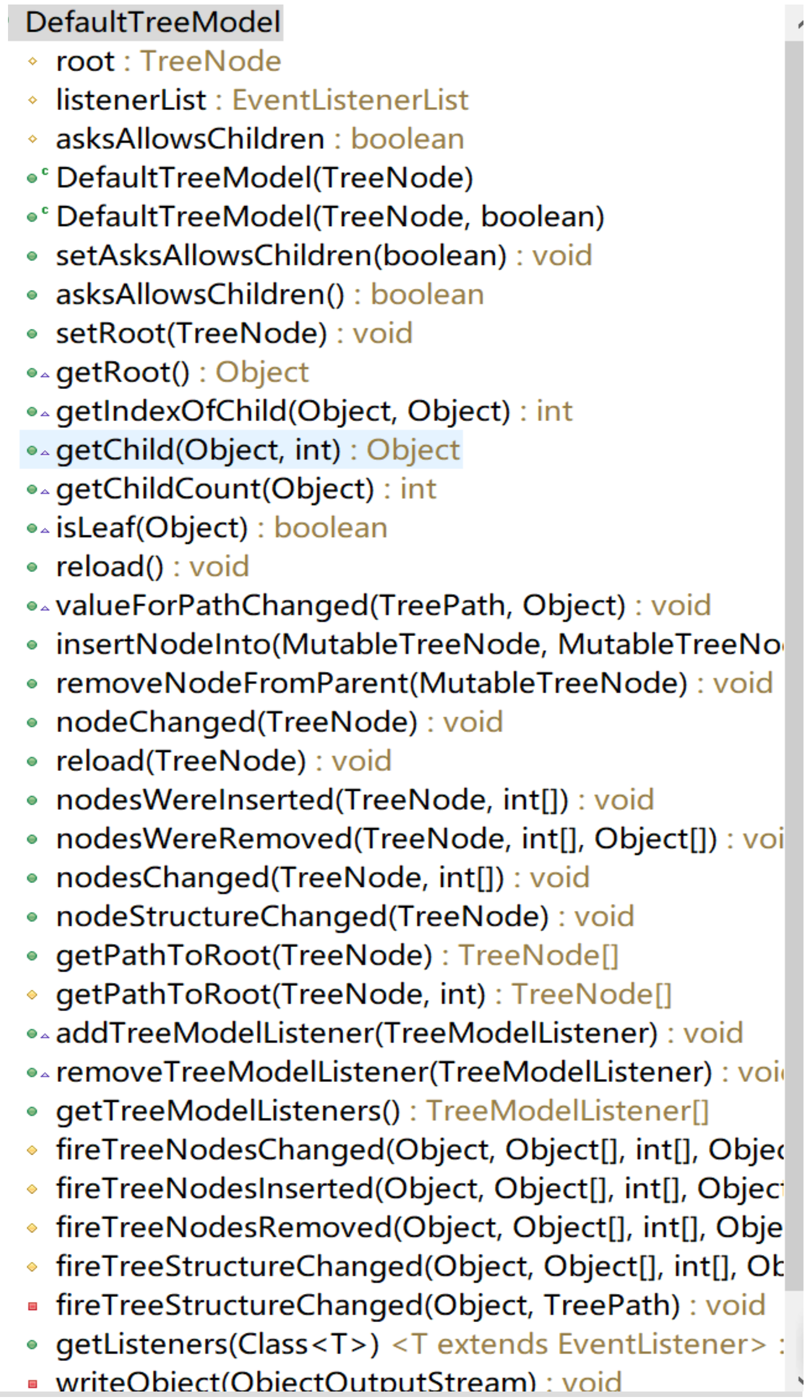
```java
@SuppressWarnings("serial")
public class JTree extends JComponent implements Scrollable, Accessible
{
    /**
     * @see #getUIClassID
     * @see #readObject
     */
    private static final String uiClassID = "TreeUI";

    /**
     * The model that defines the tree displayed by this object.
     */
    transient protected TreeModel        treeModel;
```



```java
public class DefaultTreeModel implements Serializable, TreeModel {
    /** Root of the tree. */
    protected TreeNode root;
```



```java
// Serialization support.
private void writeObject(ObjectOutputStream s) throws IOException {
    Vector<Object> values = new Vector<Object>();

    s.defaultWriteObject();
    // Save the root, if its Serializable.
    if(root != null && root instanceof Serializable) {
        values.addElement("root");
        values.addElement(root);
    }
    s.writeObject(values);
}

private void readObject(ObjectInputStream s)
    throws IOException, ClassNotFoundException {
    s.defaultReadObject();

    Vector          values = (Vector)s.readObject();
    int             indexCounter = 0;
    int             maxCounter = values.size();

    if(indexCounter < maxCounter && values.elementAt(indexCounter).
       equals("root")) {
        root = (TreeNode)values.elementAt(++indexCounter);
        indexCounter++;
    }
}
```



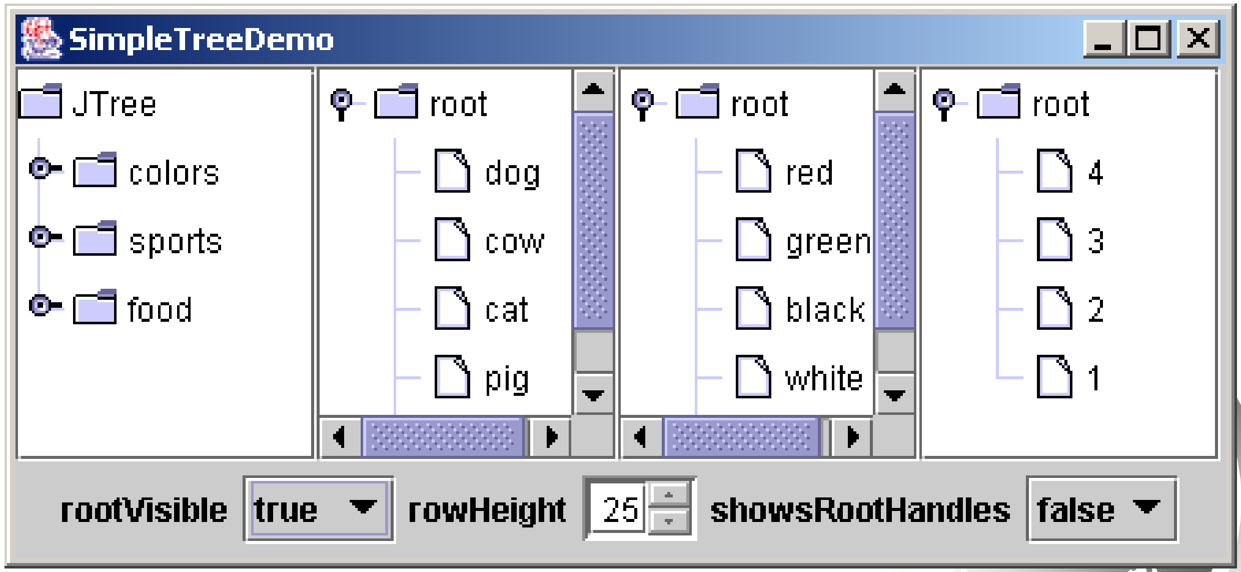
DefaultTreeModel
- root : TreeNode
- listenerList : EventListenerList
- asksAllowsChildren : boolean
- DefaultTreeModel(TreeNode)
- DefaultTreeModel(TreeNode, boolean)
- setAsksAllowsChildren(boolean) : void
- asksAllowsChildren() : boolean
- setRoot(TreeNode) : void
- getRoot() : Object
- getIndexOfChild(Object, Object) : int
- getChild(Object, int) : Object
- getChildCount(Object) : int
- isLeaf(Object) : boolean
- reload() : void
- valueForPathChanged(TreePath, Object) : void
- insertNodeInto(MutableTreeNode, MutableTreeNo
- removeNodeFromParent(MutableTreeNode) : void
- nodeChanged(TreeNode) : void
- reload(TreeNode) : void
- nodesWereInserted(TreeNode, int[]) : void
- nodesWereRemoved(TreeNode, int[], Object[]) : voi
- nodesChanged(TreeNode, int[]) : void
- nodeStructureChanged(TreeNode) : void
- getPathToRoot(TreeNode) : TreeNode[]
- getPathToRoot(TreeNode, int) : TreeNode[]
- addTreeModelListener(TreeModelListener) : void
- removeTreeModelListener(TreeModelListener) : voi
- getTreeModelListeners() : TreeModelListener[]
- fireTreeNodesChanged(Object, Object[], int[], Objec
- fireTreeNodesInserted(Object, Object[], int[], Objec
- fireTreeNodesRemoved(Object, Object[], int[], Obje
- fireTreeStructureChanged(Object, Object[], int[], Ob
- fireTreeStructureChanged(Object, TreePath) : void
- getListeners(Class<T>) <T extends EventListener> :
- writeObject(ObjectOutputStream) : void

The JTree Class

javax.swing.JTree	
#cellEditor: TreeCellEditor	Specifies a cell editor used to edit entries in the tree.
#cellRenderer: TreeCellRenderer	Specifies whether individual cells can be selected (Obsolete since JDK 1.3).
#editable: boolean	Specifies whether the cells are editable (default: false).
#model: TreeModel	Maintains the tree model.
#rootVisible: boolean	Specifies whether the root is displayed (depending on the constructor).
#rowHeight: int	Specifies the height of the row for the node displayed in the tree (default: 16 pixels).
#scrollsOnExpand: boolean	If true, when a node is expanded, as many of the descendants are scrolled to be visible (default: 16 pixels).
#selectionModel: TreeSelectionModel	Models the set of selected nodes in this tree.
#showsRootHandles: boolean	Specifies whether the root handles are displayed (default: true).
#toggleClickCount: int	Number of mouse clicks before a node is expanded (default: 2).
-anchorSelectionPath: TreePath	The path identified as the anchor.
-expandsSelectedPaths: boolean	True if paths in the selection should be expanded (default: true).
-leadSelectionPaths: TreePath	The path identified as the lead.
+JTree()	Creates a JTree with a sample tree model, as shown in Figure 24.35.
+JTree(value: java.util.Hashtable)	Creates a JTree with an invisible root and the keys in the Hashtable key/value pairs as its children.
+JTree(value: Object[])	Creates a JTree with an invisible root and the elements in the array as its children.
+JTree(newModel: TreeModel)	Creates a JTree with the specified tree model.
+JTree(root: TreeNode)	Creates a JTree with the specified tree node as its root.
+JTree(root: TreeNode, asksAllowsChildren: boolean)	Creates a JTree with the specified tree node as its root and decides whether a node is a leaf node in the specified manner.
+JTree(value: Vector)	Creates a JTree with an invisible root and the elements in the vector as its children.
+addSelectionPath(path: TreePath): void	Adds the specified TreePath to the current selection.
+addSelectionPaths(paths: TreePath[]): void	Adds the specified TreePaths to the current selection.
+addSelectionRow(row: int): void	Adds the path at the specified row to the current selection.
+addSelectionRows(rows: int[]): void	Adds the path at the specified rows to the current selection.
+clearSelection() : void	Clears the selection.
+collapsePath(path: TreePath): void	Ensures that the node identified by the specified path is collapsed and viewable.
+getSelectionPath(): TreePath	Returns the path from the root to the first selected node.
+getSelectionPaths(): TreePath[]	Returns the paths from the root to all the selected nodes.
+getLastSelectedPathComponent()	Returns the last node in the first selected TreePath.
+getRowCount():int	Returns the number of rows currently being displayed.
+removeSelectionPath(path: TreePath): void	Removes the node in the specified path.
+removeSelectionPaths(paths: TreePath[]):void	Removes the node in the specified paths.

Example: Simple Tree Demo

Problem: Write a program to create four trees: a default tree using the no-arg constructor, a tree created from an array of objects, a tree created from a vector, and a tree created from a hash table. Enable the user to dynamically set the properties for rootVisible, rowHeight, and showsRootHandles.



rootVisible true rowHeight 25 showsRootHandles false

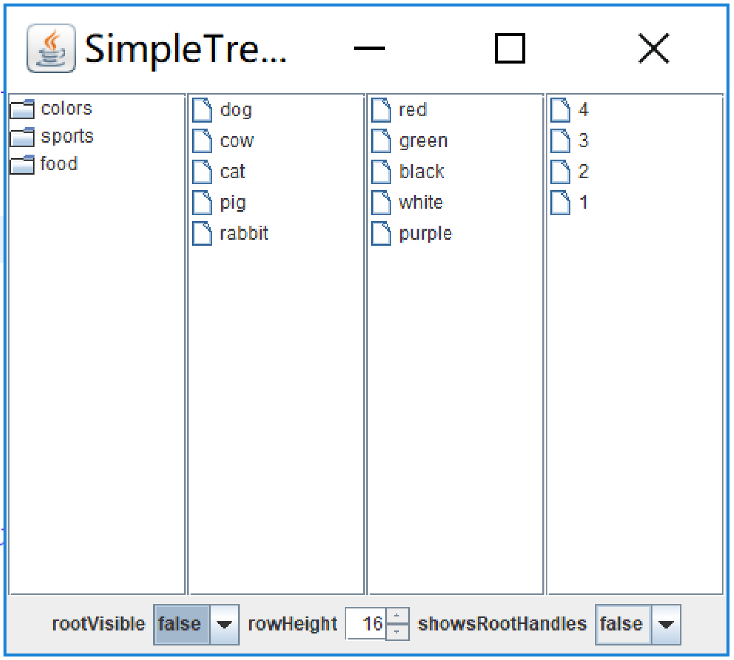
SimpleTreeDemo

Run

```
boolean rootVisible =
    jcboRootVisible.getSelectedItem().equals("tr
jTree1.setRootVisible(rootVisible);
jTree2.setRootVisible(rootVisible);
jTree3.setRootVisible(rootVisible);
jTree4.setRootVisible(rootVisible);
```

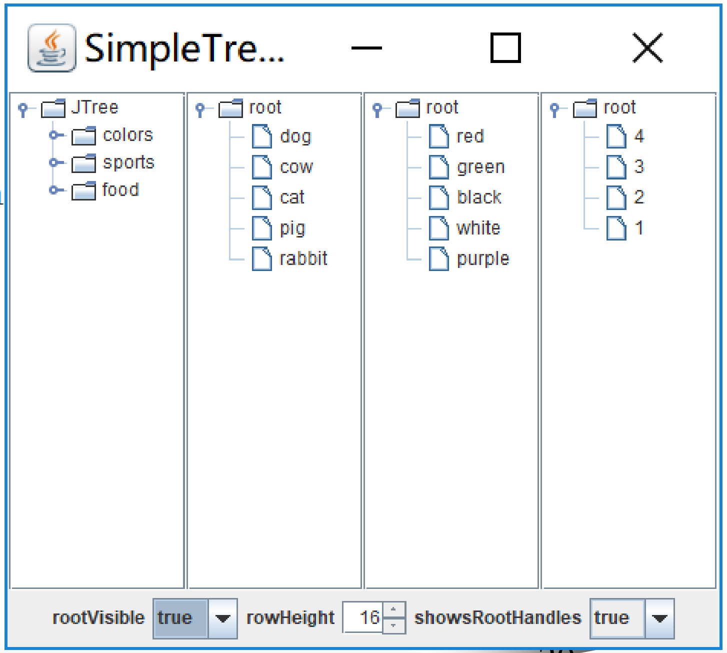
```
boolean showsRootHandles =
    jcboShowsRootHandles.getSelectedItem().equals("tru
jTree1.setShowsRootHandles(showsRootHandles);
jTree2.setShowsRootHandles(showsRootHandles);
jTree3.setShowsRootHandles(showsRootHandles);
jTree4.setShowsRootHandles(showsRootHandles);
```

```
int height =
    ((Integer)(jSpinnerRowHeight.getValue())).intVa
jTree1.setRowHeight(height);
jTree2.setRowHeight(height);
jTree3.setRowHeight(height);
jTree4.setRowHeight(height);
```



```
colors    dog      red      4
sports    cow      green    3
food      cat      black    2
          pig      white    1
          rabbit   purple
```

rootVisible false rowHeight 16 showsRootHandles false



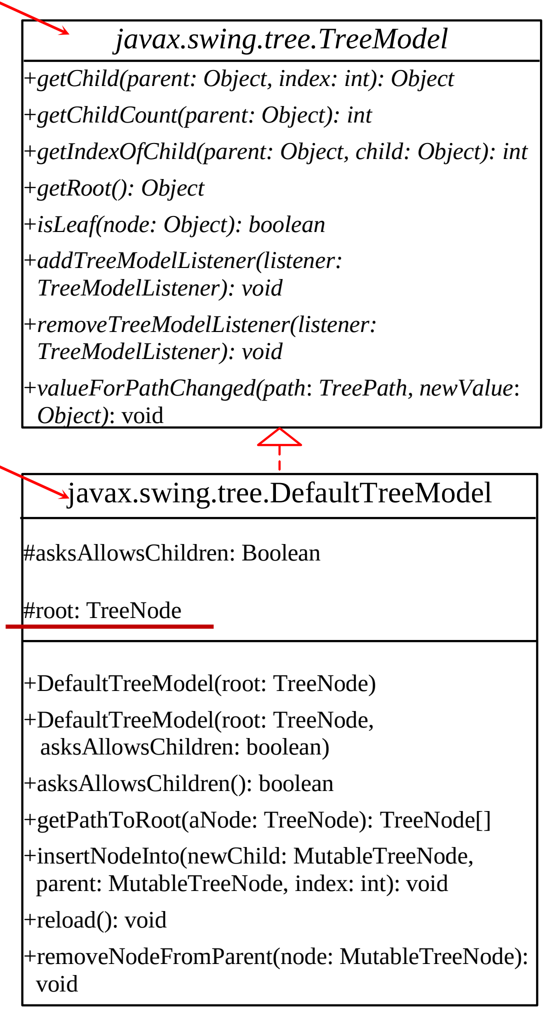
```
JTree        root        root        root
  colors       dog         red         4
  sports       cow         green       3
  food         cat         black       2
               pig         white       1
               rabbit      purple
```

rootVisible true rowHeight 16 showsRootHandles true

TreeModel and DefaultTreeModel

TreeModel contains the structural information about the tree, and tree data are stored and managed by TreeNode.

DefaultTreeModel is a concrete implementation for TreeModel that uses TreeNodes.



javax.swing.tree.TreeModel	
+getChild(parent: Object, index: int): Object	Returns the child of parent at the index in the parent's child array.
+getChildCount(parent: Object): int	Returns the number of children of the specified parent in the tree model.
+getIndexOfChild(parent: Object, child: Object): int	Returns the index of child in parent. If parent or child is null, returns –1.
+getRoot(): Object	Returns the root of the tree. Returns null if the tree is empty.
+isLeaf(node: Object): boolean	Returns true if the specified node is a leaf.
+addTreeModelListener(listener: TreeModelListener): void	Adds a listener for the TreeModelEvent posted after the tree changes.
+removeTreeModelListener(listener: TreeModelListener): void	Removes a listener previously added with addTreeModelListener.
+valueForPathChanged(path: TreePath, newValue: Object): void	Messaged when the user has altered the value for the item identified by path to newValue.

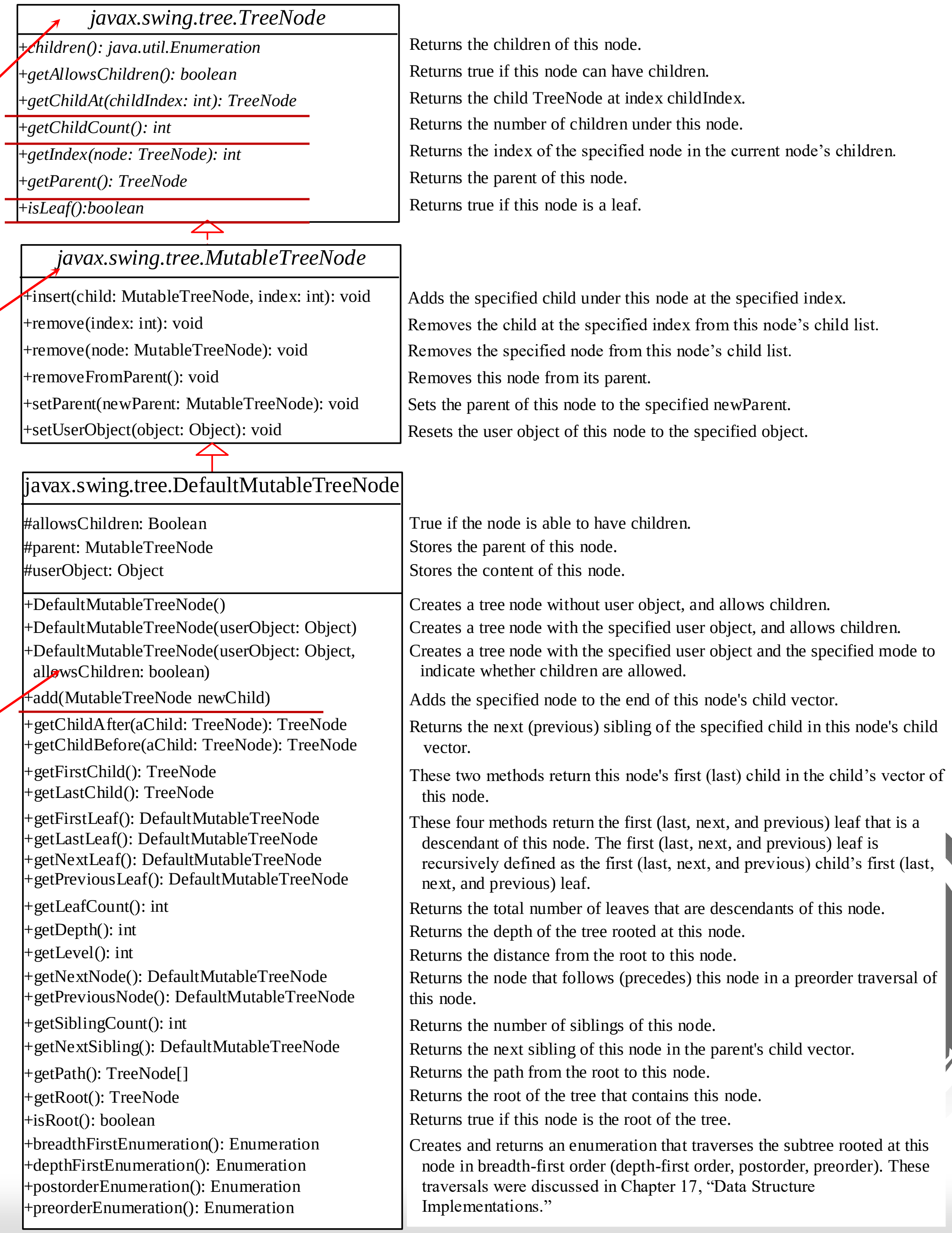
javax.swing.tree.DefaultTreeModel	
#asksAllowsChildren: Boolean	Tells how leaf nodes are determined. True if only nodes that do not allow children are leaf nodes, false if nodes that have no children are leaf nodes.
#root: TreeNode	The root of the tree.
+DefaultTreeModel(root: TreeNode)	Creates a DefaultTreeModel with the specified root.
+DefaultTreeModel(root: TreeNode, asksAllowsChildren: boolean)	Creates a DefaultTreeModel with the specified root and decides whether a node is a leaf node in the specified manner.
+asksAllowsChildren(): boolean	Returns asksAllowsChildren.
+getPathToRoot(aNode: TreeNode): TreeNode[]	Returns the nodes in an array from root to the specified node.
+insertNodeInto(newChild: MutableTreeNode, parent: MutableTreeNode, index: int): void	Inserts newChild at location index in parents children.
+reload(): void	Reloads the model (invoke this method if the tree has been modified)
+removeNodeFromParent(node: MutableTreeNode): void	Removes the node from its parent.

TreeNode, MutableTreeNode, and DefaultMutableTreeNode

TreeNode stores models a single node in the tree.

MutableTreeNode defines a subinterface of TreeNode with additional methods for changing the content of the node, for inserting and removing a child node, for setting a new parent, and for removing the node itself.

DefaultMutableTreeNode is a concrete implementation of MutableTreeNode.



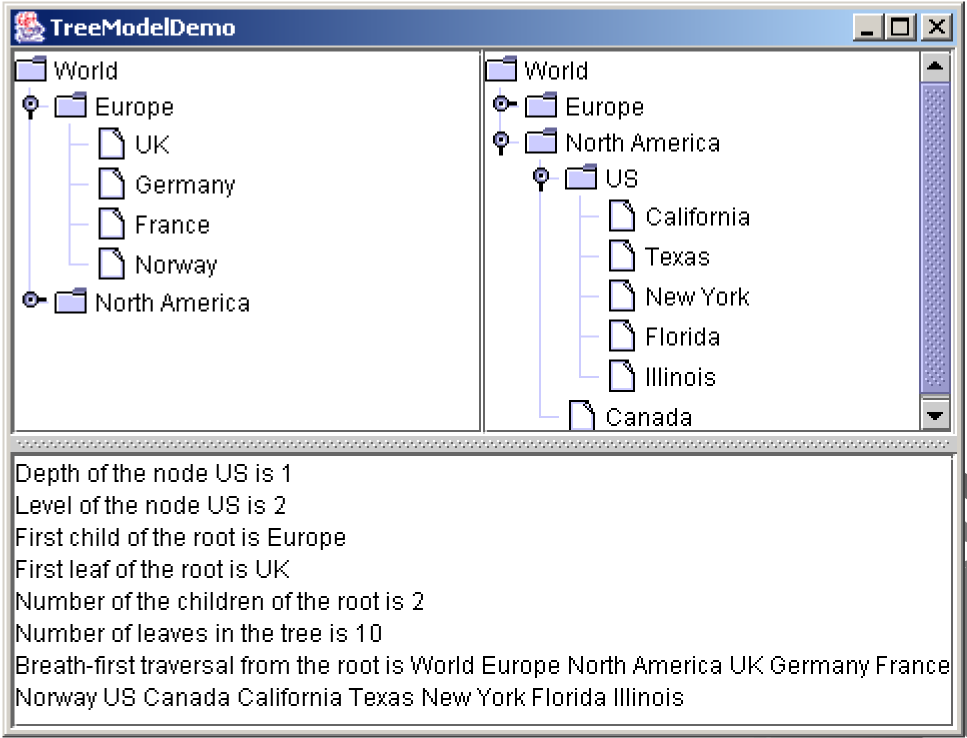
javax.swing.tree.TreeNode	
+children(): java.util.Enumeration	Returns the children of this node.
+getAllowsChildren(): boolean	Returns true if this node can have children.
+getChildAt(childIndex: int): TreeNode	Returns the child TreeNode at index childIndex.
+getChildCount(): int	Returns the number of children under this node.
+getIndex(node: TreeNode): int	Returns the index of the specified node in the current node's children.
+getParent(): TreeNode	Returns the parent of this node.
+isLeaf():boolean	Returns true if this node is a leaf.

javax.swing.tree.MutableTreeNode	
+insert(child: MutableTreeNode, index: int): void	Adds the specified child under this node at the specified index.
+remove(index: int): void	Removes the child at the specified index from this node's child list.
+remove(node: MutableTreeNode): void	Removes the specified node from this node's child list.
+removeFromParent(): void	Removes this node from its parent.
+setParent(newParent: MutableTreeNode): void	Sets the parent of this node to the specified newParent.
+setUserObject(object: Object): void	Resets the user object of this node to the specified object.

javax.swing.tree.DefaultMutableTreeNode	
#allowsChildren: Boolean	True if the node is able to have children.
#parent: MutableTreeNode	Stores the parent of this node.
#userObject: Object	Stores the content of this node.
+DefaultMutableTreeNode()	Creates a tree node without user object, and allows children.
+DefaultMutableTreeNode(userObject: Object)	Creates a tree node with the specified user object, and allows children.
+DefaultMutableTreeNode(userObject: Object, allowsChildren: boolean)	Creates a tree node with the specified user object and the specified mode to indicate whether children are allowed.
+add(MutableTreeNode newChild)	Adds the specified node to the end of this node's child vector.
+getChildAfter(aChild: TreeNode): TreeNode +getChildBefore(aChild: TreeNode): TreeNode	Returns the next (previous) sibling of the specified child in this node's child vector.
+getFirstChild(): TreeNode +getLastChild(): TreeNode	These two methods return this node's first (last) child in the child's vector of this node.
+getFirstLeaf(): DefaultMutableTreeNode +getLastLeaf(): DefaultMutableTreeNode +getNextLeaf(): DefaultMutableTreeNode +getPreviousLeaf(): DefaultMutableTreeNode	These four methods return the first (last, next, and previous) leaf that is a descendant of this node. The first (last, next, and previous) leaf is recursively defined as the first (last, next, and previous) child's first (last, next, and previous) leaf.
+getLeafCount(): int	Returns the total number of leaves that are descendants of this node.
+getDepth(): int	Returns the depth of the tree rooted at this node.
+getLevel(): int	Returns the distance from the root to this node.
+getNextNode(): DefaultMutableTreeNode +getPreviousNode(): DefaultMutableTreeNode	Returns the node that follows (precedes) this node in a preorder traversal of this node.
+getSiblingCount(): int	Returns the number of siblings of this node.
+getNextSibling(): DefaultMutableTreeNode	Returns the next sibling of this node in the parent's child vector.
+getPath(): TreeNode[]	Returns the path from the root to this node.
+getRoot(): TreeNode	Returns the root of the tree that contains this node.
+isRoot(): boolean	Returns true if this node is the root of the tree.
+breadthFirstEnumeration(): Enumeration +depthFirstEnumeration(): Enumeration +postorderEnumeration(): Enumeration +preorderEnumeration(): Enumeration	Creates and returns an enumeration that traverses the subtree rooted at this node in breadth-first order (depth-first order, postorder, preorder). These traversals were discussed in Chapter 17, "Data Structure Implementations."

Example: Tree Model Demo

Problem: Write a program to create two trees that displays world, continents, countries and states. The two trees display identical contents. The program also displays the properties of the tree in a text area.

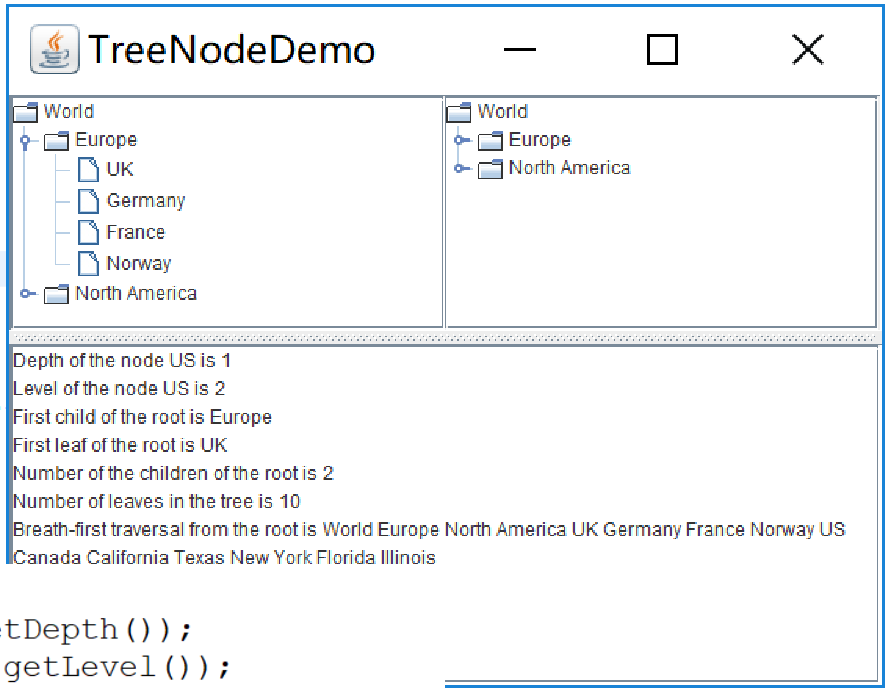


```
Depth of the node US is 1
Level of the node US is 2
First child of the root is Europe
First leaf of the root is UK
Number of the children of the root is 2
Number of leaves in the tree is 10
Breath-first traversal from the root is World Europe North America UK Germany France
Norway US Canada California Texas New York Florida Illinois
```

TreeNodeDemo

Run

```java
// Create the first tree
DefaultMutableTreeNode root, europe, northAmerica, us;

europe = new DefaultMutableTreeNode("Europe");
europe.add(new DefaultMutableTreeNode("UK"));
europe.add(new DefaultMutableTreeNode("Germany"));
europe.add(new DefaultMutableTreeNode("France"));
europe.add(new DefaultMutableTreeNode("Norway"));
```



```
Depth of the node US is 1
Level of the node US is 2
First child of the root is Europe
First leaf of the root is UK
Number of the children of the root is 2
Number of leaves in the tree is 10
Breath-first traversal from the root is World Europe North America UK Germany France Norway US
Canada California Texas New York Florida Illinois
```

```java
// Get tree information
jtaMessage.append("Depth of the node US is " + us.getDepth());
jtaMessage.append("\nLevel of the node US is " + us.getLevel());
jtaMessage.append("\nFirst child of the root is " +
  root.getFirstChild());
jtaMessage.append("\nFirst leaf of the root is " +
  root.getFirstLeaf());
jtaMessage.append("\nNumber of the children of the root is " +
  root.getChildCount());
jtaMessage.append("\nNumber of leaves in the tree is " +
  root.getLeafCount());
String breadthFirstSearchResult = "";

// Breadth-first traversal
Enumeration bf = root.breadthFirstEnumeration();
while (bf.hasMoreElements())
  breadthFirstSearchResult += bf.nextElement().toString() + " ";
jtaMessage.append("\nBreath-first traversal from the root is "
  + breadthFirstSearchResult);
```

The TreePath Class

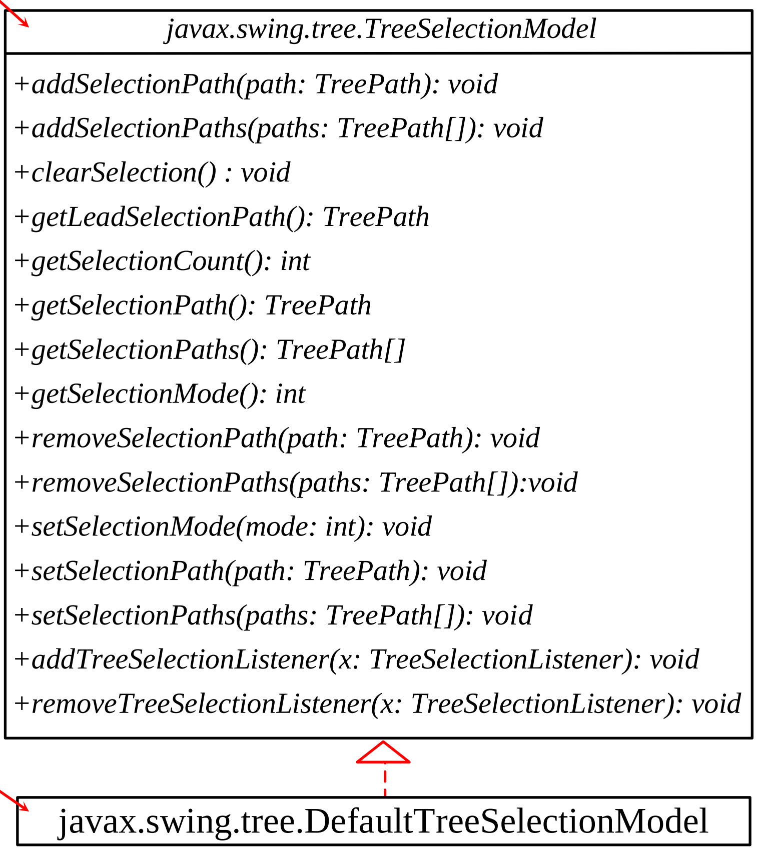
The TreePath class represents a path from an ancestor to a descendant in a tree.

javax.swing.tree.TreePath	
+TreePath(singlePath: Object)	Constructs a TreePath containing only a single element.
+TreePath(path: Object[])	Constructs a path from an array of objects.
+getLastPathComponent(): Object	Returns the last component of this path.
+getParentPath(): TreePath	Returns a path containing all but the last path component.
+getPath(): Object[]	Returns an ordered array of objects containing the components of this TreePath.
+getPathComponent(element: int): Object	Returns the path component at the specified index.
+getPathCount(): int	Returns the number of elements in the path.
+isDescendant(aTreePath: TreePath): Boolean	Returns true if aTreePath contains all the components in this TreePath.
+pathByAddingChild(child: Object): TreePath	Returns a new path containing all the elements of this TreePath plus child.

TreeSelectionModel and DefaultTreeSelectionModel

The selection of tree nodes is defined in the TreeSelectionModel interface.

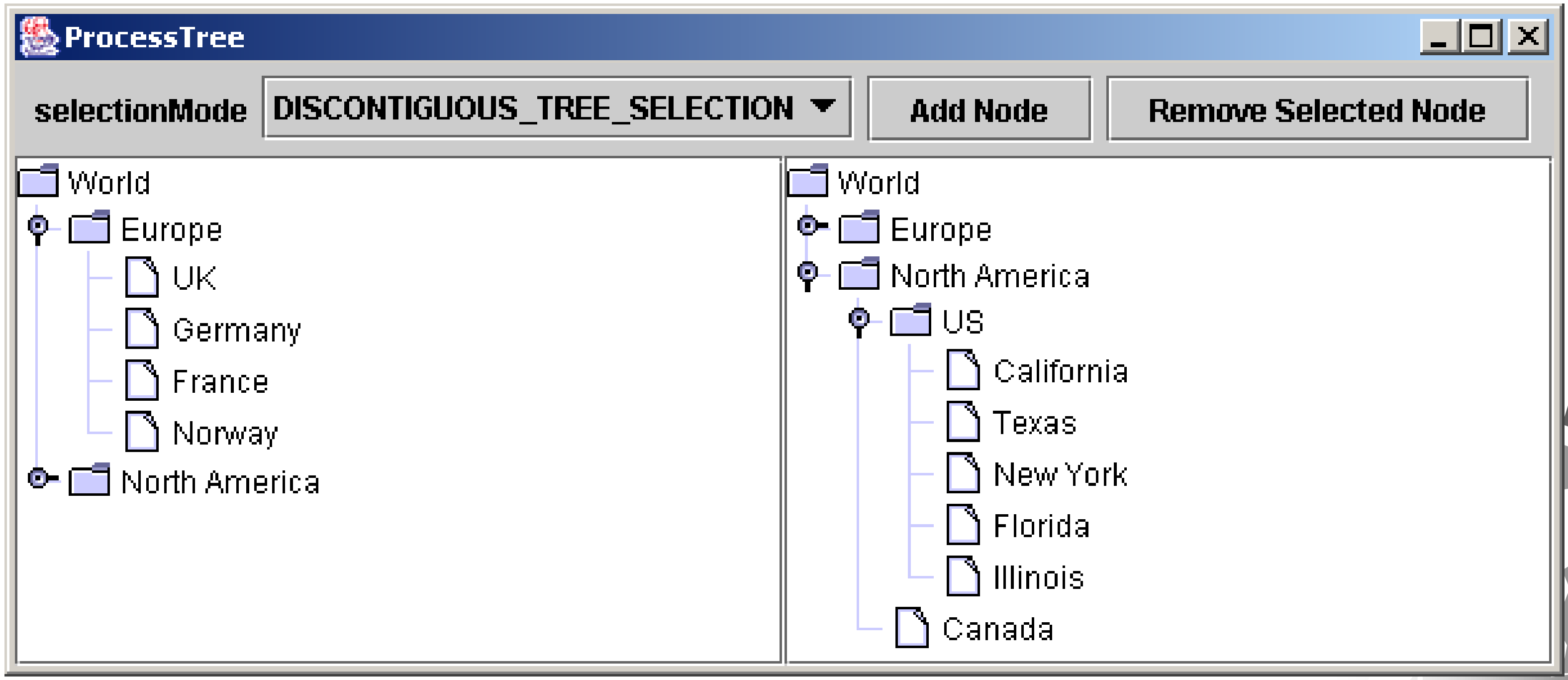
The DefaultTreeSelectionModel class is a concrete implementation of the TreeSelectionModel, which maintains an array of TreePath objects representing the current selection.


javax.swing.tree.TreeSelectionModel
javax.swing.tree.DefaultTreeSelectionModel

+addSelectionPath(path: TreePath): void	Adds the specified TreePath to the current selection.
+addSelectionPaths(paths: TreePath[]): void	Adds the specified TreePaths to the current selection.
+clearSelection() : void	Clears the selection.
+getLeadSelectionPath(): TreePath	Returns the last path in the selection.
+getSelectionCount(): int	Returns the number of paths in the selection.
+getSelectionPath(): TreePath	Returns the first path in the selection.
+getSelectionPaths(): TreePath[]	Returns all the paths in the selection.
+getSelectionMode(): int	Returns the current selection mode,
+removeSelectionPath(path: TreePath): void	Removes path from the selection.
+removeSelectionPaths(paths: TreePath[]):void	Removes paths from the selection.
+setSelectionMode(mode: int): void	Sets the selection mode.
+setSelectionPath(path: TreePath): void	Sets the selection to path.
+setSelectionPaths(paths: TreePath[]): void	Sets the selection to paths.
+addTreeSelectionListener(x: TreeSelectionListener): void	Register a TreeSelectionListener.
+removeTreeSelectionListener(x: TreeSelectionListener): void	Remove a TreeSelectionListener.

Example: Modifying Trees

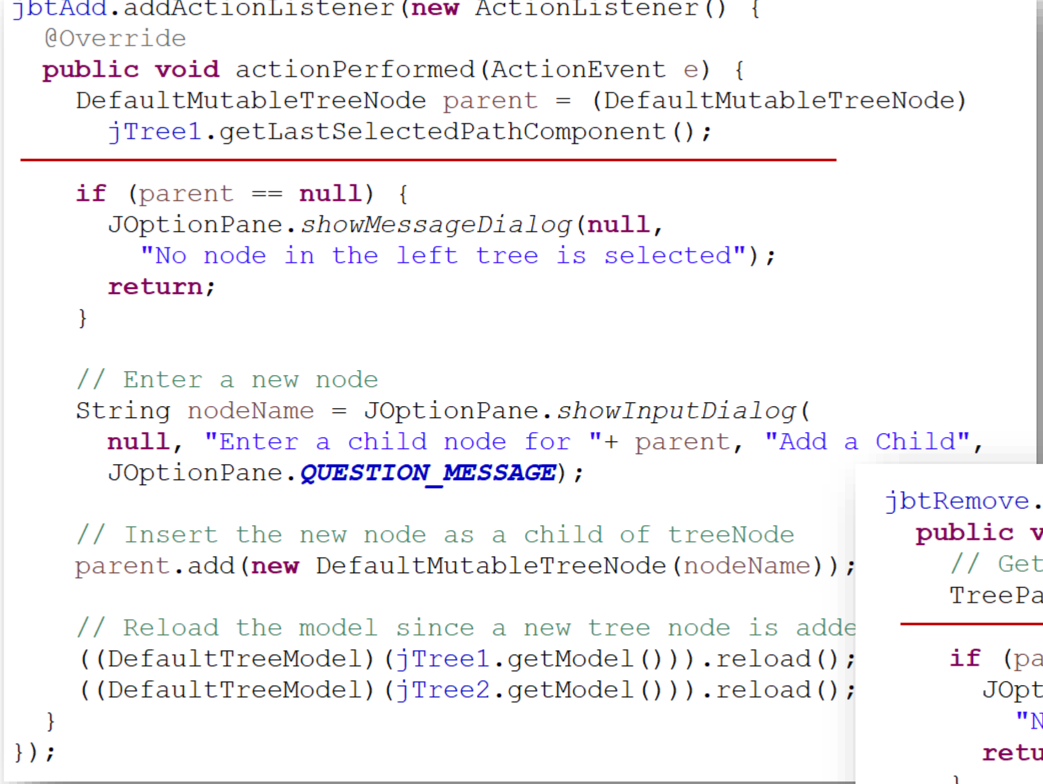
Problem: Write a program to create two trees that displays the same contents: world, continents, countries and states, as shown in Figure 36.44. For the left tree on the left, enable the user to choose a selection mode, add a new child under the first selected node, and remove all the selected nodes.



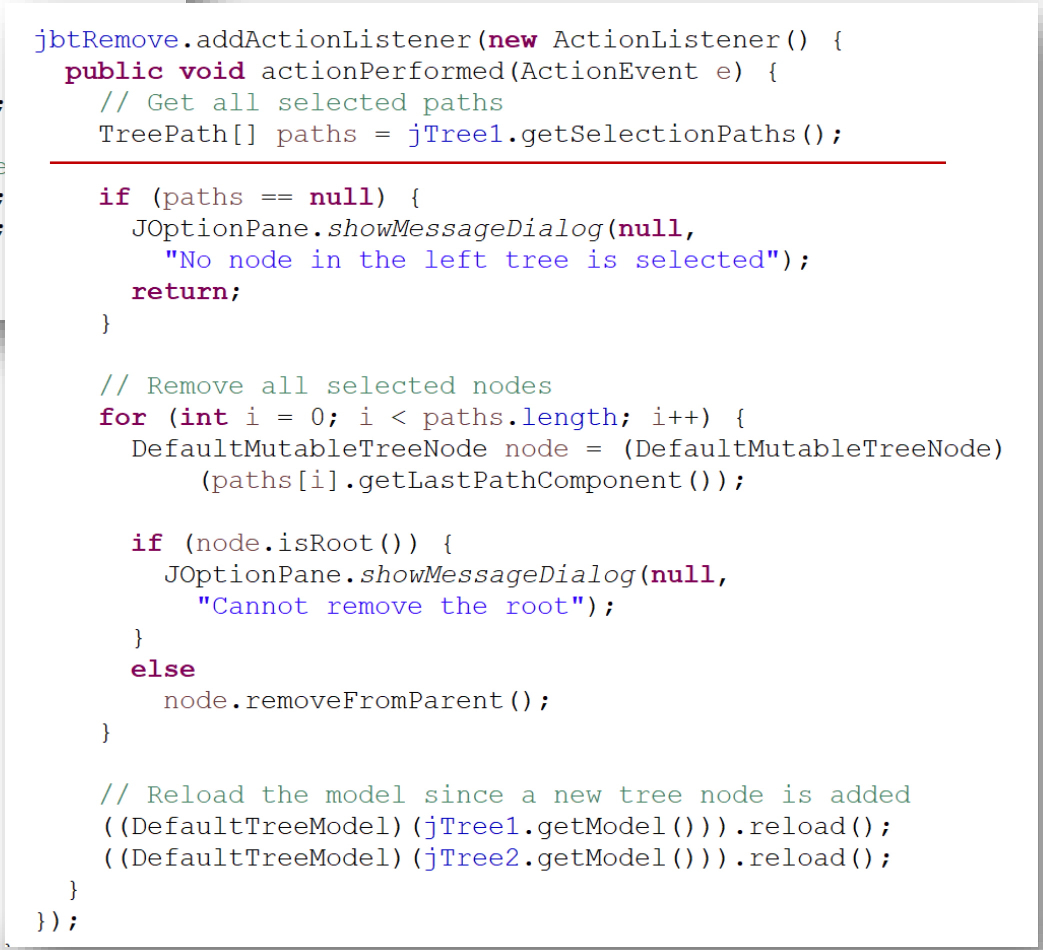
ModifyTree

Run

```
// Register listeners
jcboSelectionMode.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    if (jcboSelectionMode.getSelectedItem().
        equals("CONTIGUOUS_TREE_SELECTION"))
      jTree1.getSelectionModel().setSelectionMode(
        TreeSelectionModel.CONTIGUOUS_TREE_SELECTION);
    else if (jcboSelectionMode.getSelectedItem().
        equals("DISCONTIGUOUS_TREE_SELECTION"))
      jTree1.getSelectionModel().setSelectionMode(
        TreeSelectionModel.DISCONTIGUOUS_TREE_SELECTION);
    else
      jTree1.getSelectionModel().setSelectionMode(
        TreeSelectionModel.SINGLE_TREE_SELECTION);
  }
});

jchkEditable.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    jTree1.setEditable(jchkEditable.isSelected());
  }
});
```



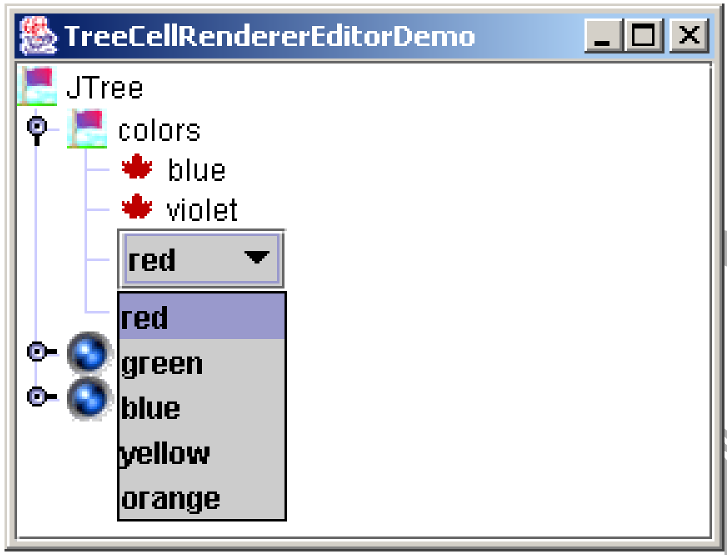
```
jbtAdd.addActionListener(new ActionListener() {
  @Override
  public void actionPerformed(ActionEvent e) {
    DefaultMutableTreeNode parent = (DefaultMutableTreeNode)
      jTree1.getLastSelectedPathComponent();

    if (parent == null) {
      JOptionPane.showMessageDialog(null,
        "No node in the left tree is selected");
      return;
    }

    // Enter a new node
    String nodeName = JOptionPane.showInputDialog(
      null, "Enter a child node for "+ parent, "Add a Child",
      JOptionPane.QUESTION_MESSAGE);

    // Insert the new node as a child of treeNode
    parent.add(new DefaultMutableTreeNode(nodeName));

    // Reload the model since a new tree node is adde
    ((DefaultTreeModel)(jTree1.getModel())).reload();
    ((DefaultTreeModel)(jTree2.getModel())).reload();
  }
});
```



```
jbtRemove.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
    // Get all selected paths
    TreePath[] paths = jTree1.getSelectionPaths();

    if (paths == null) {
      JOptionPane.showMessageDialog(null,
        "No node in the left tree is selected");
      return;
    }

    // Remove all selected nodes
    for (int i = 0; i < paths.length; i++) {
      DefaultMutableTreeNode node = (DefaultMutableTreeNode)
          (paths[i].getLastPathComponent());

      if (node.isRoot()) {
        JOptionPane.showMessageDialog(null,
          "Cannot remove the root");
      }
      else
        node.removeFromParent();
    }

    // Reload the model since a new tree node is added
    ((DefaultTreeModel)(jTree1.getModel())).reload();
    ((DefaultTreeModel)(jTree2.getModel())).reload();
  }
});
```

Tree Node Rendering

```
DefaultTreeCellRenderer renderer =
    (DefaultTreeCellRenderer)jTree1.getCellRenderer();
  renderer.setLeafIcon(yourCustomLeafImageIcon);
  renderer.setOpenIcon(yourCustomOpenImageIcon);
  renderer.setClosedIcon(yourCustomClosedImageIcon);
  renderer.setBackgroundSelectionColor(Color.red);
```

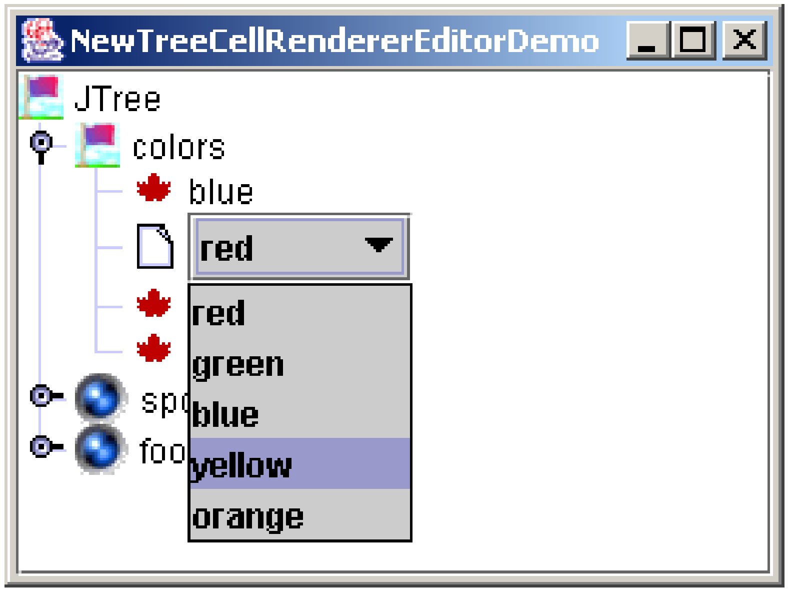


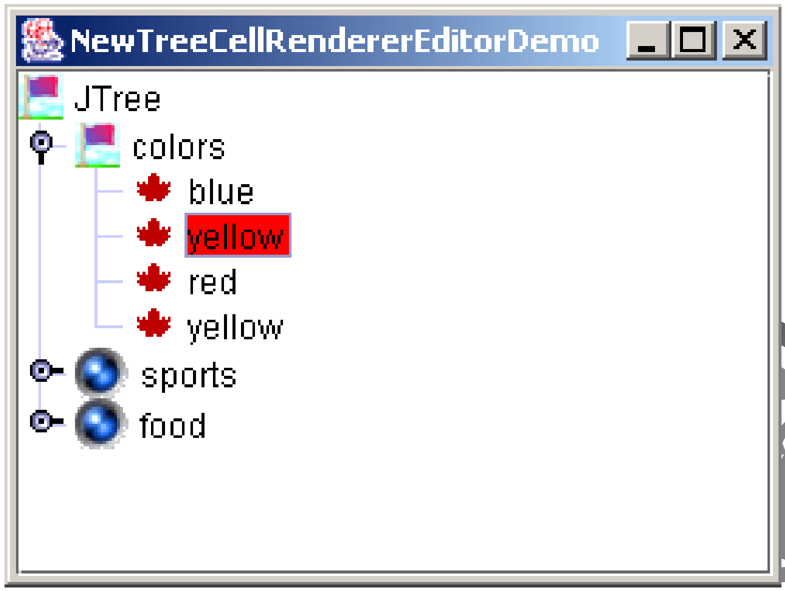
Tree Editing

```
// Customize editor
  JComboBox jcboColor = new JComboBox();
  jcboColor.addItem("red");
  jcboColor.addItem("green");
  jcboColor.addItem("blue");
  jcboColor.addItem("yellow");
  jcboColor.addItem("orange");

  jTree1.setCellEditor(new DefaultCellEditor(jcboColor));
  jTree1.setEditable(true);
```

Tree Rendering and Editing

```
jTree1.setCellEditor
    (new DefaultTreeCellEditor(jTree1,
      new DefaultTreeCellRenderer(),
      new DefaultCellEditor(jcboColor)));
```





TreeCellRendererEditorDemo

Tree Events

JTree can fire TreeSelectionEvent and TreeExpansionEvent, among many other events.

Whenever a new node is selected, JTree fires a TreeSelectionEvent. Whenever a node is expanded or collapsed, JTree fires a TreeExpansionEvent.

To handle the tree selection event, a listener must implement the TreeSelectionListener interface, which contains a single handler named valueChanged method.

TreeExpansionListener contains two handlers named treeCollapsed and treeExpanded for handling node expansion or node closing.