Advanced Programming
CSE 201
Instructor: Sambuddho

(Semester: Monsoon 2025)
Week 10 - Advanced GUI With Swing

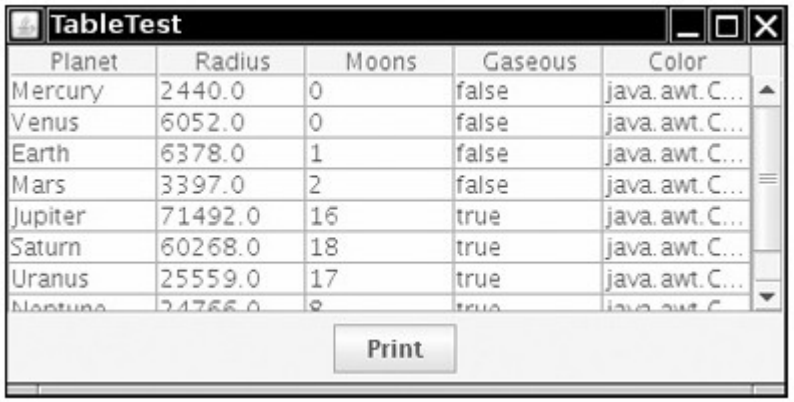
JTable – What & Why?

- JTable displays 2-D tabular data (rows x columns) with built-in UX (headers, scroll, selection).
- JTable is the view; it does not store data – it relies on a TableModel for data.
- Supports editing, sorting, reordering, printing; customizable rendering.

Creating Your First JTable

- Quick prototype approach using Object[][] and column names.
- Good for demos; switch to TableModel for real apps.

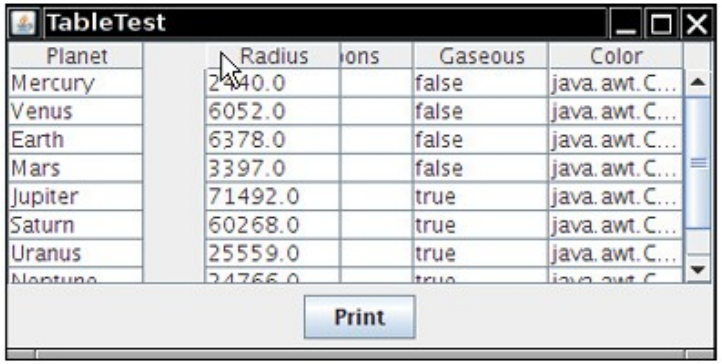
```
var columns = new String[] { "Planet",
"Radius", "Moons", "Gaseous" };
var data = new Object[][] {
    { "Mercury", 2440.0, 0, false },
    { "Earth", 6378.0, 1, false }
};
var table = new JTable(data, columns);
```



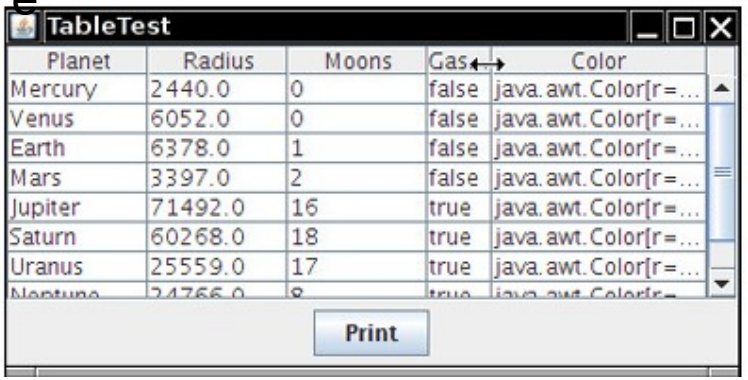
Adding JTable to a GUI with Scroll

- Wrap JTable in JScrollPane to get scrollbars and fixed headers.
- Without a scroll pane, headers won't stay in view during scroll.

```
add(new JScrollPane(table),
BorderLayout.CENTER);
```



Resizable



Default Interactions You Get for Free

- Resize columns by dragging boundaries; reorder columns via header drag.
- Basic cell editing for String/Boolean; selection highlighting.
- Header click sorts when row sorter is enabled (next slide).

One-Line Sorting Enablement

- Click header to toggle ascending/descending order.
- Sorting changes the VIEW order, not the model's data order.

```
table.setAutoCreateRowSorter(true);
```

How JTable Displays Values

- JTable calls toString() on cell objects by default for rendering text.
- Non-text types (e.g., Color) will display their toString form unless custom-rendered.
- Custom renderers let you show icons, color swatches, etc.

Printing a JTable

- Built-in print dialog and straightforward API.
- Useful for invoices, reports, export to paper/PDF workflows.

```
try {
    table.print();
} catch (PrinterException e) {
    e.printStackTrace();
}
```

javax.swing.JTable 1.2

- `JTable(Object[][] entries, Object[] columnNames)`
  constructs a table with a default table model.
- `void print()` 5.0
  displays a print dialog box and prints the table.
- `boolean getAutoCreateRowSorter()` 6
- `void setAutoCreateRowSorter(boolean newValue)` 6
  get or set the `autoCreateRowSorter` property. The default is `false`. When set, a default row sorter is automatically set whenever the model changes.
- `boolean getFillsViewportHeight()` 6
- `void setFillsViewportHeight(boolean newValue)` 6
  get or set the `fillsViewportHeight` property. The default is `false`. When set, the table always fills the enclosing viewport.

JTable Example

```
package table;

import java.awt.*;
import java.awt.print.*;

import javax.swing.*;

/**
 * This program demonstrates how to show a simple table.
 * @version 1.14 2018-05-01
 * @author Cay Horstmann
 */
public class TableTest
{
   public static void main(String[] args)
   {
      EventQueue.invokeLater(() ->
         {
            var frame = new PlanetTableFrame();
            frame.setTitle("TableTest");
            frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
            frame.setVisible(true);
         });
   }
}

/**
 * This frame contains a table of planet data.
 */
class PlanetTableFrame extends JFrame
{
   private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "Color" };
   private Object[][] cells =
   {
      { "Mercury", 2440.0, 0, false, Color.YELLOW },
      { "Venus", 6052.0, 0, false, Color.YELLOW },
      { "Earth", 6378.0, 1, false, Color.BLUE },
      { "Mars", 3397.0, 2, false, Color.RED },
      { "Jupiter", 71492.0, 16, true, Color.ORANGE },
      { "Saturn", 60268.0, 18, true, Color.ORANGE },
      { "Uranus", 25559.0, 17, true, Color.BLUE },
      { "Neptune", 24766.0, 8, true, Color.BLUE },
      { "Pluto", 1137.0, 1, false, Color.BLACK }
   };

   public PlanetTableFrame()
   {
      var table = new JTable(cells, columnNames);
      table.setAutoCreateRowSorter(true);
      add(new JScrollPane(table), BorderLayout.CENTER);
      var printButton = new JButton("Print");
      printButton.addActionListener(event ->
         {
            try { table.print(); }
            catch (SecurityException | PrinterException e) { e.printStackTrace(); }
         });
      var buttonPanel = new JPanel();
      buttonPanel.add(printButton);
      add(buttonPanel, BorderLayout.SOUTH);
      pack();
   }
}
```

Common Early Pitfalls

- Keeping data in Object[][] for too long → hard to maintain and update.
- Tight coupling between UI and data; violates MVC separation.
- Solution: Use TableModel for dynamic, scalable tables.

Cell Editing & Built-in Editors

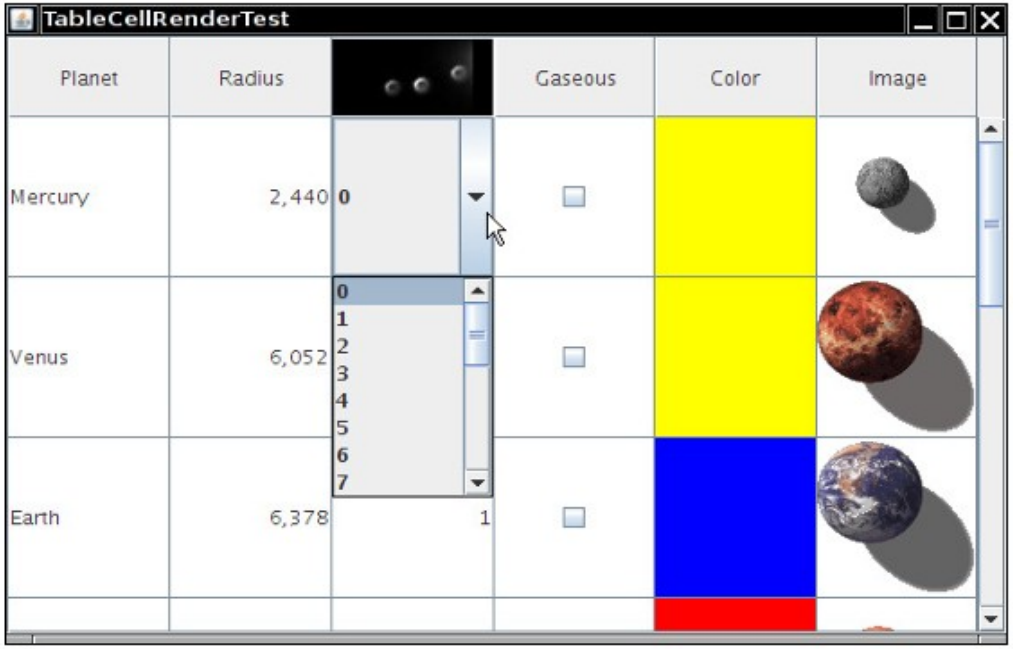
- Tables use editors to handle user input in editable cells.
- Model must indicate which cells are editable in the TableModel:

```
public boolean isCellEditable(int r, int c) {
   return c == PLANET_COLUMN || c == MOONS_COLUMN
       || c == GASEOUS_COLUMN || c == COLOR_COLUMN;
}
```

- AbstractTableModel.isCellEditable(r,c) => false
- DefaultTableModel.isCellEditable(r,c) => true
- DefaultCellEditor supports 3 types:
- JtextField
- JcheckBox
- JComboBox

Cell Editing & Built-in Editors

- DefaultCellEditor supports 3 types:
- JtextField
- JcheckBox
- JComboBox




```
var moonCombo = new JComboBox();
for (int i = 0; i ≤ 20; i++) moonCombo.addItem(i);
var moonEditor = new DefaultCellEditor(moonCombo);
moonColumn.setCellEditor(moonEditor);
```

Why JTable Needs a TableModel

- JTable is the View; TableModel provides the data.
- Model answers: row count, column count, and cell value.
- Decouples UI from data → scalable & testable apps.

TableModel Responsibilities

- Provide row & column count.
- Return value at a given cell (r,c).
- Optional: editing, column names, column classes.

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

```
public Object getValueAt(int r, int c)
{
   try
   {
      rowSet.absolute(r + 1);
      return rowSet.getObject(c + 1);
   }
   catch (SQLException e)
   {
      e.printStackTrace();
      return null;
   }
}
```

Key TableModel Methods

- Only first 3 methods mandatory.
- Defaults help minimize code.

```
int getRowCount();
int getColumnCount();
Object getValueAt(int row, int col);
default String getColumnName(int col);
default Class<?> getColumnClass(int col);
default boolean isCellEditable(int row, int col);
default void setValueAt(Object value, int row, int col);
```

Using AbstractTableModel

- Provides default implementations for most TableModel methods.
- You only implement what you need.
- Supports clean MVC and dynamic updates.

Example: Investment Table Model

- Data computed on the fly, not stored.
- Great for dynamic/derived data.

```
class InvestmentTableModel extends AbstractTableModel {
    private static final double INITIAL = 100_000.0;
    private int years, minRate, maxRate;

    public int getRowCount() { return years; }
    public int getColumnCount() { return maxRate - minRate + 1;
}
    public Object getValueAt(int r, int c) {
        double rate = (c + minRate) / 100.0;
        return "%.2f".formatted(INITIAL * Math.pow(1 + rate,
r));
    }
}
```

Naming Columns

- Override to show meaningful names.
- Improves UX and clarity.

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

Complete Example

```
1  package tableModel;
2
3  import java.awt.*;
4
5  import javax.swing.*;
6  import javax.swing.table.*;
7
8  /**
9   * This program shows how to build a table from a table model.
10  * @version 1.05 2021-09-09
11  * @author Cay Horstmann
12  */
13 public class InvestmentTable
14 {
15    public static void main(String[] args)
16    {
17       EventQueue.invokeLater(() ->
18          {
19             var frame = new InvestmentTableFrame();
20             frame.setTitle("InvestmentTable");
21             frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
22             frame.setVisible(true);
23          });
24    }
25 }
26
27 /**
28  * This frame contains the investment table.
29  */
30 class InvestmentTableFrame extends JFrame
31 {
32    public InvestmentTableFrame()
33    {
34       var model = new InvestmentTableModel(30, 5, 10);
35       var table = new JTable(model);
36       add(new JScrollPane(table));
37       pack();
38    }
39 }
```

```
45 class InvestmentTableModel extends AbstractTableModel
46 {
47    private static double INITIAL_BALANCE = 100000.0;
48
49    private int years;
50    private int minRate;
51    private int maxRate;
52
53    /**
54     * Constructs an investment table model.
55     * @param y the number of years
56     * @param r1 the lowest interest rate to tabulate
57     * @param r2 the highest interest rate to tabulate
58     */
59    public InvestmentTableModel(int y, int r1, int r2)
60    {
61       years = y;
62       minRate = r1;
63       maxRate = r2;
64    }
65
66    public int getRowCount()
67    {
68       return years;
69    }
70
71    public int getColumnCount()
72    {
73       return maxRate - minRate + 1;
74    }
75
76    public Object getValueAt(int r, int c)
77    {
78       double rate = (c + minRate) / 100.0;
79       int nperiods = r;
80       double futureBalance = INITIAL_BALANCE * Math.pow(1 + rate, nperiods);
81       return "%.2f".formatted(futureBalance);
82    }
83
84    public String getColumnName(int c)
85    {
86       return (c + minRate) + "%";
87    }
88 }
```

Benefits of Custom Models

- Cleaner than maintaining Object[][].
- Supports dynamic add/remove/update rows.
- Easier to attach to DB, files, APIs.

When to Use DefaultTableModel

- For quick simple editable tables.
- Stores data internally (like Excel).
- Not ideal for computed/business logic.

AbstractTableModel vs DefaultTableModel

- DefaultTableModel: stores data internally; easy for simple use.
- AbstractTableModel: cleaner MVC; computation-friendly.
- Choose based on application complexity.

Selections in JTable

- Row selection is enabled by default; clicking selects a row.
- Control selection mode via ListSelectionModel (single/interval/multiple).
- Enable column/cell selection explicitly if needed.

Selection Modes - API

- Row/Column/Cell selection are configurable.
- Use selection model for single vs interval vs multiple.

```
table.setRowSelectionAllowed(true);
table.setColumnSelectionAllowed(false);
table.setCellSelectionEnabled(false);
table.getSelectionModel().setSelectionMode(
    ListSelectionModel.SINGLE_INTERVAL_SELECTION);
// Get selected rows/columns
int[] rows = table.getSelectedRows();
int[] cols = table.getSelectedColumns();
```

Resizing & Reordering Columns

- User can drag boundaries to resize; drag headers to reorder.
- Programmatic control via TableColumn (min, max, preferred widths).
- Auto resize mode governs how other columns adjust.

Column Sizing & Auto-Resize Modes

- Set width constraints per column.
- Choose an auto-resize strategy appropriate for UX.

```
TableColumn col = table.getColumnModel().getColumn(0);
col.setMinWidth(80);
col.setPreferredWidth(140);
col.setMaxWidth(300);

table.setAutoResizeMode(JTable.AUTO_RESIZE_SUBSEQUENT_COLUMNS);
// Other modes: OFF, NEXT_COLUMN, LAST_COLUMN, ALL_COLUMNS
```

Sorting with TableRowSorter

- Auto-sorting: table.setAutoCreateRowSorter(true).
- For control: install TableRowSorter<TableModel> and customize.
- Provide column-specific comparators or disable sort for certain columns.

Custom Sorters per Column

- Disable sort where it makes no sense (e.g., icons).
- Provide comparators for non-Comparable types.

```
var sorter = new TableRowSorter<TableModel>(table.getModel());
table.setRowSorter(sorter);
// Disable sort on image column
sorter.setSortable(5, false);
// Custom comparator example (Color by B, then G, then R)
sorter.setComparator(4, Comparator
    .comparing(Color::getBlue)
    .thenComparing(Color::getGreen)
    .thenComparing(Color::getRed));
```

Filtering Rows - Realistic Scenario (Student Records)

- Use RowFilter with TableRowSorter for view-level filtering.
- Combine multiple filters (AND/OR/NOT) for compound criteria.
- Great for dashboards: e.g., CGPA > 8, Dept = CSE, Name starts with 'A'.

RowFilter - Student Records Example

- Mix numberFilter and regexFilter for realistic criteria.
- andFilter/orFilter/notFilter compose complex queries.

```
// Assume columns: 0:Name, 1:Dept, 2:CGPA
var sorter = new TableRowSorter<TableModel>(table.getModel());
table.setRowSorter(sorter);
var filters = new java.util.ArrayList<RowFilter<TableModel,
Integer>>();
// CGPA > 8.0 (numeric filter on Double class column)
filters.add(RowFilter.numberFilter(RowFilter.ComparisonType.AFTER,
8.0, 2));
// Dept = "CSE" (regex exact match)
filters.add(RowFilter.regexFilter("^CSE$", 1));
// Name starts with A
filters.add(RowFilter.regexFilter("^A", 0));
sorter.setRowFilter(RowFilter.andFilter(filters));
```

View vs Model Indices (Important!)

- Sorting/filtering change the VIEW order.
- Get selected rows via view indices; map to model with convertRowIndexToModel().
- Similarly for columns: convertColumnIndexToModel().

Index Mapping Example

- Always convert view index to model index before using the model.
- Prevents wrong-row updates after sorting/filtering.

```
int[] selectedViewRows = table.getSelectedRows();
for (int vr : selectedViewRows) {
    int mr = table.convertRowIndexToModel(vr);
    // Use mr to read/update TableModel safely
}
```

Hiding & Showing Columns

- removeColumn hides from VIEW, model data remains intact.
- Remember order changes; moveColumn can reposition.

```
var colModel = table.getColumnModel();
int[] selected = table.getSelectedColumns();
java.util.List<TableColumn> removed = new java.util.ArrayList<>();
for (int i = selected.length - 1; i >= 0; i--) {
    var tc = colModel.getColumn(selected[i]);
    table.removeColumn(tc);
    removed.add(tc);
}
// Restore later:
for (var tc : removed) table.addColumn(tc);
```

Complete Example

```
package tableRowColumn;

import java.awt.*;
import java.util.*;

import javax.swing.*;
import javax.swing.table.*;

/**
 * This frame contains a table of planet data.
 */
public class PlanetTableFrame extends JFrame
{
   private static final int DEFAULT_WIDTH = 600;
   private static final int DEFAULT_HEIGHT = 500;

   public static final int COLOR_COLUMN = 4;
   public static final int IMAGE_COLUMN = 5;

   private JTable table;
   private HashSet<Integer> removedRowIndices;
   private ArrayList<TableColumn> removedColumns;
   private JCheckBoxMenuItem rowsItem;
   private JCheckBoxMenuItem columnsItem;
   private JCheckBoxMenuItem cellsItem;

   private String[] columnNames = { "Planet", "Radius", "Moons", "Gaseous", "C
```

```
   private Object[][] cells = {
      { "Mercury", 2440.0, 0, false, Color.YELLOW,
         new ImageIcon(getClass().getResource("Mercury.gif")) },
      { "Venus", 6052.0, 0, false, Color.YELLOW,
         new ImageIcon(getClass().getResource("Venus.gif")) },
      { "Earth", 6378.0, 1, false, Color.BLUE,
         new ImageIcon(getClass().getResource("Earth.gif")) },
      { "Mars", 3397.0, 2, false, Color.RED,
         new ImageIcon(getClass().getResource("Mars.gif")) },
      { "Jupiter", 71492.0, 16, true, Color.ORANGE,
         new ImageIcon(getClass().getResource("Jupiter.gif")) },
      { "Saturn", 60268.0, 18, true, Color.ORANGE,
         new ImageIcon(getClass().getResource("Saturn.gif")) },
      { "Uranus", 25559.0, 17, true, Color.BLUE,
         new ImageIcon(getClass().getResource("Uranus.gif")) },
      { "Neptune", 24766.0, 8, true, Color.BLUE,
         new ImageIcon(getClass().getResource("Neptune.gif")) },
      { "Pluto", 1137.0, 1, false, Color.BLACK,
         new ImageIcon(getClass().getResource("Pluto.gif")) } };

   public PlanetTableFrame()
   {
      setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);

      var model = new DefaultTableModel(cells, columnNames)
         {
            public Class<?> getColumnClass(int c)
            {
               return cells[0][c].getClass();
            }
         };

      table = new JTable(model);

      table.setRowHeight(100);
      table.getColumnModel().getColumn(COLOR_COLUMN).setMinWidth(250);
      table.getColumnModel().getColumn(IMAGE_COLUMN).setMinWidth(100);
```

Complete Example

```
67       var sorter = new TableRowSorter<TableModel>(model);
68       table.setRowSorter(sorter);
69       sorter.setComparator(COLOR_COLUMN, Comparator.comparing(Color::getBlue)
70          .thenComparing(Color::getGreen).thenComparing(Color::getRed));
71       sorter.setSortable(IMAGE_COLUMN, false);
72       add(new JScrollPane(table), BorderLayout.CENTER);
73
74       removedRowIndices = new HashSet<>();
75       removedColumns = new ArrayList<>();
76
77       var filter = new RowFilter<TableModel, Integer>()
78          {
79             public boolean include(Entry<? extends TableModel, ? extends Integer> entry)
80             {
81                return !removedRowIndices.contains(entry.getIdentifier());
82             }
83          };
84
85       // create menu
86
87       var menuBar = new JMenuBar();
88       setJMenuBar(menuBar);
89
90       var selectionMenu = new JMenu("Selection");
91       menuBar.add(selectionMenu);
92
93       rowsItem = new JCheckBoxMenuItem("Rows");
94       columnsItem = new JCheckBoxMenuItem("Columns");
95       cellsItem = new JCheckBoxMenuItem("Cells");
96
97       rowsItem.setSelected(table.getRowSelectionAllowed());
98       columnsItem.setSelected(table.getColumnSelectionAllowed());
99       cellsItem.setSelected(table.getCellSelectionEnabled());
100
101      rowsItem.addActionListener(event ->
102         {
103            table.clearSelection();
104            table.setRowSelectionAllowed(rowsItem.isSelected());
105            updateCheckboxMenuItems();
106         });
107      selectionMenu.add(rowsItem);
108
109      columnsItem.addActionListener(event ->
110         {
111            table.clearSelection();
112            table.setColumnSelectionAllowed(columnsItem.isSelected());
113            updateCheckboxMenuItems();
114         });
115      selectionMenu.add(columnsItem);
116
117      cellsItem.addActionListener(event ->
118         {
119            table.clearSelection();
120            table.setCellSelectionEnabled(cellsItem.isSelected());
```

Complete Example

```
            updateCheckboxMenuItems();
         });
      selectionMenu.add(cellsItem);

      var tableMenu = new JMenu("Edit");
      menuBar.add(tableMenu);

      var hideColumnsItem = new JMenuItem("Hide Columns");
      hideColumnsItem.addActionListener(event ->
         {
            int[] selected = table.getSelectedColumns();
            TableColumnModel columnModel = table.getColumnModel();

            // remove columns from view, starting at the last
            // index so that column numbers aren't affected

            for (int i = selected.length - 1; i >= 0; i--)
            {
               TableColumn column = columnModel.getColumn(selected[i]);
               table.removeColumn(column);

               // store removed columns for "show columns" command

               removedColumns.add(column);
            }
         });
      tableMenu.add(hideColumnsItem);

      var showColumnsItem = new JMenuItem("Show Columns");
      showColumnsItem.addActionListener(event ->
         {
            // restore all removed columns
            for (TableColumn tc : removedColumns)
               table.addColumn(tc);
            removedColumns.clear();
         });
      tableMenu.add(showColumnsItem);

      var hideRowsItem = new JMenuItem("Hide Rows");
      hideRowsItem.addActionListener(event ->
         {
            int[] selected = table.getSelectedRows();
            for (int i : selected)
               removedRowIndices.add(table.convertRowIndexToModel(i));
            sorter.setRowFilter(filter);
         });
      tableMenu.add(hideRowsItem);

      var showRowsItem = new JMenuItem("Show Rows");
      showRowsItem.addActionListener(event ->
         {
            removedRowIndices.clear();
            sorter.setRowFilter(filter);
         });
      tableMenu.add(showRowsItem);

      var printSelectionItem = new JMenuItem("Print Selection");
      printSelectionItem.addActionListener(event ->
         {
            int[] selected = table.getSelectedRows();
            System.out.println("Selected rows: " + Arrays.toString(selected));
            selected = table.getSelectedColumns();
            System.out.println("Selected columns: " + Arrays.toString(selected));
         });
      tableMenu.add(printSelectionItem);
   }

   private void updateCheckboxMenuItems()
   {
      rowsItem.setSelected(table.getRowSelectionAllowed());
      columnsItem.setSelected(table.getColumnSelectionAllowed());
      cellsItem.setSelected(table.getCellSelectionEnabled());
   }
}
```

What is a Cell Renderer?

- Controls how each cell is displayed in a JTable.
- Affects appearance only (colors, fonts, icons, alignment).
- Does NOT change the underlying data stored in the model.
- Each column may use a different renderer.

Default vs Custom Renderers

- Default renderers convert values to text using toString().
- Boolean → checkbox, Number → right-aligned, others → text.
- Custom renderers show meaningful visuals (colors, icons, emojis).
- Makes the UI more expressive and user-friendly.

Custom Cell Rendering

- TableCellRenderer decides how cell contents are drawn.
- Has a method that returns a Component whose paint() fills the cell area.

```
interface TableCellRenderer {
   Component getTableCellRendererComponent(
       JTable table, Object value,
       boolean isSelected, boolean hasFocus,
       int row, int column);
}
```

Example - Color Renderer

```
class ColorTableCellRenderer extends JPanel
      implements TableCellRenderer {
   public Component getTableCellRendererComponent(
      JTable table, Object value, boolean isSelected,
      boolean hasFocus, int row, int column) {
      setBackground((Color) value);
      if (hasFocus)
         setBorder(UIManager.getBorder(
            "Table.focusCellHighlightBorder"));
      else setBorder(null);
      return this;
   }
}
table.setDefaultRenderer(Color.class,new ColorTableCellRenderer());
```

Rendering Table Headers

- To display icons or special content in headers, manually set both value and renderer.

```
moonColumn.setHeaderValue(
    new ImageIcon("Moons.gif"));
moonColumn.setHeaderRenderer(
    table.getDefaultRenderer(ImageIcon.class));
```

Cell Editing & Built-in Editors

- Tables use editors to handle user input in editable cells.
- Model must indicate which cells are editable in the TableModel:

```
public boolean isCellEditable(int r, int c) {
   return c == PLANET_COLUMN || c == MOONS_COLUMN
       || c == GASEOUS_COLUMN || c == COLOR_COLUMN;
}
```

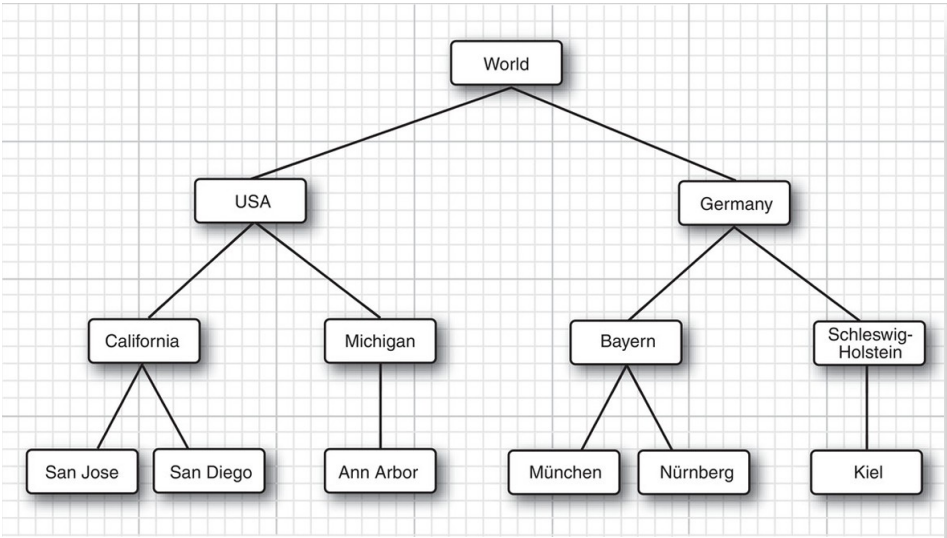
- AbstractTableModel.isCellEditable(r,c) => false
- DefaultTableModel.isCellEditable(r,c) => true
- DefaultCellEditor supports 3 types:
- JtextField
- JcheckBox
- JComboBox

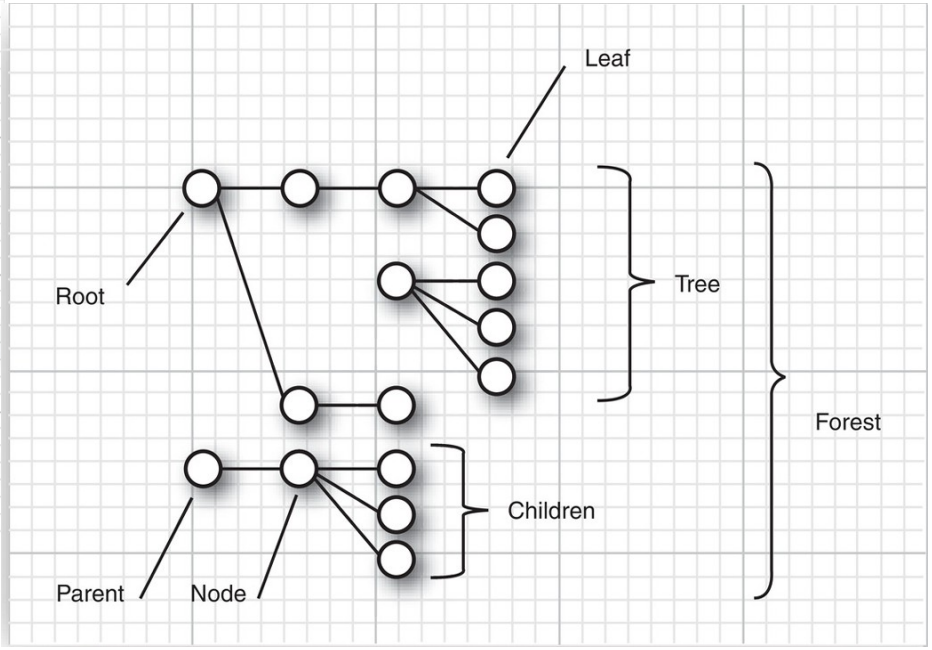
JTree — Hierarchical Data in Swing

- Displays tree-structured data (parent → children).
- Great for file explorers, org charts, university catalogs.
- Uses a TreeModel (like JTable uses TableModel).
- We'll build a University → Schools → Departments → Courses tree.

JTree Building Blocks

- Nodes: DefaultMutableTreeNode (stores user object + children).
- Paths: TreePath represents a node's ancestry from the root.
- Model: TreeModel provides structure & change notifications.
- View: JTree renders nodes and handles expand/collapse.





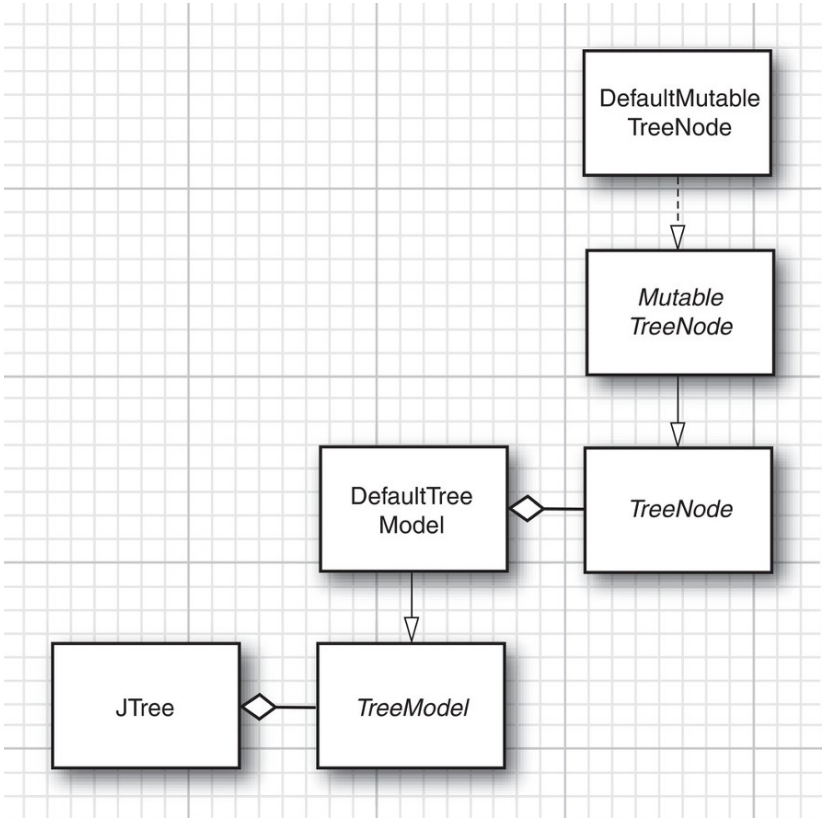
Programmatically Building a Tree

```
TreeModel model = . . .;
var tree = new JTree(model);
```

DefaultTreeModel

```
TreeNode root = . . .;
var model = new DefaultTreeModel(root);
```

DefaultMutableTreeNode class implements TreeModel interface. This can be used for creating a TreeNode.



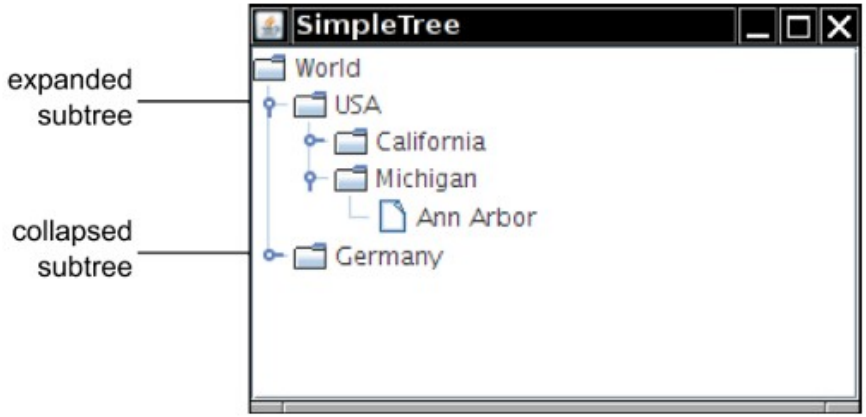
Programmatically Building a Tree

```
var root = new DefaultMutableTreeNode("World");
var country = new DefaultMutableTreeNode("USA");
root.add(country);
var state = new DefaultMutableTreeNode("California")
country.add(state);
```

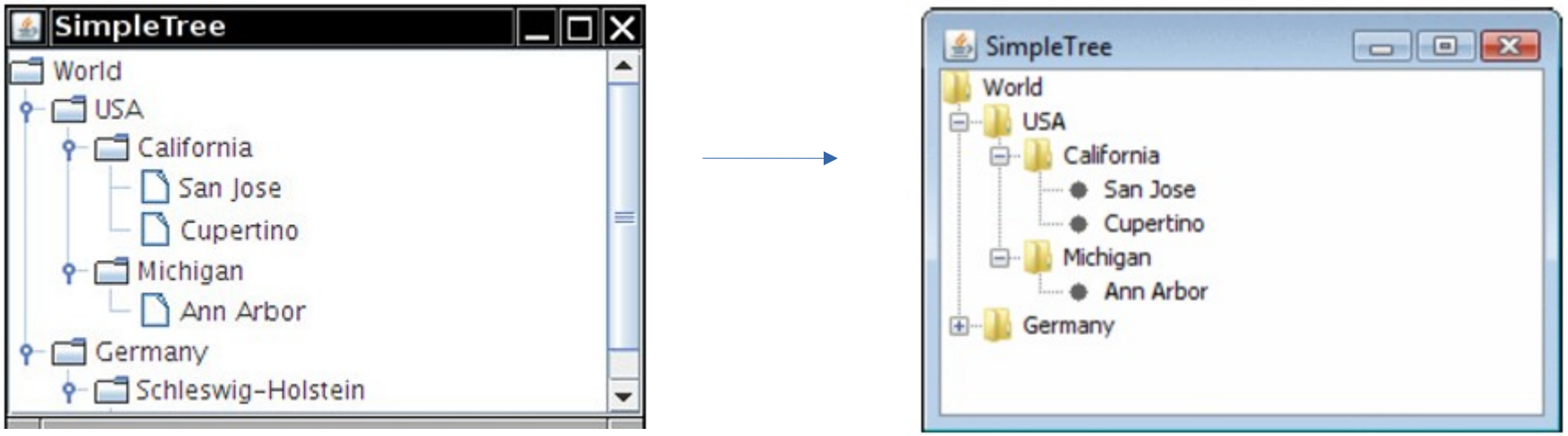

```
var treeModel = new DefaultTreeModel(root);
var tree = new JTree(treeModel);
```

Programmatically Building a Tree

```
1  package tree;
2
3  import javax.swing.*;
4  import javax.swing.tree.*;
5
6  /**
7   * This frame contains a simple tree that displays a manually constructed tree model.
8   */
9  public class SimpleTreeFrame extends JFrame
10 {
11    private static final int DEFAULT_WIDTH = 300;
12    private static final int DEFAULT_HEIGHT = 200;
13
14    public SimpleTreeFrame()
15    {
16       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
17
18       // set up tree model data
19
20       var root = new DefaultMutableTreeNode("World");
21       var country = new DefaultMutableTreeNode("USA");
22       root.add(country);
23       var state = new DefaultMutableTreeNode("California");
24       country.add(state);
25       var city = new DefaultMutableTreeNode("San Jose");
26       state.add(city);
27       city = new DefaultMutableTreeNode("Cupertino");
28       state.add(city);
29       state = new DefaultMutableTreeNode("Michigan");
30       country.add(state);
31       city = new DefaultMutableTreeNode("Ann Arbor");
32       state.add(city);
33       country = new DefaultMutableTreeNode("Germany");
34       root.add(country);
35       state = new DefaultMutableTreeNode("Schleswig-Holstein");
36       country.add(state);
37       city = new DefaultMutableTreeNode("Kiel");
38       state.add(city);
39
40       // construct tree and put it in a scroll pane
41
42       var tree = new JTree(root);
43       add(new JScrollPane(tree));
44    }
45 }
```



Changing ``Look-n-Feel'' of a JTree




```
UIManager.setLookAndFeel("com.sun.java.swing.plaf.windows
.WindowsLookAndFeel");
```

Other examples:

```
UIManager.setLookAndFeel("javax.swing.plaf.nimbus.NimbusLookAndFeel");
UIManager.setLookAndFeel("com.sun.java.swing.plaf.motif.MotifLookAndFeel");
```

Creating the JTree and Adding to UI

- Root visible + handles make navigation clear.
- Always wrap trees in JScrollPane for usability.

```
var tree = new JTree(root);
tree.setRootVisible(true);
tree.setShowsRootHandles(true); // nicer expand/collapse handles
add(new JScrollPane(tree), BorderLayout.WEST);
```

What is a Node and What is a Leaf

JTree doesn't distinguish between a node and a leaf. Nodes with children are regular nodes with directory icon and without children are automatically leaf.

The `isLeaf()` method of the DefaultMutableTreeNode class returns true if the node has no children.

For nodes that should have no children the method to call is `DefaultMutableTreeNode.setAllowChildren(false)`.

Thereafter we need to inform the DefaultTreeModel to check for value to determine if the `DefaultMutlableTreeNode` supports children or not.

```
DefaultTreeMode.setAsksAllowChildren(true);
```

Expanding/Collapsing & Programmatically Expand All

- Users can click handles or double-click to expand/collapse.
- Programmatically expand all rows after model set.
- Collapse all similarly if needed.
- Call after the tree has been realized/added to UI.
- For dynamic trees, use recursion on TreePath children.

```
for (int i = 0; i < tree.getRowCount(); i++) {
    tree.expandRow(i);
}
```

Editing Trees and Tree Paths

- The JTree class requires complete paths rather than just a single node.
- Reason: JTree doesn’t inherently understand the TreeNode interface.
- The TreeModel interface doesn’t use TreeNode; only DefaultTreeModel does.
- Other tree models may manage objects that don’t have getParent or getChild methods.
- It’s the tree model’s responsibility to define how nodes are linked.

Therefore, JTree itself must rely on full paths to navigate, since it has no knowledge of how nodes are connected internally.

```
TreePath selectionPath = tree.getSelectionPath();
var selectedNode = (DefaultMutableTreeNode) selectionPath.getLastPathComponent();
```

Editing Trees and Tree Paths

- When modifying a tree structure in Java Swing, don't directly add a child node using selectedNode.add(newNode);.
- Doing so updates the model but doesn't notify the view, so the UI won't reflect the change.
- Instead, use DefaultTreeModel.insertNodeInto(...), which updates the model and automatically notifies the view.

```
model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());
```

This appends newNode as the last child of selectedNode and ensures the tree view updates properly.

Avoid using DefaultTreeModel.reload() for minor updates—it reloads the entire tree and collapses all nodes beyond the root's children, disrupting the user experience. Use targeted update methods instead.

Editing Trees and Tree Paths

- To make a newly added node visible in a collapsed JTree, you must manually expand its parent path.
- Use DefaultTreeModel.getPathToRoot(newNode) to get the path from the root to the node, then pass it to JTree.makeVisible(path) to ensure the UI reveals it.

```
TreeNode[] nodes = model.getPathToRoot(newNode);
var path = new TreePath(nodes);
tree.makeVisible(path);
```

Editing Trees and Tree Paths: ScrollPanes

To ensure a newly expanded tree node is visible inside a scroll pane, use `tree.scrollPathToVisible(path)` instead of `makeVisible()`. This method expands all nodes along the path and scrolls the viewport so the target node comes into view.

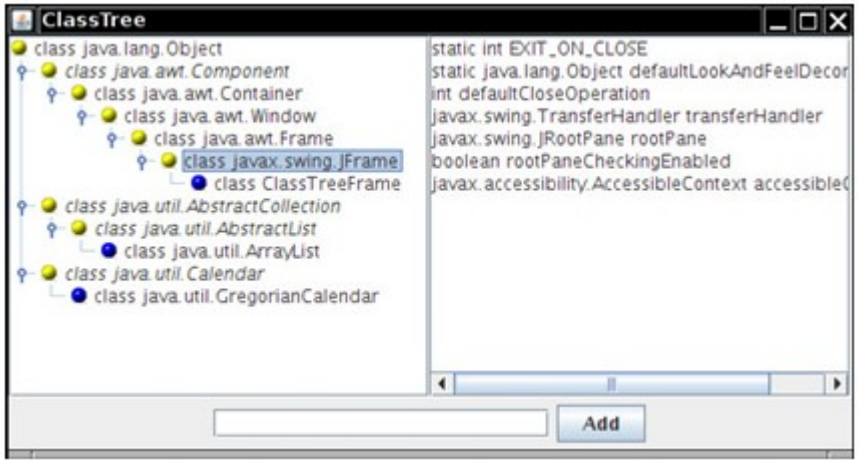
TreePath Example

```
1  package treeEdit;
2
3  import java.awt.*;
4  import javax.swing.*;
5  import javax.swing.tree.*;
6
7  /**
8   * A frame with a tree and buttons to edit the tree.
9   */
10 public class TreeEditFrame extends JFrame
11 {
12    private static final int DEFAULT_WIDTH = 400;
13    private static final int DEFAULT_HEIGHT = 200;
14
15    private DefaultTreeModel model;
16    private JTree tree;
17
18    public TreeEditFrame()
19    {
20       setSize(DEFAULT_WIDTH, DEFAULT_HEIGHT);
21
22       // construct tree
23
24       TreeNode root = makeSampleTree();
25       model = new DefaultTreeModel(root);
26       tree = new JTree(model);
27       tree.setEditable(true);
28
29       // add scroll pane with tree
30
31       var scrollPane = new JScrollPane(tree);
32       add(scrollPane, BorderLayout.CENTER);
33
34       makeButtons();
35    }
36
37    public TreeNode makeSampleTree()
38    {
39       var root = new DefaultMutableTreeNode("World");
40       var country = new DefaultMutableTreeNode("USA");
41       root.add(country);
42       var state = new DefaultMutableTreeNode("California");
43       country.add(state);
44       var city = new DefaultMutableTreeNode("San Jose");
45       state.add(city);
46       city = new DefaultMutableTreeNode("San Diego");
47       state.add(city);
48       state = new DefaultMutableTreeNode("Michigan");
49       country.add(state);
50       city = new DefaultMutableTreeNode("Ann Arbor");
51       state.add(city);
52       country = new DefaultMutableTreeNode("Germany");
53       root.add(country);
54       state = new DefaultMutableTreeNode("Schleswig-Holstein");
55       country.add(state);
56       city = new DefaultMutableTreeNode("Kiel");
57       state.add(city);
58       return root;
59    }
60
61    /**
62     * Makes the buttons to add a sibling, add a child, and delete a node.
63     */
64    public void makeButtons()
65    {
66       var panel = new JPanel();
67       var addSiblingButton = new JButton("Add Sibling");
68
69       addSiblingButton.addActionListener(event ->
70          {
71             var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
72
73             if (selectedNode == null) return;
74
75             var parent = (DefaultMutableTreeNode) selectedNode.getParent();
76
77             if (parent == null) return;
78
             var newNode = new DefaultMutableTreeNode("New");
```

TreePath Example

```
80              int selectedIndex = parent.getIndex(selectedNode);
81              model.insertNodeInto(newNode, parent, selectedIndex + 1);
82
83              // now display new node
84
85              TreeNode[] nodes = model.getPathToRoot(newNode);
86              var path = new TreePath(nodes);
87              tree.scrollPathToVisible(path);
88           });
89        panel.add(addSiblingButton);
90
91        var addChildButton = new JButton("Add Child");
92        addChildButton.addActionListener(event ->
93           {
94              var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
95
96              if (selectedNode == null) return;
97
98              var newNode = new DefaultMutableTreeNode("New");
99              model.insertNodeInto(newNode, selectedNode, selectedNode.getChildCount());
100
101             // now display new node
102
103             TreeNode[] nodes = model.getPathToRoot(newNode);
104             var path = new TreePath(nodes);
105             tree.scrollPathToVisible(path);
106          });
107       panel.add(addChildButton);
108
109       var deleteButton = new JButton("Delete");
110       deleteButton.addActionListener(event ->
111          {
112             var selectedNode = (DefaultMutableTreeNode) tree.getLastSelectedPathComponent();
113
114             if (selectedNode != null && selectedNode.getParent() != null) model
115                   .removeNodeFromParent(selectedNode);
116          });
117       panel.add(deleteButton);
118       add(panel, BorderLayout.SOUTH);
119    }
120 }
```

Listening to JTree Events



A JTree often interacts with another component (e.g., a panel or text area). When the user selects a node, some related info (class details, fields etc.) is displayed elsewhere.

Implement the `TreeSelectionListener` interface:
`void valueChanged(TreeSelectionEvent event);`
(Invoked whenever user selects or deselects tree nodes)
`tree.addTreeSelectionListener(listener);`

Example Behaviour

- User clicks on a class name.
- The listener's `valueChanged()` method fires.
- App queries selected path and shows fields in a right-hand pane

Controlling How Users Select Nodes

TreeSelectionModel

- JTree uses a TreeSelectionModel to manage which and how many nodes can be selected.
- int mode = TreeSelectionModel.SINGLE_TREE_SELECTION;
- tree.getSelectionModel().setSelectionMode(mode);

Mode Description
SINGLE_TREE_SELECTION: Only one node selectable at a time.
CONTIGUOUS_TREE_SELECTION: Adjacent nodes allowed.
DISCONTIGUOUS_TREE_SELECTION (default): Arbitrary non-adjacent nodes.
Additional Interaction
Ctrl-click → add/remove nodes from selection.
Shift-click → select a range of adjacent nodes.

Querying Selected Nodes and Displaying Details

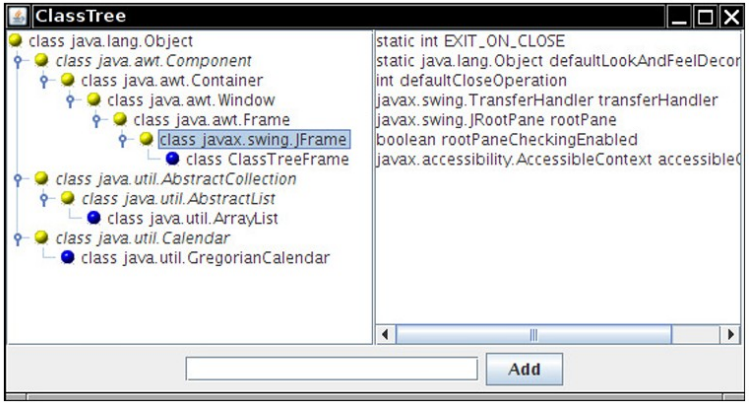
```
import javax.swing.*;
import javax.swing.event.*;
import javax.swing.tree.*;


public class TreeSelectionDemo {
    public static void main(String[] args) {
        JFrame frame = new JFrame("JTree Selection Demo");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        // Sample tree data
        DefaultMutableTreeNode root = new DefaultMutableTreeNode("Classes");
        DefaultMutableTreeNode comp = new DefaultMutableTreeNode("java.awt.Component");
        root.add(comp);
        comp.add(new DefaultMutableTreeNode("java.awt.Container"));
        comp.add(new DefaultMutableTreeNode("javax.swing.JComponent"));
        JTree tree = new JTree(root);


        // Text area to show selected node
        JTextArea infoArea = new JTextArea(5, 25);
        infoArea.setEditable(false);


        // Add listener for selection changes
        tree.addTreeSelectionListener(new TreeSelectionListener() {

            public void valueChanged(TreeSelectionEvent event) {
                TreePath path = tree.getSelectionPath();
                if (path ≠ null)
                    infoArea.setText("Selected node: " + path.getLastPathComponent());
                else
                    infoArea.setText("No selection");
            }
        });


        // Layout
        frame.add(new JScrollPane(tree), "West");
        frame.add(new JScrollPane(infoArea), "Center");
        frame.pack();
        frame.setVisible(true);
    }
}
```

To find which nodes are selected in a JTree:
`TreePath[] selectedPaths = tree.getSelectionPaths();` // Returns all currently selected paths (useful in multi-selection mode)
 – Returns an array of selected paths. If only a single path is to be selected then use `tree.getSelectionPath();`

Rendering Tree Nodes

Customising JTree node appearance is done using a tree cell renderer.

- The default renderer is DefaultTreeCellRenderer, which extends Jlabel.
- You can customize:
  - Icons for leaf, closed, and open nodes
  - Fonts
  - Background colors
  - Three ways to customize rendering:




```
var renderer = new DefaultTreeCellRenderer();
renderer.setLeafIcon(new ImageIcon("blue-ball.gif")); // used for leaf nodes
renderer.setClosedIcon(new ImageIcon("red-ball.gif")); // used for collapsed nodes
renderer.setOpenIcon(new ImageIcon("yellow-ball.gif")); // used for expanded nodes
tree.setCellRenderer(renderer);
```

Use DefaultTreeCellRenderer to set global styles.

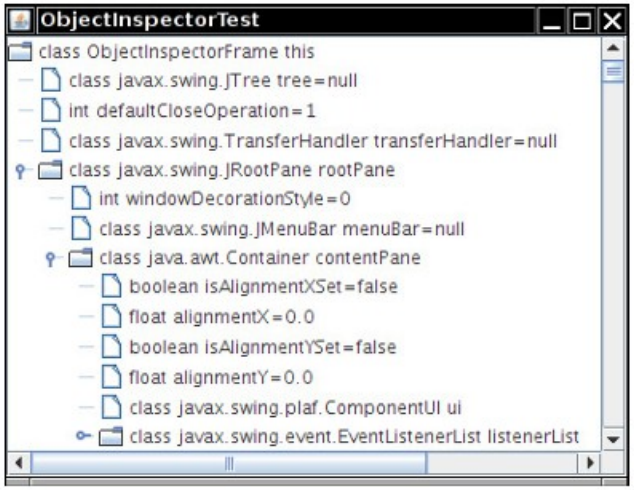
Extend DefaultTreeCellRenderer to vary appearance per node.

Implement TreeCellRenderer for full control over drawing.

The renderer does not draw expand/collapse handles --> managed by the look-and-feel system and shouldn't be changed.

Custom Tree Models

- JTree normally uses DefaultTreeModel and DefaultMutableTreeNode.
- But when data already forms a hierarchy (e.g., objects linked by references), duplicating it is wasteful.
- Normally, JTree relies on DefaultTreeModel + DefaultMutableTreeNode.
  - → Suitable when data isn't inherently hierarchical.
- But many real-world structures (e.g., Java objects, XML DOM, directory trees) are already linked.
- The Custom Tree Model directly reflects real object relationships using the TreeModel interface.
- Example program: Object Inspector - shows instance fields of any object (like a debugger).



Understanding the TreeModel Interface

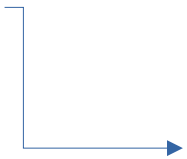
- Core Methods (Structure Discovery)

```
Object getRoot()
int getChildCount(Object parent)
Object getChild(Object parent, int index)
```

Change Notification (Model-View Synchronization)

```
int getIndexOfChild(Object parent, Object child)

boolean isLeaf(Object node)
```

```
void addTreeModelListener(TreeModelListener l)
void removeTreeModelListener(TreeModelListener l)
```

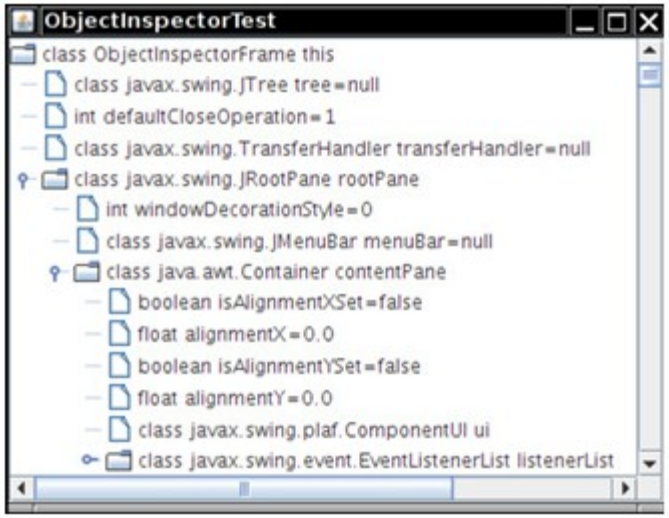


```
void treeNodesChanged(TreeModelEvent e)
void treeNodesInserted(TreeModelEvent e)
void treeNodesRemoved(TreeModelEvent e)
void treeStructureChanged(TreeModelEvent e)
```

Understanding the TreeModel Interface

Event Mechanism

- TreeModelEvent carries details about structure changes (insert/delete/modify).
- TreeModelListener reacts to 4 possible events:
  - treeNodesChanged()
  - treeNodesInserted()
  - treeNodesRemoved()
  - TreeStructureChanged()



Basic Java Object Inspection (J)Tree

Class Variable: Representing Nodes

Purpose:
Each node is a Variable object describing
<type, name, value, list of fields>.

```
public class Variable {
    private Class<?> type; private String name;
    private Object value;  private ArrayList<Field> fields;


    public Variable(Class<?> t, String n, Object v) {
        type = t; name = n; value = v; fields = new ArrayList<>();


        if (!t.isPrimitive() && !t.isArray()
            && !t.equals(String.class) && v ≠ null) {
            for (Class<?> c = v.getClass(); c ≠ null; c = c.getSuperclass()) {
                Field[] fs = c.getDeclaredFields();
                AccessibleObject.setAccessible(fs, true);
                for (Field f : fs)
                    if (!Modifier.isStatic(f.getModifiers()))
                        fields.add(f);
            }
        }
    }
    public ArrayList<Field> getFields() { return fields; }
    public Object getValue() { return value; }


    public String toString() {
        String r = type + " " + name;
        if (type.isPrimitive() || type.equals(String.class))
            r += "=" + value;
        else if (value == null) r += "=null";
        return r;
    }
}
```

Minimal TreeModel Implementation

```
public Object getRoot() { return root; }

public int getChildCount(Object parent) {
    return ((Variable) parent).getFields().size();
}

public Object getChild(Object parent, int index) {
    ArrayList<Field> fields = ((Variable) parent).getFields();
    Field f = fields.get(index);
    Object parentVal = ((Variable) parent).getValue();
    try {
        return new Variable(f.getType(), f.getName(),
f.get(parentVal));
    } catch (IllegalAccessException e) {
        return null;
    }
}
```

- Variable represents each node; fields become children.
- No explicit “tree node” class — any Object may serve as a node.

Connecting TreeModel to JTree

Purpose:
Integrating the Model with a GUI Frame

```
public class ObjectInspectorFrame extends JFrame {
    public ObjectInspectorFrame() {
        setSize(400, 300);

        // Create root variable for 'this' frame
        Variable v = new Variable(getClass(), "this", this);

        // Build custom model and assign root
        ObjectTreeModel model = new ObjectTreeModel();
        model.setRoot(v);

        // Bind model to JTree and display
        JTree tree = new JTree(model);
        add(new JScrollPane(tree), BorderLayout.CENTER);
    }
}
```

Explanation

- The GUI builds a live tree of the frame itself.
- Expanding nodes reveals the internal Swing component hierarchy.
- Demonstrates dynamic reflection + GUI synchronization.
- ObjectTreeModel fires events to refresh display when root changes (fireTreeStructureChanged).
- EventListenerList helps manage model listeners efficiently.
- Shows MVC separation:
- Model → object relationships
- View → JTree visualization
- Controller → user expansion actions.

Connecting TreeModel to JTree

```
public class ObjectInspectorFrame extends JFrame {
    public ObjectInspectorFrame() {
        setSize(400, 300);

        // Create root variable for 'this' frame
        Variable v = new Variable(getClass(), "this", this);

        // Build custom model and assign root
        ObjectTreeModel model = new ObjectTreeModel();
        model.setRoot(v);

        // Bind model to JTree and display
        JTree tree = new JTree(model);
        add(new JScrollPane(tree), BorderLayout.CENTER);
    }
}

public class ObjectTreeModel implements TreeModel
{
    private Variable root;
    private EventListenerList listenerList =
         new EventListenerList();


    public ObjectTreeModel()
    {
        root = null;
    }


    public void setRoot(Variable v)
    {
        Variable oldRoot = v;
        root = v;
        fireTreeStructureChanged(oldRoot);
    }
```

```
public class ObjectInspectorFrame extends JFrame {
    public ObjectInspectorFrame() {
        setSize(400, 300);

        // Create root variable for 'this' frame
        Variable v = new Variable(getClass(), "this", this);

        // Build custom model and assign root
        ObjectTreeModel model = new ObjectTreeModel();
        model.setRoot(v);

        // Bind model to JTree and display
        JTree tree = new JTree(model);
        add(new JScrollPane(tree), BorderLayout.CENTER);
    }
}

public class ObjectTreeModel implements TreeModel
{
    private Variable root;
    private EventListenerList listenerList =
    new EventListenerList();


    public ObjectTreeModel()
    {
        root = null;
    }


    public void setRoot(Variable v)
    {
        Variable oldRoot = v;
        root = v;
        fireTreeStructureChanged(oldRoot);
    }

   public Object getRoot()
    {
        return root;
    }

    public int getChildCount(Object parent)
    {
        return ((Variable) parent).getFields().size();
    }
```

Connecting TreeModel to JTree

```
public Object getChild(Object parent, int index)
    {
        ArrayList<Field> fields =
               ((Variable) parent).getFields();
        var f = (Field) fields.get(index);
        Object parentValue = ((Variable)
parent).getValue();
        try
        {
            return new Variable(f.getType(),
f.getName(),
            f.get(parentValue));
        }
        catch (IllegalAccessException e)
        {
            return null;
        }
    }

    public int getIndexOfChild(Object parent,
Object child)
    {
        int n = getChildCount(parent);
        for (int i = 0; i < n; i++)
            if (getChild(parent,
i).equals(child))
                return i;
        return -1;
    }

    public boolean isLeaf(Object node)
    {
        return getChildCount(node) == 0;
    }
```

```
public void valueForPathChanged(TreePath path, Object
newValue)
    {
    }

    public void addTreeModelListener(TreeModelListener l)
    {
        listenerList.add(TreeModelListener.class, l);
    }

    public void removeTreeModelListener(TreeModelListener l)
    {
        listenerList.remove(TreeModelListener.class, l);
    }

    protected void fireTreeStructureChanged(Object oldRoot)
    {
        var event = new TreeModelEvent(this, new Object[]
{ oldRoot });
        for (TreeModelListener l :
listenerList.getListeners(TreeModelListener.class))
            l.treeStructureChanged(event);
    }
}
```