Article: Extending JTree capabilities

by Ulrich Hilger, August 5, 2005

With class `JTree` of the Java Swing package a powerful visual component to work with all kinds of hierarchical data is available. Some features however need to be built individually to take full advantage of the power of `JTree`.

This article covers the following ways to enhance `JTree`

Part I: Using drag and drop on `JTree`

- switch on drag and drop for class `JTree`
- move tree nodes of a `JTree` using drag and drop

Part II: Customizing `JTree` actions during drag

- automatic scrolling during drag
- automatic expansion of tree nodes during drag
- automatic highlighting and drawing of an insertion mark during drag
- custom image drawing during drag

Part III: In place editing for tree nodes

- using a tree cell editor to allow in place text editing for nodes with custom user objects

Part IV: Summary with an example application

- how to utilize described features for an own visual component

Part I: Using drag and drop in JTree

Switching on drag and drop for JTree

The typical mechanism of Swing to switch on drag and drop for `JTree` is to provide a transfer handler and to call method `setDragEnabled` as in

```
JTree tree = new JTree();
TransferHandler handler = new MyCustomTransferHandler();
tree.setTransferHandler(handler);
tree.setDragEnabled(true);
```

switching on drag and drop

An own implementation of class `TransferHandler` is required to enable drag and drop as indicated in above example with class `MyCustomTransferHandler`. A transfer handler is used to control the drag and drop operation such as to recognize drag gestures and to handle drag and drop events.

Moving tree nodes with drag and drop

Requirements

To enable a `JTree` to handle the move of tree nodes via drag and drop an own extension of class `TransferHandler` is required to

- create a transferable class capable to transport tree nodes
- handle the start of a drag operation by creating a transport object for selected tree paths (creating an instance of our transferable class, that is)
- handle the drop operation by transporting respective node from the node it was dragged from to the node it was dropped to.

Implementation of these requirements is discussed in the following part.

Creating a Transferable

An object that implements interface `Transferable` can be used to provide data for a transfer operation. For the purpose of moving a tree node or a group of selected tree paths, a very generic implementation of interface `Transferable` as shown below is sufficient.

```
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.UnsupportedFlavorException;
import java.io.IOException;

public class GenericTransferable implements Transferable {
  public GenericTransferable(Object data) {
    super();
    this.data = data;
  }
  public DataFlavor[] getTransferDataFlavors() {
    return flavors;
  }
  public boolean isDataFlavorSupported(DataFlavor flavor) {
    return true;
  }
  public Object getTransferData(DataFlavor flavor)
    throws UnsupportedFlavorException, IOException {
    return data;
  }
  private Object data;
  private static final DataFlavor[] flavors = new DataFlavor[1];
  static {
    flavors[0] = DataFlavor.stringFlavor;
  }
}
```

our implementation of class Transferable

Above class `GenericTransferable` - as shown completely commented in [2] as well - is constructed simply with an instance of class `Object` as a paramter representing the data that is to be transferred. The object can be anything thus an array of selected tree paths (`TreePath[]`) as well. The object passed to the constructor is stored privately and returned with method getTransferData() again.

Although class `GenericTransferable` implements `DataFlavor.stringFlavor` as the supported flavor it is ignored in the context of this implementation.

Handling the start of the drag operation

Having a `Transferable` that wraps tree paths we can go on to implement what our custom `TransferHandler` should do when a drag is initiated by the user. For this purpose method `TransferHandler.createTransferable` is used as follows

```
  protected Transferable createTransferable(JComponent c) {
    Transferable t = null;
    if(c instanceof JTree) {
      t = new GenericTransferable(((JTree) c).getSelectionPaths());
    }
    return t;
  }
```

implementation of createTransferable

Our implementation of method `createTransferable` simply finds out whether or not the given component is a `JTree` and finds all tree paths currently selected in respective tree. The resulting array of tree paths is passed to the constructor of `GenericTransferable`.

All other actions involved with what to do with the transferable during drag and what to do during drag in general are handled by the standard implementation of drag and drop in Swing.

Handling the drop operation

As soon as the user releases the mouse button during a drag operation method `TransferHandler.exportDone` is invoked. We need to override method `exportDone` as follows so that our `Transferable` containig the dragged tree paths is handled in a meaningful way.

```
protected void exportDone(JComponent source, Transferable data,
                                                  int action) {
    if(source instanceof JTree) {
      JTree tree = (JTree) source;
      DefaultTreeModel model = (DefaultTreeModel) tree.getModel();
      TreePath currentPath = tree.getSelectionPath();
      if(currentPath != null) {
        MutableTreeNode targetNode = (MutableTreeNode)
                    currentPath.getLastPathComponent();
        try {
          TreePath[] movedPaths = (TreePath[])
               data.getTransferData(DataFlavor.stringFlavor);
          for(int i = 0; i < movedPaths.length; i++) {
            MutableTreeNode moveNode = (MutableTreeNode)
                           movedPaths[i].getLastPathComponent();
            if(!moveNode.equals(targetNode)) {
              model.removeNodeFromParent(moveNode);
              model.insertNodeInto(moveNode, targetNode,
                                  targetNode.getChildCount());
            }
          }
        } catch (UnsupportedFlavorException e) {
          e.printStackTrace();
        } catch (IOException e) {
          e.printStackTrace();
        }
}
```

method `exportDone` as used for our purposes

To handle a drop operation above method finds out if a node is curerntly selected and picks the selected node as the target node for the drop operation. It then fetches all tree paths from the transferable and moves each to the new parent node.

Resulting TransferHandler

```
import java.awt.datatransfer.DataFlavor;
import java.awt.datatransfer.Transferable;
import java.awt.datatransfer.UnsupportedFlavorException;
import java.io.IOException;
import javax.swing.JComponent;
import javax.swing.JTree;
import javax.swing.TransferHandler;
import javax.swing.tree.DefaultTreeModel;
import javax.swing.tree.MutableTreeNode;
import javax.swing.tree.TreePath;

public class NodeMoveTransferHandler extends TransferHandler {
  public NodeMoveTransferHandler() {
    super();
  }
  protected Transferable createTransferable(JComponent c) {
    Transferable t = null;
    if(c instanceof JTree) {
      t = new GenericTransferable(((JTree) c).getSelectionPaths());
    }
    return t;
  }
  protected void exportDone(JComponent source, Transferable data, int action) {
    if(source instanceof JTree) {
      JTree tree = (JTree) source;
      DefaultTreeModel model = (DefaultTreeModel) tree.getModel();
      TreePath currentPath = tree.getSelectionPath();
      if(currentPath != null) {
        MutableTreeNode targetNode = (MutableTreeNode)
                          currentPath.getLastPathComponent();
         try {
           TreePath[] movedPaths = (TreePath[])
               data.getTransferData(DataFlavor.stringFlavor);
           for(int i = 0; i < movedPaths.length; i++) {
             MutableTreeNode moveNode = (MutableTreeNode)
                         movedPaths[i].getLastPathComponent();
             if(!moveNode.equals(targetNode)) {
                model.removeNodeFromParent(moveNode);
                model.insertNodeInto(moveNode, targetNode,
                                 targetNode.getChildCount());
             }
           }
         } catch (UnsupportedFlavorException e) {
            e.printStackTrace();
         } catch (IOException e) {
            e.printStackTrace();
         }
       }
     }
     super.exportDone(source, data, action);
  }
  public int getSourceActions(JComponent c) {
    return TransferHandler.MOVE;
  }
}
```

our implementation of class TransferHandler

The resulting `TransferHandler` implementing the mentioned parts can be found with class `NodeMoveTransferHandler` at [1]. Without comments the listing shows that our custom transfer handler is comparably simple.

Part II: Customizing JTree actions during drag

With the additions described previously a `JTree` can be extended to support structural changes using drag and drop. However, the drag operation still lacks some features which are important to make the drag and drop operation really supportive. So far the tree does not

- scroll automatically when a node is dragged to a tree boundary making it impossible to drag a node to a node not currently visible
- expand nodes as a node is dragged over a collapsed node making it impossible to drag a node to a child node of a collapsed node.

Class DropTarget to the rescue

Fortunately there is another Java construct that can be taken for the rescue. `JTree` accepts class `DropTarget` or subclasses thereof to implement extended drag and drop features.

The general approach to implement the mentioned functions is to create a subclass of class `DropTarget`. `DropTarget` implements interface `DropTargetListener` which has methods that are constantly called during a drag operation.

By creating a class `TreeDropTarget` as shown in [3] we can add methods as appropriate and have these methods called whenever the relevant action occurs during a drag operation.

The following part of this artice explains how to extend `JTree` with the missing functionality using class `DropTarget`. Unlike to our extensions to `Transferable` and `TransferHandler` our custom implementation of class `DropTarget` is comparably lenghty which is why this article does not show the complete listing of it. Please refer to [3] instead.

Implementing automatic scrolling

Automatic scrolling means to scroll the visible part of the tree content to always show the node a currently dragged node is moved to. Scrolling needs to happen automatically because the user has to hold down the mouse button to continue a drag operation. Releasing the mouse button for scrolling would interrupt the drag operation and lead to a drop at an unwanted location.

Whenever the user moves a tree node during a drag operation, method `dragOver` of interface `DropTargetListener` is called by the default Swing drag and drop process. Consequently we implement method `dragOver` in our custom `DropTarget` as follows

```
public void dragOver(DropTargetDragEvent dtde) {
  JTree tree = (JTree) dtde.getDropTargetContext().getComponent();
  Point loc = dtde.getLocation();
   ...
  autoscroll(tree, loc);
   ...
  super.dragOver(dtde);
}
```

implementation of method dragOver of interface DropTargetListener

This way our own method `autoscroll` is called every time a node is moved during a drag operation. Inside our own method autoscroll we check whether or not the user has moved the currently dragged tree node outside a certain part of the visible rectancle of respective tree.

```
private void autoscroll(JTree tree, Point cursorLocation) {
  Insets insets = getAutoscrollInsets();
  Rectangle outer = tree.getVisibleRect();
  Rectangle inner = new Rectangle(
      outer.x+insets.left,
      outer.y+insets.top,
      outer.width-(insets.left+insets.right),
      outer.height-(insets.top+insets.bottom));
  if (!inner.contains(cursorLocation))  {
    Rectangle scrollRect = new Rectangle(
        cursorLocation.x-insets.left,
        cursorLocation.y-insets.top,
        insets.left+insets.right,
        insets.top+insets.bottom);
    tree.scrollRectToVisible(scrollRect);
  }
}
```

our autoscroll implementation

When the drag currently happens outside an inner part of the visible rectangle, a new rectangle is created representing the new new visible region and finally method `tree.scrollRectToVisble` is called to make that new rectangle visible.

Implementing automatic node expansion

As mentioned previously it is not possible to drag a node to a child node of a collapsed node without additional effort. A mechanism to expand a collapsed node automatically whenever a node is dragged over it is necessary so that a node can be dragged to one of such nodes otherwise hidden children.

The same mechanism as shown for automatic scrolling is used for automatic node expansion again using method `dragOver`.

```
public void dragOver(DropTargetDragEvent dtde) {
  JTree tree = (JTree) dtde.getDropTargetContext().getComponent();
  Point loc = dtde.getLocation();
   ...
  updateDragMark(tree, loc);
   ...
  super.dragOver(dtde);
}
```

We call another custom method of our own which serves multiple functions this time. Method `updateDragMark` either

- marks a node over which another node is currently dragged,
- expands such node, when it is collapsed and
- draws an insert marker when the dragged node currently is between two nodes

How the above three functions are achieved in methods `markNode` and `paintInsertMarker` can be seen in [3] in greater detail.

Method `updateDragMark` looks as follows

```
public void updateDragMark(JTree tree, Point location) {
    int row = tree.getRowForPath(
             tree.getClosestPathForLocation(location.x, location.y));
    TreePath path = tree.getPathForRow(row);
    if(path != null) {
      Rectangle rowBounds = tree.getPathBounds(path);
      int rby = rowBounds.y;
      int topBottomDist = insertAreaHeight / 2;
      Point topBottom = new Point(
               rby - topBottomDist, rby + topBottomDist);
      if(topBottom.x <= location.y && topBottom.y >= location.y) {
        paintInsertMarker(tree, location);
      }
      else {
        markNode(tree, location);
      }
    }
  }
```

method updateDragMark

The method takes the current mouse pointer location from the `DropTargetDragEvent` and finds the rectangle of the tree path closest to that location. It then decides if the mouse is located over a node or rather between two nodes and calls methods `markNode` or `paintInsertMarker` respectively.

Drawing a custom image during drag

When a tree node is dragged the default image does not show the actual node. Especially with automatic scrolling this may lead to the dragged node being scrolled out of the visible region along with its original location making it impossible to see what actually is dragged. To still indicate the dragged node in this situation an own image paint method is required.

The same mechanism as shown previously is used to draw a custom image again using method `dragOver`.

```
public void dragOver(DropTargetDragEvent dtde) {
  JTree tree = (JTree) dtde.getDropTargetContext().getComponent();
  Point loc = dtde.getLocation();
   ...
  paintImage(tree, loc);
   ...
  super.dragOver(dtde);
}
```

An implementation of method paintImage as called above can be fairly simple as seen below

```
  private final void paintImage(JTree tree, Point pt) {
    BufferedImage image = handler.getDragImage(tree);
    if(image != null) {
      tree.paintImmediately(rect2D.getBounds());
      rect2D.setRect((int) pt.getX()-15,
         (int) pt.getY()-15,image.getWidth(),image.getHeight());
      tree.getGraphics().drawImage(image,
         (int) pt.getX()-15,(int) pt.getY()-15,tree);
    }
  }
```

method paintImage

As with method `updateDragMark` described previously method `paintImage` takes the current mouse location from the `DropTargetDragEvent` and draws a custom image.

The rather complicated part to create a custom image is not located in class `TreeDropTarget`. The only part that ‘knows’ what is being dragged during a drag operation is the transfer handler so to draw a custom image built from the node that is currently dragged an extension to our transfer handler is necessary.

Class `NodeMoveTransferHandler` needs to be extended to remember the currently moved node whenever a transferable is created indicating a drag being initiated.

```
  protected Transferable createTransferable(JComponent c) {
    Transferable t = null;
    if(c instanceof JTree) {
      JTree tree = (JTree) c;
      t = new GenericTransferable(tree.getSelectionPaths());
      dragPath = tree.getSelectionPath();
      if (dragPath != null) {
        draggedNode =
              (MutableTreeNode) dragPath.getLastPathComponent();
      }
    }
    return t;
  }
...
...
  private MutableTreeNode draggedNode;
  private TreePath dragPath;
```

extension of method createTransferable to remember the dragged node

To produce an image from the currently moved node class `NodeMoveTransferHandler` now can be extended as shown below

```
public BufferedImage getDragImage(JTree tree) {
  BufferedImage image = null;
  try {
    if (dragPath != null) {
      Rectangle pathBounds = tree.getPathBounds(dragPath);
      TreeCellRenderer r = tree.getCellRenderer();
      DefaultTreeModel m = (DefaultTreeModel)tree.getModel();
      boolean nIsLeaf = m.isLeaf(dragPath.getLastPathComponent());
      JComponent lbl = (JComponent)r.getTreeCellRendererComponent(
           tree,draggedNode, false ,
           tree.isExpanded(dragPath), nIsLeaf, 0,false);
      lbl.setBounds(pathBounds);
      image = new BufferedImage(lbl.getWidth(), lbl.getHeight(),
                   java.awt.image.BufferedImage.TYPE_INT_ARGB_PRE);
      Graphics2D graphics = image.createGraphics();
      graphics.setComposite(
             AlphaComposite.getInstance(
                       AlphaComposite.SRC_OVER, 0.5f));
      lbl.setOpaque(false);
      lbl.paint(graphics);
      graphics.dispose();
    }
  }
  catch (RuntimeException re) {}
  return image;
}
```

method getDrawImage

Method `getDragImage` uses the cell renderer of the tree to get the default component for rendering the dragged node (a `JLabel` by default). It then builds a buffered image to draw to and adds a semi transparent drawing effect. The label denoting the dragged node is drawn to that buffered image and the result is returned to the caller being method `paintImage` of class `TreeDropTarget`.

Finally to let our `TreeDropTarget` know which transfer handler to use for getting an image to draw durng dragging we require the transfer handler in the constructor of `TreeDropTarget` (see [3]).

Part III: In place editing for tree nodes

During manipulation of a hierarchical data structure in a `JTree` a user should be able to add new elements to the tree. A simple approach for adding new tree nodes is to programmatically add a new node to the currently selected node and to name the new node with a generic expression such as “new node 1”. In a second step the user then could change the name of the new node to something meaningful. Ideally the name change would commence using in place editing, i.e. a little editor box pops up around the new node to type in a name directly.

So the steps for the process we look at are as follows

1. Find the node that is currently selected
2. create a new node with a generic name such as “new node 1”
3. add the new node to the currently selected node
4. switch on in place editing for the newly added node for the user to enter the true name

So far we are still perfectly supported by Swing as `JTree` offers a built in mechanism illustrated by the following code snippet

```
private void addNode() {
  TreePath selectedPath = myTree.getSelectionPath();
  if(selectedPath != null) {
    Object o = selectedPath.getLastPathComponent();
    if(o != null) {
      DefaultMutableTreeNode selectedNode = (DefaultMutableTreeNode) o;
      DefaultMutableTreeNode newChild =
                            new DefaultMutableTreeNode(“new node 1”);
      ((DefaultTreeModel) myTree.getModel()).insertNodeInto(
            newChild, selectedNode, selectedNode.getChildCount());
      TreePath newPath = selectedPath.pathByAddingChild(newChild);
      myTree.setSelectionPath(newPath);
      myTree.startEditingAtPath(newPath);
    }
  }
}
```

Code snippet: a generic method to add a node

With method `addNode` a node is added to a tree referenced by `myTree`. The last three lines build a selection path for the newly added node and switch on in place editing for it.

Warning: Possible data loss

The drawback of this method is that it can handle only the default implementation of tree nodes having strings as user objects. The default tree cell editor returns a string as the user object consequently. On the contrary nodes of a typical tree implementation hold a custom user object with more than a string in it. When switching to in place editing for such a node the user object is overwritten by the string returned by the default tree cell editor. This way a custom user object is gone forever when using in place editing on its name.

Using a custom tree cell renderer

Fortunately the Swing team built a method to set another cell editor with `JTree.setCellEditor()` so we can extend the default `TreeCellEditor` class with a way to keep the original user object and still allow for name changes in place of the respective tree node.

The general approach is to extend `TreeCellEditor` to

- intercept the moment when in place editing starts to temporary save the user object
- intercept the moment when in place editing ends, i.e. when the edited value is returned to return our user object with the changed name instead of a plain string

Our extension `UserTreeCellEditor` follows this approach (relevant parts highlighted in bold characters):

```
import java.awt.Component;

import javax.swing.JTree;
import javax.swing.tree.DefaultMutableTreeNode;
import javax.swing.tree.DefaultTreeCellEditor;
import javax.swing.tree.DefaultTreeCellRenderer;

import com.lightdev.lib.ui.util.HierarchicalItem;

public class UserTreeCellEditor extends DefaultTreeCellEditor {
  public UserTreeCellEditor(JTree tree, DefaultTreeCellRenderer renderer) {
    super(tree, renderer);
  }
  public Object getCellEditorValue() {
    Object returnValue = null;
    Object value = super.getCellEditorValue();
    if(item == null) {
      returnValue = value;
    }
    else {
      item.setData(value);
      returnValue = item;
    }
    return returnValue;
  }
  public Component getTreeCellEditorComponent(JTree tree, Object value,
              boolean isSelected, boolean expanded, boolean leaf, int row) {
    if(value instanceof DefaultMutableTreeNode) {
      DefaultMutableTreeNode node = (DefaultMutableTreeNode) value;
      Object userObject = node.getUserObject();
      if(userObject instanceof HierarchicalItem) {
        item = (HierarchicalItem) node.getUserObject();
      }
    }
    return super.getTreeCellEditorComponent(
               tree, value, isSelected, expanded, leaf, row);
  }
  private HierarchicalItem item;
}
```

extension of class TreeCellEditor to support in place editing of custom user objects

Our class `UserTreeCellEditor` overrides method `getTreeCellEditorComponent` to intercept the start of an edit process and method `getCellEditorValue` to intercept the end of respective edit process.

UserTreeCellEditor assumes a class of type HierarchicalItem as the user object of tree nodes being edited through our tree cell editor as shown below.

```
public interface HierarchicalItem {
  public abstract void setData(Object data);
  public abstract Object getData();
  public abstract void setId(Object id);
  public abstract Object getId();
  public abstract Object getParentId();
  public abstract void setParentId(Object parentId);
  public abstract boolean isRoot();
}
```

a possible user object of nodes being edited with UserTreeCellEditor

Assuming we do not want to lose the user object found in the tree node method getTreeCellEditor keeps a reference of the user object before it returns an editor component. Similarly method getCellEditorValue stores the edited value inside the user object and returns the entire user object instead of only the edited string (assuming the user object's toString method returns the edited value).

Part IV: Summary with example application RegionEditor

For an example about how to utilize the enhancements to `JTree` from this article let's assume we need a user interface component that allows to create, edit and remove geographical regions using a tree component. For this example we build a Java class that can be started as an application.

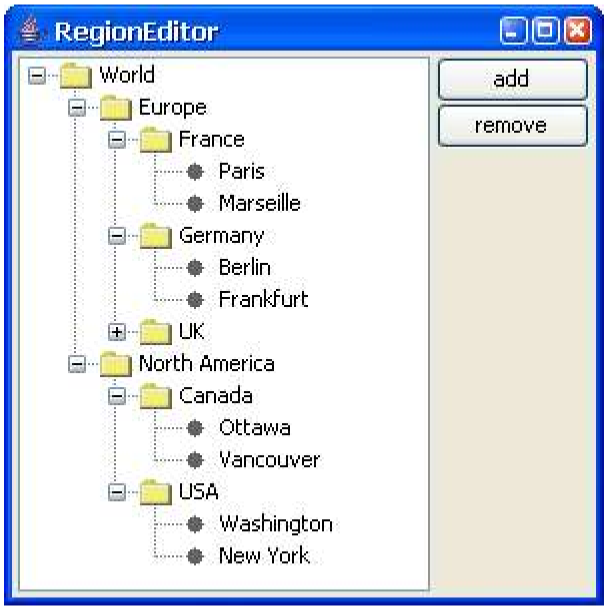
Our example application `RegionEditor` as available completely at [5] builds a single frame that has the mentioned edit functionality. To create a tree with the features described in this article it uses method `getTree`.

```
private JTree getTree() {
    regTree = new JTree(getSampleTreeRoot());
    regTree.setShowsRootHandles(true);
    regTree.setEditable(true);
    regTree.setCellEditor(new UserTreeCellEditor(
            regTree, (DefaultTreeCellRenderer) regTree.getCellRenderer()));
    NodeMoveTransferHandler handler = new NodeMoveTransferHandler();
    regTree.setTransferHandler(handler);
    regTree.setDropTarget(new TreeDropTarget(handler));
    regTree.setDragEnabled(true);
    return regTree;
}
```

method getTree of our example application

Method `RegionEditor.getTree()` utilizes classes `UserTreeCellEditor`, `NodeMoveTransferHandler` and `TreeDropTarget` to create and return an enhanced tree component.

The entire example application produces a GUI as shown below



the GUI of example application RegionEditor

The example application allows to create a region with the add button. It switches to in place editing as described for a newly created node. Exisiting nodes can be dragged and dropped to change the region structure (for cases when a part is refined, e.g. the user adds counties to USA and then moves cities to the counties they belong).

About the author

Ulrich Hilger develops software based on more than 25 years of experience in the software industry. Under the label Light Development platform independent solutions based on Java technology are the main activity sector since 1999. Light Development products are used in many industries especially in the area of documentation and class libraries. High quality and entirely own production over the whole value chain are hallmarks of the label.

References

[1] NodeMoveTransferHandler.java
http://articles.lightdev.com/tree/code/NodeMoveTransferHandler.java

[2] GenericTransferable.java
http://articles.lightdev.com/tree/code/GenericTransferable.java

[3] TreeDropTarget.java
http://articles.lightdev.com/tree/code/TreeDropTarget.java

[4] UserTreeCellEditor.java
http://articles.lightdev.com/tree/code/UserTreeCellEditor.java

[5] RegionEditor example application
http://articles.lightdev.com/tree/RegionEditor.zip

[6] This article on the web
http://articles.lightdev.com/tree/tree_article.pdf

[7] Light Development: Ulrich Hilger's homepage on the web:
http://www.lightdev.com