**The JGraph Tutorial**

Copyright © 2003 by Gaudenz Alder

**Table of Contents**

[How to use Graphs](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/t1.html#AEN18)

[Examples](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/x438.html)

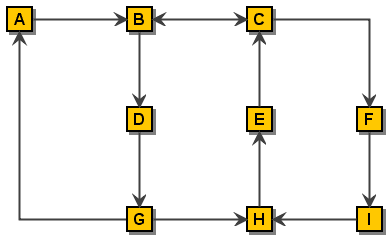
[References](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/x669.html)

This document provides an experimental analysis of the JGraph component, based on working examples. Source code from the JGraphpad application will be used, together with two additional examples. One example is a diagram editor, the other is a GXL to SVG file converter for batch processing an automatic layout.

This document is *not* a specification of the API, and it does *not* provide an in-depth study of the component's architecture. The target readers are developers who need a well-documented collection of working examples that demonstrate how to use the JGraph component in a custom application.

**How to use Graphs**

With the JGraph class, you can display objects and their relations. A JGraph object doesn't actually contain your data; it simply provides a view of the data. Like any non-trivial Swing component, the graph gets data by querying its data model. Here's a picture of a graph:



**Figure 1. A directed graph (*Digraph*)**

As the preceding figure shows, JGraph displays its data by drawing individual elements. Each element displayed by the graph contains exactly one item of data, which is called a *cell*. A cell may either be a vertex, an edge or a port. Vertices have zero or more neighbors, and edges have one or no source and target vertex. Each cell has zero or more children, and one or no parent. (Instances of ports are always children of vertices.)

The rest of this tutorial discusses the following topics:

* Creating a Graph
* Customizing a Graph
* Responding to Interaction
* Customizing a Graph's Display
* Dynamically changing a Graph

Two working examples are provided:

* Client-side Example

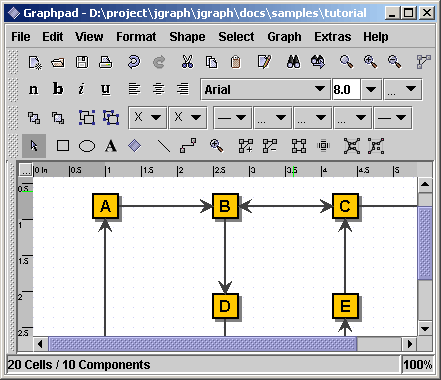
Implements a simple diagram editor with a custom graph model, marquee handler, tooltips, command history and a popup menu.

* Server-side Example

Uses the JGraph component for converting a GXL file into an SVG file, and applying a simple circle-layout.

**Creating a Graph**

Here is a picture of an application that uses a graph in a scroll pane:



**Figure 2. An application that uses a graph in a scrollpane**

The following code creates a JGraph object:

|  |
| --- |
| JGraph graph = new JGraph();  ...  JScrollPane scrollPane = new JScrollPane(graph) |

The code creates an instance of JGraph and puts it in a scroll pane. JGraph's constructor is called with no arguments in this example. Therefore, JGraph will use an instance ofDefaultGraphModel, create an instance of GraphView for it, and add a sample graph to the model.

JGraph's default implementations define the following set of keyboard bindings:

* Alt-Click forces marquee selection if over a cell
* Shift- or Ctrl-Select extends or toggles the selection
* Shift-Drag constrains the offset to one direction
* Ctrl-Drag clones the selection
* Doubleclick or F2 starts editing

To summarize, you can create a graph by invoking the JGraph constructor. You should probably put the graph inside a scroll pane, so that the graph won't take up too much space. You don't have to do anything to support cell editing, selection, marquee selection, vertex and edge resizing and moving.

**Customizing a Graph**

JGraph offers the following forms of interactions:

* In-place editing
* Moving
* Cloning
* Sizing
* Bending (Adding/Removing/Moving edge points)
* Establishing Connections
* Removing Connections

All interactions can be disabled using setEnabled(false). In-place editing is available for both, vertices and edges, and may be disabled using setEditable(false). The number of clicks that triggers editing may be changed using setEditClickCount.

Moving, cloning, sizing, and bending, establishing connections and disconnecting edges may be disabled using the respective methods, namely setMoveable, setCloneable, setSizeable,setBendable, setConnectable and setDisconnectable on the graph instance.

The model offers finer control of connection establishment and disconnection based on the acceptsSource and acceptsTarget methods. By overriding these methods, you can decide for each edge, port pair if it is valid with respect to the edge's source or target. (Before an edge is disconnected from a port, the respective method is called with the port set to null to check if disconnection is allowed.)

CellViews offer yet another level of control in that they allow/disallow being edited, moved, cloned, resized, and shaped, or connected/disconnected to or from other cells. (Note: In the context of multiple views, a cell may be connectable in one view, and not connectable in another.)

There are a number of additional methods to customize JGraph, for example, setMinimumMove to set the minimum amount of pixels before a move operation is initiated, and setSnapSizeto define the maximum distance from a cell to be selected.

With setDisconnectOnMove you can indicate if the selection should be disconnected from the unselected graph when a move operation is initiated, setDragEnabled enables/disables the use of Drag-and-Drop, and setDropEnabled sets if the graph accepts Drops from external sources. (The latter also affects the clipboard, in that it allows/disallows to paste data from exernal sources.)

**Responding to Interaction**

You can either respond to mouse events, or to events generated by JGraph. JGraph offers the following notifications:

* Changes to the model
* Changes to the view
* Changes to the selection
* Undoable edit happened

**What's "Undoable edit happened"?**

JGraph is compatible to Swing's Undo-Support. Each time the user performs an action, the model dispatches an edit that describes the change. The edit provides an undo method to undo the change.

If undo is called on an edit, the model fires a GraphModelEvent, but it does not fire an UndoableEditEvent. The latter only fires if JGraph wants to indicate that an edit was added to the command history, which is not the case for an undo. (In the context of multiple views, you must use an instance of GraphUndoManager to ensure correctness.)

**Responding to Mouse Events**

For detection of double-clicks or when a user clicks on a cell, regardless of whether or not it was selected, a MouseListener should be used in conjunction with getFirstCellForLocation. The following code prints the label of the topmost cell under the mouse pointer on a doubleclick. (The getFirstCellForLocation method scales its arguments.)

|  |
| --- |
| // MouseListener that Prints the Cell on Doubleclick  graph.addMouseListener(new MouseAdapter() {  public void mousePressed(MouseEvent e) {  if (e.getClickCount() == 2) {  // Get Cell under Mousepointer  int x = e.getX(), y = e.getY();  Object cell = graph.getFirstCellForLocation(x, y);  // Print Cell Label  if (cell != null) {  String lab = graph.convertValueToString(cell);  System.out.println(lab);  }  }  }  }); |

**Responding to Model Events**

If you are interested in handling model notifications, implement the GraphModelListener interface and add the instance using the method addGraphModelListener. The listeners are notified when cells are inserted, removed, or when the label, source, target, parent or children of an object have changed. (Make sure to also add an Observer to GraphView in order to be notified of all possible changes to a graph!)

The following code defines a listener that prints out information about the changes to the model, and adds the listener to the graph's model:

|  |
| --- |
| // Define a Model Listener  public class ModelListener implements GraphModelListener {  public void graphCellsChanged(GraphModelEvent e) {  System.out.println("Change: "+e.getChange());  }  }  // Add an Instance to the Model  graph.getModel().addGraphModelListener(new ModelListener()); |

**Responding to View Events**

Visual modifications are typically handled by the GraphView, which extends the Observable class. To respond to view-changes, implement the Observer interface and add it to the view using the method addObserver. Observers are notified when the size, position, color etc. of a cell view has changed. (Note: If the model's isAttributeStore returns true, then the view is bypassed, and all attributes are stored in the model.)

|  |
| --- |
| // Define an Observer  public class ViewObserver implements Observer {  public void update(Observable o, Object arg) {  System.out.println("View changed: "+o);  }  }  // Add an Instance to the View  graph.getView().addObserver(new ViewObserver()); |

**Responding to Selection Changes**

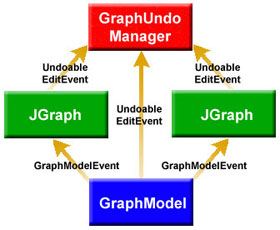
The following code prints out selection events:

|  |
| --- |
| // Define a Selection Listener  public class MyListener implements GraphSelectionListener {  public void valueChanged(GraphSelectionEvent e) {  System.out.println("Selection changed: "+e);  }  }  // Add an Instance to the Graph  graph.addGraphSelectionListener(new MyListener()); |

The preceding code creates an instance of MyListener, which implements the GraphSelectionListener interface, and registers it on the graph.

**Responding to Undoable Edits**

To enable Undo-Support, a GraphUndoManager must be added using addUndoableEditListener. The GraphUndoManager is an extension of Swing's UndoManager that maintains a command history in the context of multiple views. You can safely use an instance of UndoManager if your graph has only one view. Otherwise, a GraphUndoManager must be used for correct behaviour.



**Figure 3. Multiple Views**

The figure above shows a graph model in the context of multiple views. From a logical point of view, both, JGraph and the GraphModel fire UndoablEditEvents. However, in reality only the model supports these events and the view uses this support to dispatch its own UndoableEdits. The view is updated based on GraphModelEvents.

**Extending the Default Handlers**

JGraph provides two high-level listeners that are used to control mouse and datatransfer functionality. By overriding these, you can gain full control of marquee selection and datatransfer, that is, how cells are imported and exported via Drag-and-Drop and the clipboard. The client-side example at the end of this Tutorial provides such a custom handler. The custom marquee handler in this example is used to establish connections between cells by dragging ports.

**Customizing a Graphs Display**

JGraph performs some look-and-feel specific painting. You can customize this painting in a limited way. For example, you can modify the grid using setGridColor and setGridSize, and you can change the handle colors using setHandleColor and setLockedHandleColor. The background color may be changed using setBackground.

**Subclassing Renderers**

If you want finer control over the rendering, you can subclass one of the default renderers, and override its paint-method. A renderer is a Component-extension that paints a cell based on its attributes. Thus, neither JGraph nor its look-and-feel-specific implementation actually contain the code that paints a cell. Instead, JGraph uses the cell renderer's painting code. A new renderer may be associated with a cell by overriding the getRendererComponent method of the corresponding CellView, or the getRenderer method for extensions of the AbstractCellView class.

**Adding new Cell Types to a Graph**

The following code was taken from JGraphpad (http://jgraph.sourceforge.net/pad/jgraphpad.jnlp) to illustrate how to add new cell types and renderers. The code adds an oval vertex to the graph. The easiest way to do this is by extending JGraph. Since JGraph implements the CellViewFactory interface, it is in charge of creating views.

When creating a view, JGraph assumes a cell is a vertex if it is not an instance of Edge or Port, and calls the createVertexView method. Thus, we only need to override this method to identify an oval vertex (based on a typetest) and return the corresponding view.

|  |
| --- |
| // Overrides JGraph.createVertexView  protected VertexView createVertexView(Object v,  GraphModel model,  CellMapper cm) {  // Return an EllipseView for EllipseCells  if (v instanceof EllipseCell)  return new EllipseView(v, model, cm);  // Else Call Superclass  return super.createVertexView(v, model, cm);  } |

The oval vertex is represented by the EllipseCell class, which is an extension of the DefaultGraphCell class, and offers no additional methods. It is only used to distinguish oval vertices from normal vertices.

|  |
| --- |
| // Define EllipseCell  public class EllipseCell extends DefaultGraphCell {  // Empty Constructor  public EllipseCell() {  this(null);  }  // Construct Cell for Userobject  public EllipseCell(Object userObject) {  super(userObject);  }  } |

The EllipseView is needed to define the special visual aspects of an ellipse. It contains an inner class which serves as a renderer that provides the painting code. The view and renderer are extensions of the VertexView and VertexRenderer classes, respectively. The methods that need to be overridden are getPerimeterPoint to return the perimeter point for ellipses, getRenderer to return the correct renderer, and the renderer's paint method.

|  |
| --- |
| // Define the View for an EllipseCell  public class EllipseView extends VertexView {  static EllipseRenderer renderer = new EllipseRenderer();  // Constructor for Superclass  public EllipseView(Object cell, GraphModel model,  CellMapper cm) {  super(cell, model, cm);  }  // Returns Perimeter Point for Ellipses  public Point getPerimeterPoint(Point source, Point p) { ...  }  // Returns the Renderer for this View  protected CellViewRenderer getRenderer() {  return renderer;  }  // Define the Renderer for an EllipseView  static class EllipseRenderer extends VertexRenderer {  public void paint(Graphics g) { ... }  }  } |

The reason for overriding getRenderer instead of getRendererComponent is that the AbstractCellView class, from which we inherit, already provides a default implementation of this method that returns a configured CellViewRenderer, which in turn is retrieved through the method that was overridden.

**Adding Tooltips to a Graph**

Tooltips can be implemented by overriding JGraph's getToolTipText method, which is inherited from the JComponent class. The following displays the label of the cell under the mouse pointer as a tooltip.

|  |
| --- |
| // Return Cell Label as a Tooltip  public String getToolTipText(MouseEvent e) {  if(e != null) {  // Fetch Cell under Mousepointer  Object c = getFirstCellForLocation(e.getX(), e.getY());  if (c != null)  // Convert Cell to String and Return  return convertValueToString(c);  }  return null;  } |

The graph must be registered with Swing's TooltipManager to correctly display tooltips. This is done with the following code on startup:

|  |
| --- |
| ToolTipManager.sharedInstance().registerComponent(graph) |

**Customizing In-Place Editing**

In graphs that display complex structures, it is quite common to offer a property dialog instead of the simple in-place editing. To do this, the BasicGraphUI's startEditing and completeEditingmethods must be overridden. Then, in order to use this UI in a graph, the graph's updateUI method must be overridden, too:

|  |
| --- |
| // Define a Graph with a Custom UI  public class DialogGraph extends JGraph {  // Sets the Custom UI for this graph object  public void updateUI(){  // Install a new UI  setUI(new DialogUI());  invalidate();  }  } |

The DialogUI class takes the view's editor, and puts it in a dialog, which blocks the frame until the dialog is closed. The code for the DialogUI class is not printed here. It is included in the Tutorial.java file, which is available for download (see references at the end of this Tutorial).

**Dynamically changing a Graph**

In JGraph, either the model or the view is modified, or they are modified in parallel with a single transaction. When working on the model, objects that implement the GraphCell interface are used, whereas objects that implement the CellView interface are used in the context of a GraphView. GraphViews allow to edit CellViews, whereas GraphModels allow to insert, remove, and editGraphCells.

In this chapter, a DefaultGraphModel is used along with a graph that provides a view to the model. This way, we can clarify which methods belong to the model, and which belong to the view.

|  |
| --- |
| DefaultGraphModel model = new DefaultGraphModel();  JGraph graph = new JGraph(model); |

**Attributes**

JGraph separates the model and the view. The model is defined by the GraphModel interface, and contains objects that implement the GraphCell interface, whereas the view is represented by the GraphView class, and contains objects that implement the CellView interface. The mapping between cells and views is defined by the CellMapper interface:

http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/images/image004.png

**Figure 4. Mapping between GraphCells and CellViews**

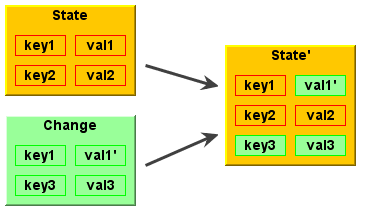
A model has zero or more views, and for each cell in the model, there exists exactly one CellView in each GraphView. The state of these objects is repesented by a map of key, value pairs. Each CellView combines the attirbutes from the corresponding GraphCell with its own attributes.

When combining the attributes from a GraphCell with the attributes from the CellView, the graph cell's attributes have precedence over the view's attributes. The special value attribute is in sync with the cell's user object.

**Dynamically Changing Attributes**

The state of a cell, and likewise of a view is represented by its attributes. In either case, the GraphConstants class is used to change the state in two steps:

1. Construct the object that constitutes the change
2. Execute the change on the model, or the graph view



**Figure 5. Using maps to change the state**

To construct the object that constitutes the change, a new map is created using the createMap method of the GraphConstants class. When a map is applied to a view's or cell's state, it does not replace the existing map. The entries of the new map are added or changed in-place. As a consequence, the GraphConstants class offers the setRemoveAttributes and setRemoveAllmethods, which are used to remove individual or all keys from the existing state. (Note: The notion of states using maps closely resembles the structure of XML-documents.)

**Automatically Changing Attributes**

The update method of the CellView interface is used to message the CellView when one of its attributes was changed programmatically, or one of its neighbours has changed attributes. Thus, the update method is a good place to automatically set attributes, like for example the points of an edge. To reflect changes to the view's corresponding cell, for example to point to the current source and target port, the refresh method is used.

**Working with the GraphModel**

You can think of the model as an access point to two independent structures: the graph structure and the group structure. The graph structure is based on the mathematical definition of a graph, ie. vertices and edges. The group structure is used to enable composition of cells, ie. parents and childs.

The graph structure is defined by the getSource and getTarget methods, which return the source and target port of an object. The port in turn is a child of a vertex, which is used as an indirection to allow multiple connection points.

The group structure is defined by the getChild, getChildCount, getIndexOfChild, and getParent methods. The objects that have no parents are called *roots*, and may be retrieved using thegetRootAt and getRootCount methods.

**Inserting a Vertex into the Model**

Here is a method that creates a new DefaultGraphCell and adds it to the model. The method adds two ports to the cell, and creates a map which maps the cells to their attributes. The attributes are in turn maps of key, value pairs, which may be accessed in a type-safe way by use of the GraphConstants class:

|  |
| --- |
| void insertVertex(Object obj, Rectangle bounds) {  // Map that Contains Attribute Maps  Map attributeMap = new Hashtable();  // Create Vertex  DefaultGraphCell cell = new DefaultGraphCell(userObject);  // Create Attribute Map for Cell  Map map = GraphConstants.createMap();  GraphConstants.setBounds(map, bounds);  // Associate Attribute Map with Cell  attributeMap.put(cell, map);  // Create Default Floating Port  DefaultPort port = new DefaultPort("Floating");  cell.add(port);  // Additional Port Bottom Right  int u = GraphConstants.PERCENT;  port = new DefaultPort("Bottomright");  // Create Attribute Map for Port  map = GraphConstants.createMap();  GraphConstants.setOffset(map, new Point(u, u));  // Associate Attribute Map with Port  attributeMap.put(port, map);  cell.add(port)  // Add Cell (and Children) to the Model  Object[] insert = new Object[]{cell};  model.insert(insert, null, null, attributeMap);  } |

The first argument to the insertVertex method is the user object - an object that contains or points to the data associated with the cell. The user object can be a string, or it can be a custom object. If you implement a custom object, you should implement its toString method, so that it returns the string to be displayed for that cell. The second argument represents the bounds of the vertex, which are stored as an attribute.

Note: The vertex is passed to the insert method without its children. The fact that parent-child relations are stored in the cells is used here to insert the children implicitly, without providing a ParentMap. (Future implementations should provide an additional argument to allow separate storage.)

The attributeMap argument is not used by the model. It is passed to the views to provide the attributes for the cell views to be created. The third parameter of the insert call can be used to provide properties, that is, attributes that are stored in the model.

**Finding the Port of a Vertex**

Since ports are treated as normal children in the model (using the model's group structure), the GraphModel interface may be used to find the "default port" of a vertex:

|  |
| --- |
| Port getDefaultPort(Object vertex, GraphModel model) {  // Iterate over all Children  for (int i = 0; i < model.getChildCount(vertex); i++) {  // Fetch the Child of Vertex at Index i  Object child = model.getChild(vertex, i);  // Check if Child is a Portif (child instanceof Port)  // Return the Child as a Port  return (Port) child;  }  // No Ports Found  return null;  } |

The code is not provided by the core API because it introduces the notion of a "default port", which is typically application dependent. (The code above uses the first port as the default port.)

**Inserting an Edge into the Model**

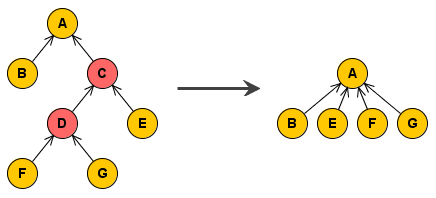
The following method creates a new DefaultEdge, and adds it to the model, along with the connections to the specified source and target port.

|  |
| --- |
| void insertEdge(Object obj, Port source, Port target) {  // Create Edge  DefaultEdge edge = new DefaultEdge(userObject);  // Create ConnectionSet for Insertion  ConnectionSet cs = new ConnectionSet(edge, source, target);  // Add Edge and Connections to the Model  Object[] insert = new Object[]{edge};  model.insert(insert, cs, null, null);  } |

Again, the first argument represents the user object of the cell, and the second and third argument specify the source and target port of the new edge. To insert connections into a graph model, an instance of ConnectionSet is required. The instance is used to collect the new ports and targets of edges, and execute the change as a single transaction.

**Removing Cells from the Model**

If a cell is removed from the model, the model checks if the cell has children, and if so, updates the group structure accordingly, that is, for all parents and children that are not to be removed. As a consequence, if a cell is removed *with* children, it can be reinserted using insert, that is, without providing the children or a ParentMap. If a cell is removed *without* children, the resulting operation is an "ungroup".



**Figure 6. Remove (ungroup) cells C and D.**

The figure above shows a group A, which contains the cell B, and a group C, which in turn contains E, a group D, which in turn contains F, and G. The second figure shows the group structure after the removal of cells C and D.

The following removes all selected cells, *including* all descendants:

|  |
| --- |
| // Get Selected Cells  Object[] cells = graph.getSelectionCells();  if (cells != null) {  // Remove Cells (incl. Descendants) from the Model  graph.getModel().remove(graph.getDescendants(cells));  } |

**Changing the Model**

The ConnectionSet and ParentMap classes are used to change the model. The attributes may be changed using a map that contains cell, attributes pairs. The attributes are in turn represented by maps of key, value pairs, which may be accessed in a type-safe way by using the GraphConstants class.

Let edge, port and vertex be instances of the DefaultEdge, DefaultPort and DefaultGraphCell classes, respectively; all contained in the model. This code changes the source of edge to port, the parent of port to vertex, and the user object of vertex to the String *Hello World*.

|  |
| --- |
| // Create Connection Set  ConnectionSet connectionSet = new ConnectionSet();  connectionSet.connect(edge, port, true);  // Create Parent Map  ParentMap parentMap = new ParentMap();  parentMap.addEntry(port, vertex);  // Create Properties for VertexMap  properties = GraphConstants.createMap();  GraphConstants.setValue(properties, "Hello World");  // Create Property Map  Map propertyMap = new Hashtable();  propertyMap.put(vertex, properties);  // Change the Model  model.edit(connectionSet, propertyMap, parentMap, null); |

The last argument of the edit call may be used to specify the initial edits that triggered the call. The edits specified are considered to be part of the transaction. (This is used to implement composite transactions, ie. transactions that change the model and the view in parallel.)

**Changing the View**

Each GraphCell has one or more associated CellViews, which in turn contain the attributes. The attributes of a CellView may be changed using editCells on the parent GraphView. In contrast to the propertyMap argument used before, and the attributeMap argument used to insert cells into the model, the attributeMap argument used here contains instances of the CellView class as keys. The attributes, again, are maps of key, value pairs.

The following changes the border color of vertex to black:

|  |
| --- |
| // Work on the Graph's View  GraphView v = graph.getView();  // Create Attributes for Vertex  Map attributes = GraphConstants.createMap();  GraphConstants.setBorderColor(attributes, Color.black);  // Get the CellView to use as Key  CellView cellView = v.getMapping(vertex, false);  // Create Attribute Map  Map attributeMap = new Hashtable();  attributeMap.put(cellView, properties);  // Change the View  v.editCells(attributeMap); |

**Creating a Data Model**

**Extending the Default Model**

The DefaultGraphModel provides three methods for subclassers, namely the acceptsSource and acceptsTarget method to allow or disallow certain connections to be established in a model, and the isAttributeStore method to gain control of the attributes that are changed by the UI. The first two methods are used in the GraphEd example in this tutorial, the latter is explained in detail in the paper. The isOrdered method is used to indicate if the model's order should be used instead of the view's, and the toBack and toFront methods are used to change this order. Note that the graph views methods with the same name are redirected to the model if the isOrdered method returns true.

**Providing a Custom Model**

If DefaultGraphModel doesn't suit your needs, then you will have to implement a custom data model. Your data model must implement the GraphModel interface. GraphModel accepts any kinds of objects as cells. It doesn't require that cells implement the GraphCell interface. You can devise your own cell representation.

|  |  |  |
| --- | --- | --- |
|  |  | [Next >>>](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/x438.html) |
|  |  | Examples |

|  |  |  |
| --- | --- | --- |
| **The JGraph Tutorial** | | |
| [<<< Previous](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/t1.html) |  | [Next >>>](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/x669.html) |

# Examples

## GXL to SVG Converter (Gxl2svg)

In this example, JGraph is used in a batch-processing environment, instead of a user interface. The input is a GXL file, which defines a graph with vertices and edges, but not its geometric pattern. The file is parsed into a graph model, and the vertices are arranged in a circle. The display is then stored as an SVG file, using the Batik library (http://xml.apache.org/batik/).

Here is the main method:

|  |
| --- |
| // Usage: java SVGGraph gxl-file svg-file  public static void main(String[] args) {  (...)  // Construct the graph to hold the ettributes  JGraph graph = new JGraph(new DefaultGraphModel());  // Read the GXL file into the model  *read(new File(args[0]), graph.getModel());*  // Apply Layout  *layout(graph);*  // Resize (Otherwise not Visible)  graph.setSize(graph.getPreferredSize());  (...)  // Write the SVG file  *write(graph, out);*  (...)  } |

### Read

The read method creates the cells defined in the GXL file, and constructs the arguments for the insert method, namely a list of cells, a connection set, and a map, from cells to attributes. To establish the connections, the vertices in the file are given unique IDs, which are referenced from the edge's source and target. (The ids map is used to look-up the vertices.)

|  |
| --- |
| public static void read(File f, GraphModel model) (...) {  (...)  // List for the new Cells  List newCells = new ArrayList();  // ConnectionSet to Specify Connections  ConnectionSet cs = new ConnectionSet();  // Map from Cells to AttributesHashtable  attributes = new Hashtable();  // Map from ID to Vertex  Map ids = new Hashtable(); |

In the innermost loop of the read method, cells are created based on the current XML element. (Let label point to the element's label, and type point to the element's type, which is "node" for vertices and "edge" for edges.)

The following creates a vertex with a port:

|  |
| --- |
| // Create Vertex  if (type.equals("node")) {  (...)  // Fetch ID Value  id = tmp.getNodeValue();  // Need unique valid ID  if (id != null && !ids.keySet().contains(id)) {  // Create Vertex with label  *DefaultGraphCell v = new DefaultGraphCell(label);*  // Add One Floating Port  *v.add(new DefaultPort());*  // Add ID, Vertex pair to Hashtable  ids.put(id, v);  // Add Default Attributes  attributes.put(v, createDefaultAttributes());  // Update New Cell List  newCells.add(v);  }  } |

Otherwise, an edge is created and its source and target IDs are looked up in the ids map. The first child, which is the port of the returned vertex is used to establish the connection in the connection set. (The source and target ID are analogous, so only the source ID is shown below.)

|  |
| --- |
| // Create Edge  else if (type.equals("edge")) {  // Create Edge with label  *DefaultEdge edge = new DefaultEdge(label);*  // Fetch Source ID Value  id = tmp.getNodeValue();  // Find Source Port  if (id != null) {  // Fetch Vertex for Source ID  DefaultGraphCell v = (DefaultGraphCell) ids.get(id);  if (v != null)  // Connect to Source Port  *cs.connect(edge, v.getChildAt(0), true);*  }  (...)  // Update New Cell List  newCells.add(edge);  } |

When the innermost loop exits, the new cells are inserted into the model, together with the connection set and attributes. Note that the toArray method is used to create the first argument, which is an array of Objects.

|  |
| --- |
| // Insert the cells (View stores attributes)  model.insert(newCells.toArray(), cs, null, attributes);  } |

A helper method creates the attributes for the vertex, which are stored in a map, from keys to values. This map is then stored in another map, with the vertex as a key. The latter is used as a parameter for the insert method.

|  |
| --- |
| // Create Attributes for a Black Line Border  public static Map createDefaultAttributes() {  // Create an AttributeMap  Map map = GraphConstants.createMap();  // Black Line Border (Border-Attribute must be null)  GraphConstants.setBorderColor(map, Color.black);  // Return the Map  return map;  } |

### Layout

Here is the layout method. Using this method, the vertices of the graph are arranged in a circle. The method has two loops: First, all cells are filtered for vertices, and the maximum width or height is stored. (The order of the cells in the circle is equal to the order of insertion.)

|  |
| --- |
| public static void layout(JGraph graph) {  // Fetch All Cell Views  CellView[] views = graph.getView().getRoots();  // List to Store the Vertices  List vertices = new ArrayList();  // Maximum width or height  int max = 0;  // Loop through all views  for (int i = 0; i < views.length; i++) {  // Add Vertex to List  if (views[i] instanceof VertexView) {  *vertices.add(views[i]);*  // Fetch Bounds  Rectangle b = views[i].getBounds();  // Update Maximum  if (bounds != null)  max = Math.max(Math.max(b.width, b.height), max);  }  } |

The number of vertices and the maximum width or height is then used to compute the minimum radius of the circle, and the angle step size. The step size is equal to the circle, divided by the number of vertices.

|  |
| --- |
| // Compute Radius  int r = (int) Math.max(vertices.size()\*max/Math.PI, 100);  // Compute Radial Step  double phi = 2\*Math.PI/vertices.size(); |

With the radius and step size at hand, the second loop is entered. In this loop, the position of the vertices is changed (without using the history).

|  |
| --- |
| // Arrange vertices in a circle  for (int i = 0; i < vertices.size(); i++) {  // Cast the Object to a CellView  CellView view = (CellView) vertices.get(i);  // Fetch the Bounds  Rectangle bounds = view.getBounds();  // Update the Location  if (bounds != null)  *bounds.setLocation(r+(int) (r\*Math.sin(i\*phi)),*  *r+(int) (r\*Math.cos(i\*phi)));*  }  } |

### Write

Then the graph is written to an SVG file using the write method. To implement this method, the Batik library is used. The graph is painted onto a custom Graphics2D, which is able to stream the graphics out as SVG.

|  |
| --- |
| // Stream out to the Writer as SVG  public static void write(JGraph graph, Writer out) (...) {  // Get a DOMImplementation  DOMImplementation dom = GenericDOMImplementation.getDOMImplementation();  // Create an instance of org.w3c.dom.Document  Document doc = dom.createDocument(null, "svg", null);  // Create an instance of the SVG Generator  SVGGraphics2D svgGenerator = new SVGGraphics2D(doc);  // Render into the SVG Graphics2D Implementation  *graph.paint(svgGenerator);*  // Use CSS style attribute  boolean useCSS = true;  // Finally, stream out SVG to the Writer  svgGenerator.stream(out, useCSS);  } |

## Simple Diagram Editor (GraphEd)

This example provides a simple diagram editor, which has a popup menu, a command history, zoom, grouping, layering, and clipboard support. The connections between cells may be established in a special connect mode, which allows dragging the ports of a cell.

The connect mode requires a special marquee handler, which is also used to implement the popup menu. The history support is implemented by using the GraphUndoManager class, which extends Swing's UndoManager. A custom graph overrides the edge view's default bindings for adding and removing points (the right mouse button that is used for the popup menu, and shift-click is used instead). Additionally, a custom model that does *not* allow self-references is implemented.

The editor object extends JPanel, which can be inserted into a frame. It provides the inner classes MyGraph, MyModel and MyMarqueeHandler, which extend JGraph, DefaultGraphModel andBasicMarqueeHandler respectively. A GraphSelectionListener is used to update the toolbar buttons, and a KeyListener is responsible to handle the *delete* keystroke, however, the implementation of these interfaces is not shown below.

Here are the class definition and variable declarations:

|  |
| --- |
| public class Editor extends JPanel (...) {  // JGraph object  protected JGraph graph;  // Undo Manager  protected GraphUndoManager undoManager; |

The main method constructs a frame with an editor and displays it:

|  |
| --- |
| // Usage: java -jar graphed.jar  public static void main(String[] args) {  // Construct Frame  JFrame frame = new JFrame("GraphEd");  (...)  // Add an Editor Panel  *frame.getContentPane().add(new Editor());*  (...)  // Set Default Size  frame.setSize(520, 390);  // Show Frame  frame.show();  } |

### Constructor

The constructor of the editor panel creates the MyGraph object and initializes it with an instance of MyModel. Then, the GraphUndoManager is constructed, and the undoableEditHappenedis overridden to enable or disable the undo and redo action using the updateHistoryButtons method (which is not shown).

|  |
| --- |
| // Construct an Editor Panel  public Editor() {  // Use Border Layout  setLayout(new BorderLayout());  // Construct the Graph object  *graph = new MyGraph(new MyModel());*  // Custom Graph Undo Manager  undoManager = new GraphUndoManager() {  // Extend Superclass  public void undoableEditHappened(UndoableEditEvent e) {  // First Invoke Superclass  super.undoableEditHappened(e);  // Update Undo/Redo Actions  updateHistoryButtons();  }  }; |

The undo manager is then registered with the graph, together with the selection and key listener.

|  |
| --- |
| // Register UndoManager with the Model  *graph.getModel().addUndoableEditListener(undoManager);*  // Register the Selection Listener for Toolbar Update  graph.getSelectionModel().addGraphSelectionListener(this);  // Listen for Delete Keystroke when the Graph has Focus  graph.addKeyListener(this); |

Finally, the panel is constructed out of the toolbar and the graph object:

|  |
| --- |
| // Add a ToolBar  add(createToolBar(), BorderLayout.NORTH);  // Add the Graph as Center Component  *add(new JScrollPane(graph), BorderLayout.CENTER);*  } |

The Editor object provides a set of methods to change the graph, namely, an insert, connect, group, ungroup, toFront, toBack, undo and redo method, which are explained below. The cut, copy and paste actions are implemented in the JGraph core API, and are explained at the end.

### Insert

The insert method is used to add a vertex at a specific point. The method creates the vertex, and adds a floating port.

|  |
| --- |
| // Insert a new Vertex at point  public void insert(Point pt) {  // Construct Vertex with no Label  *DefaultGraphCell vertex = new DefaultGraphCell();*  // Add one Floating Port  vertex.add(new DefaultPort()); |

Then the specified point is applied to the grid, and used to construct the bounds of the vertex, which are subsequently stored in the vertices' attributes, which are created and accessed using the GraphConstants class:

|  |
| --- |
| // Snap the Point to the Grid  pt = graph.snap(new Point(pt));  // Default Size for the new Vertex  Dimension size = new Dimension(25,25);  // Create a Map that holds the attributes for the Vertex  Map attr = GraphConstants.createMap();  // Add a Bounds Attribute to the Map  *GraphConstants.setBounds(attr, new Rectangle(pt, size));* |

To make the layering visible, the vertices' attributes are initialized with a black border, and a white background:

|  |
| --- |
| // Add a Border Color Attribute to the Map  GraphConstants.setBorderColor(attr, Color.black);  // Add a White Background  GraphConstants.setBackground(attr, Color.white);  // Make Vertex Opaque  GraphConstants.setOpaque(attr, true); |

Then, the new vertex and its attributes are inserted into the model. To associate the vertex with its attributes, an additional map, from the cell to its attributes is used:

|  |
| --- |
| // Construct the argument for the Insert (Nested Map)  Hashtable nest = new Hashtable();  // Associate the Vertex with its Attributes  nest.put(vertex, attr);  // Insert wants an Object-Array  Object[] arg = new Object[]{vertex};  // Insert the Vertex and its Attributes  *graph.getModel().insert(arg, null, null, nest);*  } |

### Connect

The connect method may be used to insert an edge between the specified source and target port:

|  |
| --- |
| // Insert a new Edge between source and target  public void connect(Port source, Port target) {  // Construct Edge with no label  *DefaultEdge edge = new DefaultEdge(edge);*  // Use Edge to Connect Source and Target  ConnectionSet cs = new ConnectionSet(edge, source, target); |

The new edge should have an arrow at the end. Again, the attributes are created using the GraphConstants class. The attributes are then associated with the edge using a nested map, which is passed to the model's insert method:

|  |
| --- |
| // Create a Map that holds the attributes for the edge  Map attr = GraphConstants.createMap();  // Add a Line End Attribute  GraphConstants.setLineEnd(attr, GraphConstants.SIMPLE);  // Construct a Map from cells to Maps (for insert)  Hashtable nest = new Hashtable();  // Associate the Edge with its Attributes  nest.put(edge, attr);  // Insert wants an Object-Array  Object[] arg = new Object[]{edge};  // Insert the Edge and its Attributes  *graph.getModel().insert(arg, cs, null, nest);*  } |

### Group

The group method is used to compose a new group out of an array of cells (typically the selected cells). Because the cells to be grouped are already contained in the model, a parent map must be used to change the cells' existing parents, and the cells' layering-order must be retrieved using the order method of the GraphView object.

|  |
| --- |
| // Create a Group that Contains the Cells  public void group(Object[] cells) {  // Order Cells by View Layering  cells = graph.getView().order(cells);  // If Any Cells in View  if (cells != null && cells.length > 0) {  // Create Group Cell  *DefaultGraphCell group = new DefaultGraphCell();* |

In the following, the entries of the parent map are created by associating the existing cells (children) with the new cell (parent). The parent map and the new cell are then passed to the model's insert method to create the group:

|  |
| --- |
| // Create Change Information ParentMap  map = new ParentMap();  // Insert Child Parent Entries  for (int i = 0; i < cells.length; i++)  map.addEntry(cells[i], group);  // Insert wants an Object-Array  Object[] arg = new Object[]{group};  // Insert into model  *graph.getModel().insert(arg, null, map, null);*  }  } |

### Ungroup

The inverse of the above is the ungroup method, which is used to replace groups by their children. Since the model makes no distinction between ports and children, we use the cell's *view* to identify a group. While the getChildCount for vertices with ports returns the number of ports, the corresponding VertexView's isLeaf method only returns true if at least one child is not a port, which is the definition of a group:

|  |
| --- |
| // Determines if a Cell is a Group  public boolean isGroup(Object cell) {  // Map the Cell to its View  CellView view = graph.getView().getMapping(cell, false);  if (view != null)  return !view.isLeaf();  return false;  } |

Using the above method, the ungroup method is able to identify the groups in the array, and store these groups in a list for later removal. Additionally, the group's children are added to a list that makes up the future selection.

|  |
| --- |
| // Ungroup the Groups in Cells and Select the Children  public void ungroup(Object[] cells) {  // Shortcut to the model  GraphModel m = graph.getModel();  // If any Cells  if (cells != null && cells.length > 0) {  // List that Holds the Groups  ArrayList groups = new ArrayList();  // List that Holds the Children  ArrayList children = new ArrayList();  // Loop Cells  for (int i = 0; i < cells.length; i++) {  // If Cell is a Group  *if (isGroup(cells[i])) {*  // Add to List of Groups  groups.add(cells[i]);  // Loop Children of Cell  for (int j = 0; j < m.getChildCount(cells[i]); j++)  // Get Child from Model  children.add(m.getChild(cells[i], j));  }  }  // Remove Groups from Model (Without Children)  *m.remove(groups.toArray());*  // Select Children  graph.setSelectionCells(children.toArray());  }  } |

### To Front / To Back

The following methods use the graph view's functionality to change the layering of cells, or get redirected to the model based on isOrdered.

|  |
| --- |
| // Brings the Specified Cells to Front  public void toFront(Object[] c) {  if (c != null && c.length > 0)  graph.getView().toFront(graph.getView().getMapping(c));  }    // Sends the Specified Cells to Back  public void toBack(Object[] c) {  if (c != null && c.length > 0)  graph.getView().toBack(graph.getView().getMapping(c));  } |

### Undo / Redo

The following methods use the undo manager's functionality to undo or redo a change to the model or the graph view:

|  |
| --- |
| // Undo the last Change to the Model or the View  public void undo() {  try {  undoManager.undo(graph.getView());  } catch (Exception ex) {  System.err.println(ex);  } finally {  updateHistoryButtons();  }  }    // Redo the last Change to the Model or the View  public void redo() {  try {  undoManager.redo(graph.getView());  } catch (Exception ex) {  System.err.println(ex);  } finally {  updateHistoryButtons();  }  } |

### Graph Component

Let's look at the implementation of the MyGraph class, which is GraphEd's main UI component. A custom graph is necessary to override the default implementation's use of the right mouse button to change an edge's points, and also to change flags, and to use the custom model and marquee handler, which are printed below. The MyGraph class is implemented as an inner class:

|  |
| --- |
| // Custom Graph  public class MyGraph extends JGraph {  // Construct the Graph using the Model as its Data Source  public MyGraph(GraphModel model) {  super(model);  // Use a Custom Marquee Handler  *setMarqueeHandler(new MyMarqueeHandler());*  // Tell the Graph to Select new Cells upon Insertion  setSelectNewCells(true);  // Make Ports Visible by Default  setPortsVisible(true);  // Use the Grid (but don't make it Visible)  setGridEnabled(true);  // Set the Grid Size to 6 Pixel  setGridSize(6);  // Set the Snap Size to 1 Pixel  setSnapSize(1);  } |

To override the right mouse button trigger with a shift trigger, an indirection is used. By overriding the createEdgeView method, we can define our own EdgeView class, which in turn overrides the isAddPointEvent method, and isRemovePointEvent method to check the desired trigger:

|  |
| --- |
| // Override Superclass Method to Return Custom EdgeView  protected EdgeView createEdgeView(Edge e, CellMapper c) {  // Return Custom EdgeView  return new EdgeView(e, this, c) {  // Points are Added using Shift-Click  public boolean isAddPointEvent(MouseEvent event) {  return event.isShiftDown();  }  // Points are Removed using Shift-Click  public boolean isRemovePointEvent(MouseEvent event) {  return event.isShiftDown();  }  };  }  } |

### Graph Model

The custom model extends DefaultGraphModel, and overrides its acceptsSource and acceptsTarget methods. The methods prevent self-references, that is, if the specified port is equal to the existing source or target port, then they return false:

|  |
| --- |
| // A Custom Model that does not allow Self-References  public class MyModel extends DefaultGraphModel {  // Source only Valid if not Equal to Target  public boolean acceptsSource(Object edge, Object port) {  *return (((Edge) edge).getTarget() != port);*  }  // Target only Valid if not Equal to Source  public boolean acceptsTarget(Object edge, Object port) {  *return (((Edge) edge).getSource() != port);*  }  } |

### Marquee Handler

The idea of the marquee handler is to act as a "high-level" mouse handler, with additional painting capabilites. Here is the inner class definition:

|  |
| --- |
| // Connect Vertices and Display Popup Menus  public class MyMarqueeHandler extends BasicMarqueeHandler {  // Holds the Start and the Current Point  protected Point start, current;    // Holds the First and the Current Port  protected PortView port, firstPort; |

The isForceMarqueeEvent method is used to fetch the subsequent mousePressed, mouseDragged and mouseReleased events. Thus, the marquee handler may be used to gain control over the mouse. The argument to the method is the event that triggered the call, namely the mousePressed event. (The graph's portsVisible flag is used to toggle the connect mode.)

|  |
| --- |
| // Gain Control (for PopupMenu and ConnectMode)  public boolean isForceMarqueeEvent(MouseEvent e) {  // Wants to Display the PopupMenu  if (SwingUtilities.isRightMouseButton(e))  return true;  // Find and Remember Port  port = getSourcePortAt(e.getPoint());  // If Port Found and in ConnectMode (=Ports Visible)  *if (port != null && graph.isPortsVisible())*  return true;  // Else Call Superclass  return super.isForceMarqueeEvent(e);  } |

The mousePressed method is used to display the popup menu, or to initiate the connection establishment, if the global port variable has been set.

|  |
| --- |
| // Display PopupMenu or Remember Location and Port  public void mousePressed(final MouseEvent e) {  // If Right Mouse Button  if (SwingUtilities.isRightMouseButton(e)) {  // Scale From Screen to Model  Point l = graph.fromScreen(e.getPoint());  // Find Cell in Model Coordinates  Object c = graph.getFirstCellForLocation(l.x,l.y);  // Create PopupMenu for the Cell  JPopupMenu menu = createPopupMenu(e.getPoint(), c);  // Display PopupMenu  menu.show(graph, e.getX(), e.getY());  // Else if in ConnectMode and Remembered Port is Valid  } else if (port != null && !e.isConsumed() &&  graph.isPortsVisible()) {  // Remember Start Location  start = graph.toScreen(port.getLocation(null));  // Remember First Port  firstPort = port;  // Consume Event  e.consume();  } else  // Call Superclass  super.mousePressed(e);  } |

The mouseDragged method is messaged repeatedly, before the mouseReleased method is invoked. The method is used to provide the live-preview, that is, to draw a line between the source and target port for visual feedback:

|  |
| --- |
| // Find Port under Mouse and Repaint Connector  public void mouseDragged(MouseEvent e) {  // If remembered Start Point is Valid  if (start != null && !e.isConsumed()) {  // Fetch Graphics from Graph  Graphics g = graph.getGraphics();  // Xor-Paint the old Connector (Hide old Connector)  paintConnector(Color.black, graph.getBackground(), g);  // Reset Remembered Port  port = getTargetPortAt(e.getPoint());  // If Port was found then Point to Port Location  if (port != null)  current = graph.toScreen(port.getLocation(null));  // Else If no Port found Point to Mouse Location  else  current = graph.snap(e.getPoint());  // Xor-Paint the new Connector  paintConnector(graph.getBackground(), Color.black, g);  // Consume Event  e.consume();  }  // Call Superclass  super.mouseDragged(e);  } |

The following method is called when the mouse button is released. If a valid source and target port exist, the connection is established using the editor's connect method:

|  |
| --- |
| // Establish the Connection  public void mouseReleased(MouseEvent e) {  // If Valid Event, Current and First Port  if (e != null && !e.isConsumed() && port != null &&  firstPort != null && firstPort != port) {  // Fetch the Underlying Source Port  Port source = (Port) firstPort.getCell();  // Fetch the Underlying Target Port  Port target = (Port) port.getCell();  // Then Establish Connection  connect(source, target);  // Consume Event  e.consume();  } else {  // Else Repaint the Graph  graph.repaint();  }  // Reset Global Vars  firstPort = port = null;  start = current = null;  // Call Superclass  super.mouseReleased(e);  } |

The marquee handler also implements the mouseMoved method, which is messaged independently of the others, to change the mouse pointer when over a port:

|  |
| --- |
| // Show Special Cursor if Over Port  public void mouseMoved(MouseEvent e) {  // Check Mode and Find Port  if (e != null && getSourcePortAt(e.getPoint()) != null &&  !e.isConsumed() && graph.isPortsVisible()) {  // Set Cusor on Graph (Automatically Reset)  graph.setCursor(new Cursor(Cursor.HAND\_CURSOR));  // Consume Event  e.consume();  }  // Call Superclass  super.mouseReleased(e);  } |

Here are the helper methods used by the custom marquee handler. The first is simply used to retrieve the port at a specified position. (The method is named getSourcePortAtbecause another method must be used to retrieve the target port.)

|  |
| --- |
| // Returns the Port at the specified Position  public PortView getSourcePortAt(Point point) {  // Scale from Screen to Model  Point tmp = graph.fromScreen(new Point(point));  // Find a Port View in Model Coordinates and Remember  return graph.getPortViewAt(tmp.x, tmp.y);  } |

The getTargetPortAt checks if there is a cell under the mouse pointer, and if one is found, it returns its "default" port (first port).

|  |
| --- |
| // Find a Cell and Return its Defaulr Port  protected PortView getTargetPortAt(Point p) {  // Find Cell at point (No scaling needed here)  Object cell = graph.getFirstCellForLocation(p.x, p.y);  // Shortcut Variable  GraphModel model = graph.getModel();  // Loop Children to find first PortView  for (int i = 0; i < model.getChildCount(cell); i++) {  // Get Child from Model  Object tmp = graph.getModel().getChild(cell, i);  // Map Cell to View  tmp = graph.getView().getMapping(tmp, false);  // If is Port View and not equal to First Port  if (tmp instanceof PortView && tmp != firstPort)  // Return as PortView  return (PortView) tmp;  }  // No Port View found  return getSourcePortAt(point);  } |

The paintConnector method displays a preview of the edge to be inserted. (The paintPort method is not shown.)

|  |
| --- |
| // Use Xor-Mode on Graphics to Paint Connector  void paintConnector(Color fg, Color bg, Graphics g) {  // Set Foreground  g.setColor(fg);  // Set Xor-Mode Color  g.setXORMode(bg);  // Highlight the Current Port  paintPort(graph.getGraphics());  // If Valid First Port, Start and Current Point  if (firstPort != null && start != null &&  current != null) {  // Then Draw A Line From Start to Current Point  *g.drawLine(start.x, start.y, current.x, current.y);*  }  } // End of Editor.MyMarqueeHandler |

The rest of the Editor class implements the methods to create the popup menu (not shown), and the toolbar. These methods mostly deal with the creation of action objects, but the copy, paste, and cut actions are exceptions:

|  |
| --- |
| // Creates a Toolbar  public JToolBar createToolBar() {  JToolBar toolbar = new JToolBar();  toolbar.setFloatable(false);  (...)  // Copy  *action = graph.getTransferHandler().getCopyAction();*  url = getClass().getClassLoader().getResource("copy.gif");  action.putValue(Action.SMALL\_ICON, new ImageIcon(url));  *toolbar.add(copy = new EventRedirector(action));*  (...)  } |

Because the source of an event that is executed from the toolbar is the JToolBar instance, and the copy, cut and paste actions assume a graph as the source, an indirection must be used to change the source to point to the graph:

|  |
| --- |
| // This will change the source of the actionevent to graph.  protected class EventRedirector extends AbstractAction {  protected Action action;  // Construct the "Wrapper" Action  public EventRedirector(Action a) {  super("", (ImageIcon) a.getValue(Action.SMALL\_ICON));  this.action = a;  }    // Redirect the Actionevent to the Wrapped Action  public void actionPerformed(ActionEvent e) {  e = new ActionEvent(*graph*, e.getID(),  e.getActionCommand(), e.getModifiers());  *action.actionPerformed(e);*  }  } |

|  |  |  |
| --- | --- | --- |
| [<<< Previous](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/t1.html) | [Home](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/t1.html) | [Next >>>](http://dev.cs.uni-magdeburg.de/java/jgraph/tutorial/x669.html) |
| The JGraph Tutorial |  | References |