

JGoTM for Swing

Graphical Object Editor Classes

User Guide

This guide provides information on using the **com.nwoods.jgo** package Version 5.2, for use with the Java 2 platform (JDK/SDK-SE 1.2 or later).

September 2006

**Northwoods Software Corporation
142 Main St.
Nashua, NH 03060**

<http://www.nwoods.com/go>

<mailto:JGo@nwoods.com>

JGo User Guide

Copyright © 1999-2006 Northwoods Software Corporation

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise without the prior written permission of the publisher.

Northwoods Software Corporation makes no representations that the use of its products in the manner described in this publication will not infringe on existing or future patent rights, nor do the descriptions contained in this publication imply the granting of licenses to make, use, or sell equipment or software in accordance with the description.

Possession, use, or copying of the software described in this publication is authorized only pursuant to a valid written license from Northwoods or an authorized sublicensor.

Neither Northwoods Software Corporation nor its employees are responsible for any errors that may appear in this publication. The information in this publication is subject to change without notice.

The following are trademarks of Northwoods Software Corporation: Northwoods, JGo, GO++, GoDiagram, GoLayout, GoInstruments, Sanscript, Flowgram, the Northwoods logo, and Fully Visual Programming.

The following are third-party trademarks:

Java and all Java-based marks are trademarks or registered trademarks of Sun Microsystems, Inc. in the U.S. and other countries.

Borland and JBuilder are registered trademarks of Borland Software Corporation in the U.S. and other countries.

All other trademarks and registered trademarks are property of their respective holders.

CONTENTS

Preface	v
1. Introduction.....	1
2. JGo Concepts	3
Design Philosophy	3
Documents.....	3
Views	4
Graphical Objects	5
Selection	6
Grouping	6
Graphs	7
A Minimal Applet/Application	7
3. JGoDocument and JGoObject Details	9
JGoDocument	9
JGoObject	13
JGoDrawable	16
JGoText	17
JGoImage	18
JGoArea.....	19
JGoPort.....	21
JGoLink.....	24
4. JGoView Details.....	29
Display	30
Events	31
User Editing	37
5. Nodes.....	41
JGoBasicNode	42
JGoIconicNode	44
JGoTextNode.....	45
JGoSubGraph	46
SimpleNode	47
GeneralNode	48
MultiPortNode	48
MultiTextNode.....	49
ListArea.....	50
RecordNode.....	50
Comment	51
General Concepts When Defining Nodes.....	51
6. Undo and Redo	53
UndoableEdit and JGoDocumentChangedEdit	53
JGoUndoManager, CompoundEdits and Transactions	59
Defining Menu Commands	61
7. Performance Hints	63
8. JGo Support for XML and SVG	64
SVG Support using Batik and SVGGoView.....	64
XML and SVG Support Using JAXP and the JGo SVG Package	65
Custom XML Support Using JAXP	68
9. Building a Sample Application Using JGo Beans.....	71
Register the JGo Beans with the Development Environment	72
Visually Construct the User Interface	72
Add Event Listeners.....	74

JGo User Guide

Customize the Palette.....	78
Add Clipboard Support	81
Add Undo/Redo Support.....	82
Add Auto-layout Support.....	85
Add XML/SVG Serialization Support	86

PREFACE

Purpose of this guide

This guide provides an overview of JGo. All of the classes are part of the **com.nwoods.jgo** or **com.nwoods.jgo.svg** packages.

For more detailed information about the classes and members in the JGo package, see the JGo Class Reference Guide, a set of HTML pages generated by JavaDoc.

This guide also uses as examples some of the code provided in the **com.nwoods.jgo.examples** package or its subpackages.

Who should use this guide

This guide is intended for application programmers using the Java2 platform to build Swing applications. Familiarity with Java and Swing is assumed.

Structure of this guide

This guide is organized as follows:

- **Introduction**— summarizes the capabilities of the JGo software.
- **JGo Concepts**— describes the overall design of the JGo classes.
- **JGoDocument and JGoObject Details**— describes the details of the JGo model classes.
- **JGoView Details**— describes the details of screen updating and input handling.
- **Nodes**—describe the many kinds of nodes that come with JGo.
- **Undo and Redo**— describes what you need to do to support undo and redo
- **Performance Hints**— suggestions for how to avoid some performance bottlenecks
- **Building a Sample Application**— walks through building an application

Other Guides

Another package, **com.nwoods.jgo.layout**, provides sophisticated layout algorithms for nodes in graphs. The layout package is licensed and documented separately.

Yet another package, **com.nwoods.jgo.instruments**, provides JGo objects that act as dials/gauges/meters. The instruments package is licensed and documented separately.

1. INTRODUCTION

The JGo package, **com.nwoods.jgo**, is a set of classes built on the Java 2 platform, including Java2D and Swing. JGo makes it easy to deliver user interfaces that allow users to see and manipulate diagrams of two-dimensional graphical objects arranged in a scrollable, zoomable window. You can use JGo to build stand-alone Swing applications, Swing applets, and thin-client web applications using servlets.

JGo provides a variety of basic graphical objects such as rectangles, ellipses, polygons, text, images, and lines. You can group objects together to form more complex objects. You can customize their appearances and behaviors by setting properties and overriding methods.

A JGo view is a component that displays a JGo document. It supports mouse-based object manipulation, including selecting, resizing, moving and copying using drag-and-drop. A JGo view also supports in-place editing, printing, and grids.

A JGo document implements a model that supports manipulation of objects. Adding an object to the document makes it visible in the document's views. You can organize objects in layers. JGo provides support for composing and manipulating graphs (node & arc diagrams), where nodes have ports that are connected by links.

The JGo library is flexible and extensible. Many predefined example node classes make it easy to build many kinds of diagrams. You can easily customize most objects for application-specific purposes by setting properties or by subclassing. You can add completely new graphical objects to the existing framework.

JGo is written entirely in portable Java source code. It only depends on the standard Java2 packages and does not explicitly call any native functions.

JGo contains very few user-visible text strings. You can easily localize your JGo application by setting one class's array of strings.

The **com.nwoods.jgo.svg** package is included with JGo and provides classes to enable serialization of JGoDocuments to and from XML documents. The XML document format used is an extension of the SVG (Scalable Vector Graphics) XML document type. JGoDocuments can be saved and faithfully restored in this format and can also be viewed with standard SVG viewers (however SVG files generated by other tools cannot necessarily be faithfully read by JGo). Refer to the "Serialization" section of "JGoDocument" in the "JGoDocument and JGoObject Details" chapter for more information on this topic, and to Chapter 8.

2. JGo CONCEPTS

This guide assumes you are already familiar with the Java2 platform, Java2D and Swing. JGo builds directly on this framework, so understanding them is a prerequisite for understanding JGo. All JGo classes follow the convention of using "JGo" in their name (e.g., **JGoView**); any other class names in this document are Java classes (e.g., **java.awt.Graphics2D**) or part of the samples.

Although property names in Java technically should start with a lower-case letter, in this document they are often capitalized for clarity.

Design Philosophy

JGo has been designed to be high performance, easy to use and flexible enough to meet a large array of requirements. We have extended Swing in as straight forward a method as possible, because that minimizes the learning curve and also minimizes compatibility problems with new releases of Java.

While JGo does not provide every last feature you may need, we strive to provide hooks in the right places so that you can write all your code in your derived objects without modifying JGo. We consider it a misfeature of JGo if there is something you can't achieve without modifying the JGo sources.

Documents

JGo uses the model-view-controller architecture. **JGoDocument** serves as the model, i.e. a container providing the abstract representation of the things the user may see in a view.

Documents provide the runtime storage for the displayable objects. A document is the object that contains the list of layers of graphical objects to be displayed in one or more views. When you want to have a graphical object appear to the user, you create it, make sure it has a reasonable size and position and any other properties you care about, and then add it to a document's layer.

Class **JGoDocument** extends **Object**, so a document and its objects do not depend on the existence of a window. Each document has a list of **JGoLayer** instances. **JGoLayer** implements **JGoObjectCollection**, i.e. a list of **JGoObjects**. The graphical objects that a document layer contains are instances of subclasses of **JGoObject**. For convenience, **JGoDocument** also implements **JGoObjectCollection**, treating all of the objects in all of its layers as one long list of objects.

Each document has a number of properties that affect its appearance and behavior. These include properties such as paper color and whether the user can modify the document.

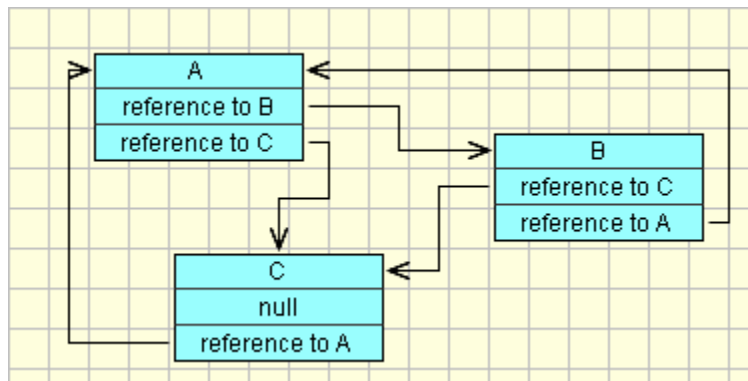
JGo User Guide

When a **JGoDocument** is modified, it fires **JGoDocumentEvents** by notifying all registered **JGoDocumentListeners** of all changes to the document, to its layers, or to any of its objects.

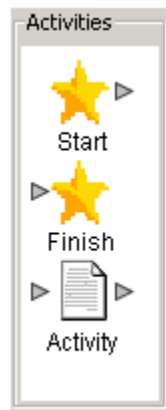
Views

Views provide a window on the graphical objects stored in a document. A view defines how the user sees the objects and interacts with them. Each view is a **JGoDocumentListener** of the document being viewed so that it can keep its window up-to-date with all of the objects in the document.

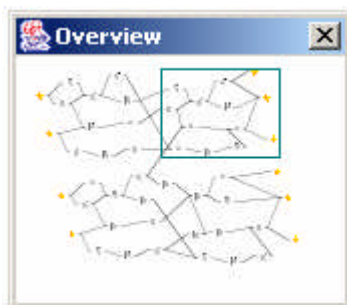
Class **JGoView** extends **javax.swing.JComponent** and provides the basic functionality of displaying objects in layers, with an optional background grid or image:



If you need a read-only collection of objects that are laid out in a grid to be selected and dragged by the user into another view, use the **JGoPalette** class:



If you need a reduced-scale view of a different view that allows the user to pan that other view, use the **JGoOverview** class:



JGoView, in conjunction with the graphical objects, provides a default user interaction style that is consistent with standard usability guidelines for selection, moving, resizing and other user interactions. However, user interactions defined by **JGoView** and the objects themselves are highly customizable. Some of this customization is achieved via properties on the objects. Much customization can be accomplished by registering **JGoDocumentListeners** on the document and **JGoViewListeners** on the view. More powerful customization can be achieved through the subclassing of the **JGoView** and **JGoObjects** and overriding member functions.

JGoView fires **JGoViewEvents** for interactive user actions such as selecting or clicking on objects or in the background, and for programmatic changes to view properties.

Graphical Objects

All graphical objects in JGo are derived from **JGoObject**, which in turn is derived from Java's **Object**.

JGoObject defines the basics of a graphical object: a bounding rectangle, mechanisms for controlling the size and location of the object, and the common properties **Visible**, **Selectable**, **Resizable** and **Draggable**. **JGoObject** defines its own **paint** method that defines the appearance of that object. Thus the full power of Java2D is available for drawing your custom objects, if you really need it.

The simplest way to think about **JGoObject** is that it is a rectangular area that knows how to draw itself into a view. In fact, in many cases of creating a new simple object, the only code you need to write is the paint method. For example, here is the only significant code in **JGoRectangle**:

```
public void paint(Graphics2D g, JGoView view)
{
    Rectangle rect = getBoundingRect();
    drawRect(g, rect.x, rect.y, rect.width, rect.height);
}
```

There are three kinds of primitive **JGoObjects**:

- Drawable shapes, such as rectangles, ellipses and strokes. Each **JGoDrawable** instance can have a pen for drawing the outline of the shape and a brush for painting (filling) the inside of the shape.
- Text, in various fonts, sizes and colors. **JGoText** objects also support multiple lines, wrapping and in-place editing.

- Images, for various kinds of images such as JPEGs and GIFs. **JGoImage** objects can get their image information from files or URLs, either on disk, across the net, or in a JAR file.

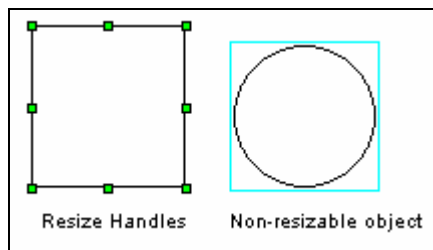
JGoObject provides numerous hooks so that custom derived objects can provide exactly the desired look and feel. More information is provided in the next chapter.

Selection

The **JGoSelection** class is used by a view to maintain a separate list of the objects selected. Each view has its own selection. In addition, the selection class notifies objects of gaining and losing selection events, and has support (in conjunction with **JGoObject**) for the appearance of a selected object. Objects can define their own selection appearance or use the default provided by the **JGoObject** and **JGoSelection** classes. Normally **JGoSelection** uses a class called **JGoHandle**, which is derived from **JGoRectangle**, to make selection handles appear on the screen.

The following illustration shows the default selection appearance for resizable and non-resizable objects. Also note that, by default, the primary selection color is green and the secondary selection color is blue-green.

JGoView has some useful methods for manipulating the selection: adding objects to the selection and moving or deleting the selected objects.



Selection appearance for resizable and non-resizable objects

Grouping

JGo provides two ways of making groups of objects: areas and layers. Areas provide a way of making a single “object” out of other objects. Layers are a way of viewing multiple collections of objects in a document.

JGoArea is a **JGoObject** that contains other objects within its bounding rectangle. **JGoArea** in many ways acts like a document, because objects can be added or removed from it, and the stacking (painting) order within the area can be changed. **JGoArea** can contain any object that is derived from **JGoObject**. Since **JGoArea** is itself derived from **JGoObject**, areas can contain other areas, to any depth.

An object that does not have a parent area is called a “top-level” object. The objects in an area are often called its children. Removing a group from a document effectively causes the group’s children to disappear also.

Several very commonly used areas that implement “nodes” with “ports” are provided in the JGo package, such as **JGoBasicNode** and **JGoTextNode**. Many other area and node classes are provided in the examples subdirectory.

JGoLayer is a collection of top-level **JGoObjects** held by a **JGoDocument**. Layers are just a way to split up the list of objects owned by a document. Each document starts off with one layer. You can add and remove layers from a document. You can also change the order of the layers in a document, thereby making potentially many objects all over the document appear in front of or behind other collections of objects. Furthermore you can affect the visibility and transparency of all of those objects in a layer all at once. Unlike **JGoArea**, **JGoLayer** does not extend **JGoObject**, so one cannot have layers within layers. Each **JGoObject** can belong to at most one layer at a time.

Graphs

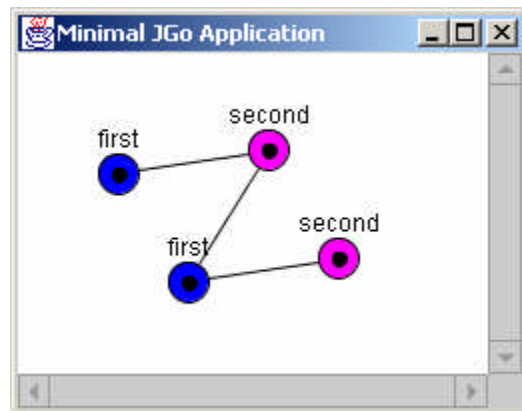
One of the principal uses of JGo is to make it easy to build applications where users can see and manipulate graphs of nodes connected by links. JGo provides this functionality with the **JGoNode**, **JGoPort** and **JGoLink** classes. Nodes contain one or more ports. Links connect two ports.

The example classes provide some pre-built implementations of useful nodes. You can extend them easily if you need to customize their appearance or behavior.

The sample apps provide some pre-built implementations of graphical browsers and editors.

A Minimal Applet/Application

A very basic use of JGo is provided in the examples directory, **MinimalApp.java**.



All of the interesting JGo code is in the **init** method:

```
private JGoView myView;

// constructor
public MinimalApp() {
    // create a JGo view (a JComponent) and add to the applet
    myView = new JGoView();
```

JGo User Guide

```
        getContentPane().add(myView);
    }

    public void init() { // only here do we anything JGo specific
        // add JGoObjects to the document, not to the view
        JGoDocument doc = myView.getDocument();

        // create two nodes for fun...
        JGoBasicNode node1 = new JGoBasicNode("first");
        // specify position
        node1.setLocation(new Point(100, 100));
        // specify color
        node1.setBrush(JGoBrush.makeStockBrush(Color.blue));
        // add to the document
        doc.addObjectAtTail(node1);

        JGoBasicNode node2 = new JGoBasicNode("second");
        node2.setLocation(new Point(200, 100));
        node2.setBrush(JGoBrush.makeStockBrush(Color.magenta));
        doc.addObjectAtTail(node2);
    }
```

This minimal applet or application just puts up two **JGoBasicNodes** of different colors. The user can link them together, select the nodes and/or link(s), move them around, or copy them. JGo provides all of this functionality automatically.

For more detailed information on building apps, see chapter 9.

3. JGoDOCUMENT AND JGoOBJECT DETAILS

JGoDocument

JGoDocument represents a group of **JGoObjects** that can be displayed by a **JGoView**. **JGoDocument** represents the model in the model-view-controller architecture; **JGoView** plays the role of the view and as a default controller.

A document should be thought of as an ordered list of objects. The objects are drawn in sequential order, so objects at the beginning of the list appear "behind" objects that are at the end. You can add, remove, and iterate over the document's objects by using the document's implementation of the **JGoObjectCollection** interface

In addition to all of the objects held by the document, the document has its own notion of the background color, called the paper color. This is independent of the **JGoView** (i.e. **JComponent**) background color, which affects the view's border and which is used as the background when the document's paper color is null.

JGoDocument also supports undo and redo by cooperating with a **JGoUndoManager** that listens for and records changes to the document.

Layers

The ordered list of objects is actually partitioned into sublists by the use of **JGoLayer** instances. Although you normally think of a document as owning the objects in it, actually a document directly owns an ordered list of layers. Each layer in turn owns an ordered list of objects.

When you have created an instance of a **JGoObject**, you'll want to add it to a document by calling the **JGoDocument** **addObjectAtTail** or **addObjectAtHead** methods. The former method places the new object in front of all other document objects; the latter places it behind all others. If you are making use of layers, once you have decided the layer in which to put the new object, you can call the same methods on the layer instead. You can "move" an object from one layer to another by calling these methods too.

Initially a document has one layer. Each document has a notion of the default layer in which a view may create new objects if the exact layer is not explicitly specified.

Each document also has a property referring to the layer that by default holds links. It is moderately common to create a separate layer for links, to display all links either in front of the default layer or behind the default layer that presumably holds all of the nodes.

JGo User Guide

Initially, since there is only one document layer, the value of **getLinksLayer()** is the same as the value of **getDefaultLayer()**.

You may find it convenient when adding **JGoObjects** programmatically to your document to call the **JGoDocument.add** method. This method automatically adds the object at the tail end of the links layer if the object is a link; otherwise it adds it at the tail end of the default layer.

The **removeAll** method removes all **JGoObjects** from the document without removing any layers. The **deleteContents** method removes all objects and layers. But you might find it easier to just throw away the old **JGoDocument** and create and initialize a new one instead of trying to re-initialize the old one.

Document Coordinates and Size

The **JGoObjects** held in the document each have a size and position. The coordinate system used by the document comes from the default coordinate system for components, i.e. positive coordinates increase rightwards and downwards and each unit corresponds to a pixel. **JGoViews** have a coordinate system that may be translated and scaled from that of the document, so as to support panning and zooming.

The document's size is automatically expanded to encompass all of its objects. Normally a document has all of its objects at positive coordinates (i.e., the lower right quadrant). However, if there are objects with negative coordinates, the **documentTopLeft** property will indicate the actual “origin” with a negative X and/or Y value. This property combined with the **documentSize** property gives the full extent of all of the objects in the document. It is possible to set either of these properties, but by default they will automatically get re-set as existing objects are moved or resized or as new objects are added. However the document's size does not automatically contract as objects are moved or removed. **JGoDocument.updateDocumentSize** is called when an object's bounding rectangle is changed; you can override that method to implement your own policies regarding document size and top-left position.

Events

The document keeps track of all registered **JGoDocumentListeners**. **JGoView** is a predefined implementer of **JGoDocumentListener**. It needs to notice when document objects change so that it can update the visible rendering of those objects. You can register your own listeners to notice changes to the document or its objects. The **fireUpdate** method actually does the notification of all document listeners.

JGoDocumentEvent is the class that represents an event for a document; it extends **java.util.EventObject**. Besides remembering which document the event occurred for, it also remembers the kind of event and the previous state, if appropriate. The kinds of events include such things as a **JGoObject** being INSERTED, REMOVED, or CHANGED. For some kinds of events, there is additional information that further describes the event. In particular, the CHANGED event has an object specific sub-hint describing the exact kind of change and a previous value.

In the Demo1 example we only show arrowheads at the “to” port of links connected to the ports of **JGoBasicNodes**. One way of achieving that effect is to notice when links

JGoDocument and JGoObject Details

get created in the document, and making sure they have the appropriate arrowheads or other desired characteristics.

```
. . .
myDoc.addDocumentListener(new JGoDocumentListener() {
    public void documentChanged(JGoDocumentEvent e) {
        processDocChange(e);
    }
});

public void processDocChange(JGoDocumentEvent e)
{
    switch(e.getHint()) {
        case JGoDocumentEvent.INSERTED:
            if (e.getJGoObject() instanceof JGoLink) {
                // make sure each link has an arrowhead at its front,
                // if the "to" port is a basic node port
                JGoLink link = (JGoLink)e.getJGoObject();
                JGoPort port = link.getToPort();
                if (port != null &&
                    port.getParent() instanceof JGoBasicNode) {
                    // only have an arrowhead at the "to" end
                    link.setArrowHeads(false, true);
                }
                // if the link connects ports on different classes,
                // highlight it in red
                if (link.getFromPort() != null &&
                    port.getClass() != link.getFromPort().getClass()) {
                    link.setHighlight(JGoPen.make(JGoPen.SOLID,
                        link.getPen().getWidth()+4, Color.red));
                }
            }
            break;
    }
}
```

Instead of adding a document listener, an alternate way to get notification of events from a **JGoDocument** is to create a subclass of **JGoView** and override the **documentChanged** method, since each view is also a document listener.

Note that this code will be called whenever any object is programmatically added to the document, not just when users interactively draw a new link. If you just want this to happen when the user interactively draws a new link, implement a **JGoViewListener** to look for the **JGoViewEvent.LINK_CREATED** case, or override **JGoView.newLink**.

Copying

You can add a copy of a collection of objects to a document by calling **copyFromCollection**. The way objects are copied is controlled by the **JGoObject.copyObject** methods of all the copied objects and by the **JGoCopyEnvironment**. The **JGoCopyEnvironment** also holds the results of the copying.

If you just want to add a copy of a single object to a document, call **addCopy**.

In order to work out-of-the-box with the clipboard and with drag-and-drop, **JGoDocument** implements **Transferable** and provides a default **DataFlavor**.

Serialization and Persistence

JGoDocument and **JGoObject** implement **Serializable**. Serialization is used by the default drag-and-drop mechanism and by copy-and-paste.

Serialized objects will not be compatible with future JGo releases. Thus you should not use the standard Java serialization mechanism to implement long-term storage of documents or collections of objects.

For long-term persistence, you may wish to read and write into your own existing database or file format. In this case, your **JGoDocument** subclass, and perhaps your **JGoObject** subclasses, will be responsible for transforming the real information into a network of **JGoObjects**. Any user driven or programmatic changes to these objects must then be transformed back into the database's representation of the information. The document will also have to act as a listener for any independent changes to the underlying database, if that is supported.

Alternatively, if the format of your serialized data is not predetermined, serialization to and from XML can be easily provided by the **com.nwoods.jgo.svg** package. Details are provided in Chapter 8.

You may find it helpful to turn on the **JGoDocument** property **MaintainsPartID**, which automatically assigns a unique integer value to each **JGoIdentifiablePart** that is added to the document. **JGoNode**, **JGoLink**, and **JGoPort** all implement **JGoIdentifiablePart**, so you could use the **PartID** of ports to be able to identify which ports each link connects, and to identify other references to nodes or links. The example class, **Comment**, also implements **JGoIdentifiablePart**, so that it is easy to store and update such comment objects.

The **com.nwoods.jgo.examples.flower** sample application provides an example of serializing to and from several different file formats, including raw serialized objects (not recommended for long-term storage), a custom XML format created specifically for the flower example, and extended SVG XML as provided by the **com.nwoods.jgo.svg** package.

Navigating Documents

Frequently you will want to iterate over all of the objects in a document or in a layer, perhaps just to find and operate on a certain subset of the objects. Because **JGoDocument** and **JGoLayer** and **JGoArea** implement **JGoObjectSimpleCollection**, you can easily walk the list of objects as follows:

```

JGoListPosition pos = myDoc.getFirstObjectPos();
while (pos != null) {
    JGoObject obj = myDoc.getObjectAtPos(pos);
    pos = myDoc.getNextObjectPosAtTop(pos);
    if (obj instanceof MyNode) {
        MyNode n = (MyNode)obj;
        // do something with MyNode n
    }
}

```

Of course you can check for other classes too, such as **JGoLink**. And you can replace “myDoc” with a layer or an area to iterate over the objects in those collections.

Controlling Link Creation By the User

JGoDocument has a property, **ValidCycle**, that controls whether users are allowed to draw links between nodes that might cause cycles or loops in the graph, or that would violate a tree-structure, seen abstractly. This property is observed by the **validLink** predicate of **JGoPort**. More information is provided in the description of ports on page 22.

JGoObject

JGoObject is the superclass of all objects that can be contained in a **JGoDocument** (**JGoLayer**) or a **JGoView** and that can be displayed in a view.

JGoObjects are efficient; if most AWT components are considered heavyweight, and if most Swing components are considered lightweight, **JGoObjects** are flyweight.

Bounding Rectangle and Location

Each object has a size and a position, in document coordinates. There are many methods for getting and setting the bounding rectangle for the object, or for just the **Left**, **Top**, **Width**, or **Height** properties. All ultimately go through the basic **getBoundingRect** and **setBoundingRect** methods.

Although normally one can think of the location of an object being the same as the top-left corner, that may not be natural for some objects. Thus each object has its own notion of **Location**; by default this is the same as the top-left. For example, **JGoText** overrides **getLocation** and **setLocation** to use the text alignment in determining the natural position of the object.

Note that when the location is not the same as the top-left position, the order of setting the **Location** and the **Size** of an object may matter. This is because setting the size of an object is likely to cause the location to change. You may want to use (and override if appropriate) the **setSizeKeepingLocation** method.

There are a number of convenience methods for dealing with the standard nine spots of an object (corners, sides, and center), and for repositioning two objects so that their particular user-specified spots coincide. See **getSpotLocation**, **setSpotLocation**, and **setSpotLocationOffset**. The standard spots are:

- **Center**

JGo User Guide

- **TopLeft**
- **TopCenter**
- **TopRight**
- **RightCenter**
- **BottomRight**
- **BottomCenter**
- **BottomLeft**
- **LeftCenter**

The spot locations are also used to identify the standard handles. There are also **NoSpot** and **NoHandle** values for situations where is no particular spot or handle.

Ownership

Most **JGoObjects** should either belong directly to a **JGoDocument** (actually a **JGoLayer**) as a top-level document object, or to a **JGoArea** that belongs to a document/layer. In either case **getDocument** returns this document and **getLayer** returns the layer within the document; for children of areas, **getParent** will return that **JGoArea** instead of **null**.

Occasionally some objects will properly belong to a **JGoView** instead of to a **JGoDocument**, because they really represent part of the "view" of the document and not of the document itself. Predefined cases include selection handles and the in-place text editor. The size and position of view objects are in document coordinates.

Whenever any object is added or removed from a document or a view, the appropriate **INSERTED** or **REMOVED** event is fired for all listeners.

Events

As you define your own subclasses, you can define customized default behaviors for responding to various events. **JGoObjects** do not have their own listeners and events because it is assumed that most of the objects of a certain class in a graph want to behave the same way. This is unlike the situation where one expects to add controls to a dialog without subclassing and yet have radically different behaviors for each one.

The standard "event" handling methods are:

- **geometryChange** - the object has changed size and/or position
- **geometryChangeChild** - a child object has changed size and/or position
- **handleResize** - the user is reshaping this object interactively
- **ownerChange** - the object has just been added or removed from a document or view
- **paint** - render this object through a **Graphics2D** if the object **isVisible()**; if you override this method to draw beyond the bounding rectangle, be sure to override **expandRectByPenWidth**

JGoDocument and JGoObject Details

- **doMouseClicked** - the user just clicked on this object
- **doMouseDblClick** - the user just double-clicked on this object
- **doUncapturedMouseMove** - the user just passed the mouse over this object without any mouse down
- **gainedSelection** - this object just got added to some view's selection
- **lostSelection** - this object just got removed from some view's selection
- **redirectSelection** - this object is about to be selected; maybe select something else
- **getToolTipText** - return a string to display in a tool tip

Properties

In addition to the bounding rectangle and the location, each object has a number of boolean properties:

- **Visible** – can this object be seen in a view
- **Selectable** – can the user select this object in a view
- **Draggable** – can the user move this object in a view
- **Resizable** – can the user reshape this object in a view
- **4ResizeHandles** – does this object only have corner selection handles
- **AutoRescale** – whether resizing a parent area will resize this object
- **DragsNode** – whether this selected object, when moved, should move the top-level parent object instead
- **PickableBackground** – for areas, whether the area (if selectable) should become selected if picking in the area but not on a child object
- **BoundingRectInvalid** – whether to call **computeBoundingRect** to get a new bounding rectangle
- **Initializing** – normally used by areas to indicate that the area is being constructed or copied, to avoid repeated expensive calculations, such as in **JGoArea.layoutChildren**
- **SuspendUpdates** – should the object temporarily skip notifying listeners
- **SkipsUndoManager** – like **SuspendUpdates**, but instructs only the undo manager to stop recording information from events for this object; other updates, such as for the view's display, proceed normally

Remember that properties such as **Selectable** and **Resizable** just control the standard built-in behavior that JGo views allow the user to do interactively using the mouse. You can always select or resize objects programmatically, regardless of these property values, by explicitly calling methods such as **JGoView.getSelection().extendSelection(o)** and **JGoObject.setSize(w,h)**.

JGo User Guide

When an object's property changes, a **CHANGED** document event is sent to all document listeners. As you define subclasses with additional properties or other state, you will need to remember to make such notifications. It is easiest to call the **JGoObject.update** method after the object's state changes, because it can take care of the notification details for you.

A **CHANGED** document event has a flags/hint value which is useful in identifying the kind of change that occurred. For example, a call to **JGoObject.setVisible** will result in a call to **JGoObject.update** with a hint of **ChangedVisible**. This additional discrimination is important for optimizing update behavior and supporting undo and redo.

If you want to make a copy of a single object without adding it to a **JGoDocument**, you can call the **copy** method. (If you do want to add a copy to a document, call **JGoDocument.addCopy**.)

As you add fields to your subclasses, you will want to make sure the fields are copied appropriately when the object is copied by overriding **copyObject**. Because **JGoObject** implements **Serializable**, you will need to make sure all and only those fields that are needed to be serialized are not declared **transient**, or you will need to define **writeObject** and **readObject**.

In addition, if you want to support undo and redo, you will need to make sure your subclass also handles new properties correctly in the **copyNewValueForRedo** and **changeValue** methods. See the chapter about undo and redo for more details.

JGoDrawable

The principal subclasses of **JGoObject** include **JGoDrawable**, **JGoText**, **JGoImage**, and **JGoArea**. These are discussed in the following sections.

Drawable shapes include both closed and filled two-dimensional objects and unfilled (linear) objects such as **JGoStrokes**. Strokes are multi-segmented straight or curved lines. Strokes can also have arrowheads.

Most drawables, though, are things like rectangles, rounded rectangles, ellipses and polygons.

Each **JGoDrawable** has a pen (**JGoPen**) and a brush (**JGoBrush**) to specify how to draw the outline of the drawable shape and how to paint the inside of the shape. There are a few predefined pen and brush values that are static values in the **JGoPen** and **JGoBrush** classes. Pens and brushes are considered immutable objects, so you can freely share them among multiple **JGoObjects**.

You can also construct your own **JGoPen** and **JGoBrush** values. This is useful when you want a thicker pen, a dotted pen, a partially transparent brush, or a gradient or a textured paint brush.

If you define your own drawable class, you will probably need to consider overriding at least the following methods: **paint**, **expandRectByPenWidth**, **isPointInObj**, and **getNearestIntersectionPoint**.

JGoText

The **JGoText** class displays text strings. There are many properties that help determine the appearance and behavior of a **JGoText** object:

- **Text** – the string to be displayed
- **FaceName** – the string name of the font family to be used; the default is "SansSerif"
- **FontSize** – the point size specifying the height and width of the characters; the default is 12
- **Alignment** – how each line of text is aligned within the whole text object; the default is **ALIGN_LEFT**; this also determines the **Location** for the object
- **TextColor** – the color for the characters; the default is **Color.black**
- **BkColor** – the color for the background behind the text; the default is **Color.white**
- **Transparent** – if true, the background color (**BkColor**) is not painted; otherwise the whole text object is filled with the background color
- **Bold** – whether the text is in a bold style
- **Italic** – whether the text is in an italicized style
- **Underline** – whether the text is underlined
- **StrikeThrough** – whether the text appears “crossed out”
- **Multiline** – whether embedded newline characters force a line break in the display of the text string, or whether line wrapping takes place
- **Clipping** – whether the text drawing is clipped to the bounds of the text object; for speed this defaults to false
- **AutoResize** – whether the size of the text object is automatically adjusted as the text string is changed
- **2DScale** – whether the user can resize a text object horizontally as well as vertically
- **Editable** – whether double-clicking or clicking on the text object causes the view to bring up a text field or text area editor for the user to edit-in-place.
- **EditOnSingleClick** – by default if the text object is **Editable**, double clicking starts editing-in-place; when this property is true, only a single click is needed.
- **SelectBackground** – whether selecting a text object causes the background to be displayed (**Transparent** set to false) instead of getting selection handle(s) as most objects normally do
- **Wrapping** – whether to automatically insert line breaks even when there is no newline character embedded in the string; **Multiline** must also be true for wrapping to take place, and any embedded newline characters are ignored

JGo User Guide

- **WrappingWidth** – when **Wrapping** is true, specifies the width at which text will be wrapped to the next line, in document coordinates

When a **JGoText** object is constructed, the **FaceName** and **FontSize** properties default to the values of the static properties **JGoText.getDefaultFontFaceName()** and **JGoText.getDefaultFontSize()**. By default, text objects are not **Resizable**. They support only single lines of text and do not wrap or clip.

The **AutoResize** property, which defaults to true, causes the text string to be remeasured each time the string value is changed and the bounding rectangle to be updated accordingly. The **Location** (as determined by the **Alignment**) will stay the same, but the width and height will match the dimensions of that text string, in the given font and style. If you set **AutoResize** to false or if you explicitly change the **Size** of the text object, you run the risk of painting beyond the bounds of the text object, which will result in improper updates of the view. In this case it is wise to set the **Clipping** property to be true, to make sure that the text is not drawn beyond the bounds of the object. The **Clipping** property defaults to false for performance reasons.

The **SelectBackground** property determines how a selected text object appears by controlling the transparency of the text's background instead of adding selection handles.

For improved performance the **paint** method calls the **paintGreek** method to allow it to decide on simpler renditions of the text at small scales. The standard implementation uses the static **JGoText.getPaintNothingScale()** and **JGoText.getPaintGreekScale()** properties to decide if the text should be painted at all or if it should just be drawn as a single line.

Users can edit text in-place. If the **Editable** property is true, then a double click (or a single click if **EditOnSingleClick** is true) on the text object will invoke **doBeginEdit** to create and display a **JTextField** or a **JTextArea** component. The **Multiline** and **Wrapping** properties determine the behavior of the Enter key. When **Multiline** is true and **Wrapping** is false, the text editing component accepts the Enter key as inserting a newline; when **Multiline** is false, the Enter key calls **doEndEdit** to finish editing, resulting in a modified **Text** string value. In either case the Escape key calls **doEndEdit** without changing the string value.

JGoImage

The **JGoImage** class displays images, including GIF files and JPEG files. The images can be kept as separate files or stored as a resource, referred to by a disk pathname or by a URL.

Properties:

- **Image** – the underlying **Image** object
- **Filename** – the file pathname from which the image is loaded; null if the **URL** is used instead
- **URL** – the URL from which the image is loaded; null if the **Filename** is used instead

- **TransparentColor** – the **Color** that is to be drawn instead of a transparent background for the image; this defaults to null, so that images will show transparent backgrounds naturally
- **NaturalSize** – the unstretched dimensions of the **Image**, independent of the dimensions of the **JGoImage** object that you may have assigned

Initially a **JGoImage** instance will have no **Image** to display, and thus will appear empty. You can assign an image by calling one of the three overloaded methods named **loadImage**, taking **Image**, **String**, or **URL** as a first argument. For convenience in dealing with “relative” paths using URLs, the **loadImage(String, boolean)** method first checks to see if the static **getDefaultBase** method returns non-null. If so, it returns the result of calling **loadImage(new URL(getDefaultBase(), filename), wait)** instead.

You should override **JGoImage.loadImage** if you have alternate means of getting an **Image** in memory and you depend on serialization. Setting the **Image** property by calling **loadImage(Image, boolean)** works, but the **Image** is not serialized. When a **JGoImage** is serialized and deserialized, it depends on the **loadImage** method to reproduce the **Image**. If **loadImage** fails because there is no loadable **Filename** or **URL** value and you have not provided an alternative means of getting an **Image**, no image will show in the view for the deserialized object.

The **JGoImage** class keeps a cache of **Image** values for the files and URLs it loads from. This saves time and space when the same image file is used by more than one object. However, if the external file may have changed, you can clear the cached **Image** by calling one of the static **resetImage** or **resetAllImages** methods.

JGoArea

JGoArea implements the concept of a “group” of objects that can be manipulated together. An area, like a document, contains a list of **JGoObjects**. These objects must **not** also be contained directly by the document, or by other areas—i.e., objects cannot be shared.

Just as with document and layers, you can add objects in front of or behind the existing objects in the area, by using the **JGoArea.addObjectAtTail**, **addObjectAtHead**, **insertObjectAfter** and **insertObjectBefore** methods. If the object is already part of the area, it will make sure the object is at the appropriate Z-order position. Otherwise the object must not belong to any other area or be a top-level object in a layer. You can “reparent” objects (between different areas or to/from top-level) within the same layer by using the **addCollection** method.

JGoArea is a subclass of **JGoObject**, which means that areas can contain other areas. This is the Composite pattern. Using this mechanism, an object hierarchy can be created.

An area does not really have its own independent bounding rectangle. Instead the bounding rectangle is really the bounding rectangle for all of the children. In fact **getBoundingRect** is not meaningful when there are no objects in an area.

JGoDocument and **JGoView** treat areas specially--they search in them when you use **pickObject** or **getNextObject**.

JGo User Guide

Often you will want to have this whole "group" of objects be selected instead of any part. Make the parts (child objects) not selectable; when clicking on a child object, that child object will not be selected, but the parent area will be if that parent area **isSelectable()**.

Clicking on any background within the area, i.e. not on a child object, does not select the area. If you call **setPickableBackground(true)** (and the area is selectable and the children are not), then clicking anywhere within the area's bounding rectangle will select the area, including at any points where there are no visible parts of the area.

Note: in version 5.1 the **PickableBackground** property supercedes the **GrabChildSelection** property, which is now deprecated, even though the meanings of the two properties differ from each other. Now the **Selectable** property really means the same thing for **JGoAreas** as it does for other kinds of **JGoObjects**, and you don't need to turn on/off the **GrabChildSelection** property to make a **JGoArea** selectable/unselectable. Setting the **PickableBackground** property to true covers the same purpose as setting **Selectable** to true for **JGoAreas** in versions before 5.1. For more information regarding compatibility and upgrading your code, see the release notes.

It is commonplace for each top-level area to be selectable, but not to have pickable backgrounds, and for all of the children to be not selectable. As you construct your area by adding **JGoObjects**, you will typically need to remember to call **setSelectable(false)** on each child object.

If a **JGoArea** object is removed from the document, all of its children are also removed.

The coordinates for objects within the area are kept in document coordinates; they are not relative to the area.

JGoArea Management

Any object's position and/or size is changed by a call to **setBoundingRect**. Such a change will also invoke the **geometryChange** method and all document listeners with a CHANGED document event and a **ChangedGeometry** hint. Remember that these hooks get called *after* the size and position have been changed. Override **setBoundingRect** itself if you want to prevent certain geometry changes from happening at all, but do so very cautiously.

The default behavior implemented by **JGoArea.geometryChange** moves all the children and resizes them by the same scale that the whole area is resized. This is performed by the standard implementation of the **rescaleChildren** method. For areas that are "nodes", (an icon, a label and some ports), the built-in resize is probably not appropriate, especially when text strings are included. You can set a child's **AutoRescale** property to false to prevent **rescaleChildren** from changing the size of the object. In fact, **JGoText** objects have a default value of false for **AutoRescale**.

Particularly since not all children are scaled proportionately, you will need to specify the size and position of the children explicitly. It is fairly common for each subclass of **JGoArea** to override **layoutChildren** in order to re-position and perhaps re-size the area's children to maintain a certain appearance.

The standard **JGoArea.moveChildren** method is called by **geometryChange** to move all of the area's children when neither the width nor the height of the whole area has been changed.

When an area's child object's size or position is changed by a call to **setBoundingRect**, the parent area is notified by a call to **geometryChangeChild**. This allows the area to adjust its notion of its position and size. The default behavior for **JGoArea.geometryChangeChild** is to call **layoutChildren**; the argument will be the child object that was moved or resized.

But remember that the change was instigated by a change to a child, and not to the area as a whole. Be careful to avoid infinite adjustment loops or differing behavior depending on the order of changes. This might happen if you look at the bounding rectangle of the whole area after changing the bounding rectangle of a child. Since the bounding rectangle of the whole area is the union of the bounding rectangles of its children, moving a child may change the bounding rectangle of the area, which may throw your child layout out of whack. Instead, try to position and size all of the children relative to a particular child that the user would think of as being the primary object. For example, for a **JGoTextNode**, that primary object is the **JGoText**, so **JGoTextNode.layoutChildren** method sizes and positions all other child objects relative to that text object.

You need to consider whether users trying to move or copy a child object should instead move or copy the parent. Because most children are not **Selectable** this is not an issue—the parent object will be selected and moved. But if they are selectable and you want them to move independently, you will want to set the **JGoObject.DragsNode** property to false. (This happens automatically when you add a child to **JGoSubGraph**.) Even then if your override of **JGoArea.layoutChildren** automatically repositions each of the children to the “right” place when the area is resized, that will keep the child in its original location! If you want to allow children to be selected and able to be moved on their own, you should make sure that the **geometryChange** and **layoutChildren** methods do not control their positioning.

On the other hand, if you want the children of a group to be individually selectable but you do not want the user to move them independently, you should set the **JGoObject.DragsNode** property to true for each of these children. This will let a user's drag of a selected child drag the whole top-level area.

If the object's shape isn't like the bounding rectangle, you may need to override **JGoObject.isPointInObj** to improve picking

JGoPort

JGoPort acts as a connection point for **JGoLink** objects. Each port has a collection of **JGoLinks** that are attached to the port.

Appearance

By default a **JGoPort** appears as an ellipse, but it can use any other **JGoObject** to control its appearance. The predefined styles are:

- **StyleHidden** – nothing is drawn
- **StyleObject** – another object (a “Port Object”) provides the representation
- **StyleEllipse** – uses an ellipse (or circle)
- **StyleTriangle** – uses a triangle “pointing” appropriately

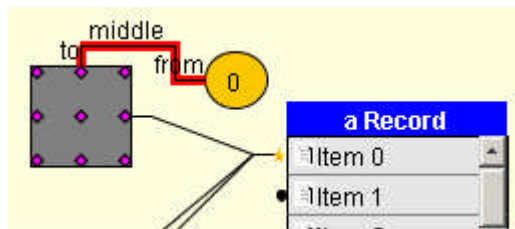
JGo User Guide

- **StyleRectangle** – uses a rectangle (or square)
- **StyleDiamond** – uses a four-sided polygon with the vertices at the midpoints of the bounding rectangle's edges

The following example node has two ports of **StyleEllipse** and two ports of **StyleTriangle**:



In the following screenshot there is a node with nine ports of **StyleDiamond**, with a link to a **JGoBasicNode** whose port is **StyleHidden**, and another link to the first item in a **RecordNode** whose port is of **StyleObject**; the object providing the appearance for the port is a **JGoImage** displaying a star. The second item has an elliptical port, filled with a black brush.



JGoPort is a subclass of **JGoDrawable**, so you can easily control the appearance of the non-hidden, non-Object ports by calling **setPen** and/or **setBrush**.

Ports can also share many Port Objects. Your application can, for example, pre-allocate several different **JGoImage** instances corresponding to the kinds of states you want to display to the user. As each port changes state, you just need to call **setPortObject** with the appropriate image. Because potentially many ports will share these Port Objects, they must not be part of any document (or area or view). Before each Port Object is painted, its bounding rectangle will be set to the bounding rectangle of the port.

Linking Ports

For your application, some ports may be valid sources for links, some may be valid destinations, and some may be both or neither. It may be that some particular pairs of ports cannot have a valid new link between them. For example, you may want to avoid having two different links connecting the same two ports. **JGoView** calls the **isValidSource**, **isValidDestination** and **validLink** methods to allow the particular port classes the ability to control whether the user can draw a link starting at a given port and ending at one.

The standard definition of **JGoPort.validLink** also calls **isValidSelfNode** and **isValidDuplicateLinks** to decide if it is OK to create a link with both ports part of the same node and to decide if it is OK to create more than one link in one direction between the same pair of ports. It also checks the **JGoDocument.getValidCycle** property to

possibly call one of the **JGoDocument.makeDirectedCycle** or **makeUndirectedCycle** methods.

The **ValidSource**, **ValidDestination**, **ValidSelfNode**, and **ValidDuplicateLinks** properties are all settable. The **ValidCycle** property of **JGoDocument** is settable too, of course.

When the mouse is over a port where the user can start drawing a link, the cursor changes to a Hand cursor.

Because ports have a size, the exact point at which a link should terminate may want to depend on the dimensions of the port. Furthermore it is common for there to be different points depending on whether the link is coming in or going out of the port or where the port is located relative to the rest of the node. This notion is supported by the **FromSpot** and **ToSpot** properties, which remember the object spots that links connected to this port should end at. The **getLinkPoint** method is responsible for calculating this **Point**; the default behavior depends on the **FromSpot** and **ToSpot** values.

Override the **getLinkPoint** method to produce more sophisticated link appearances. Usually if the link direction for the port is on one side, the link point will be on the same side to avoid overlapping the link with the visual appearance of the port. Note that the link point need not be in the bounding rectangle of the port, although if it is too far away it might be confusing or disconcerting for the user.

If you expect the link point to vary dynamically, you may wish to specify **NoSpot** as the value for one or both of the **FromSpot** and **ToSpot** properties. Override the **getLinkPointFromPoint** method and calculate the link point. The **X** and **Y** arguments specify approximately where the link is coming from or going to.

Links that are connected to a port may be constrained to come into the port or come out of the port from certain directions. The direction is calculated by **getLinkDir**. The standard directions correspond to the spot locations. If the spot is **Center** or **NoSpot** you will want to override this method to return the desired direction.

Navigating Links

Each port has a collection of links that are attached to the port. The port does not own any of the links; normally the document owns all links. From a port you can iterate over all the links to get to all the ports connected by those links. For example, here is the code in the Family Tree example where the document is positioning all the “children” **PersonNodes** for a particular mother/father pair. All of the children are linked to the mother/father marriage at a “marriage port”, here held in a variable named **mp**.

```
// now look at each child
JGoListPosition childpos = mp.getFirstLinkPos();
while (childpos != null) {
    JGoLink childlink = mp.getLinkAtPos(childpos);
    childpos = mp.getNextLinkPos(childpos);

    JGoPort childp = childlink.getOtherPort(mp);
    PersonNode childnode = (PersonNode)childp.getParent();
}
```

JGo User Guide

```
layoutTree(childnode, childrect);  
}
```

This code iterates over the links at the `mp` port. It gets the port at the other end of the link. Then it gets the **PersonNode** for that other port by getting the port's parent container and assuming it is of the correct class. Finally it actually calls a function with that node representing the child.

Another method that can be useful for finding directly connected links or nodes is the **JGoNode.findAll** method.

JGoLink

JGoLink is a **JGoStroke** that connects two different **JGoPorts**. Normally you create a link by allocating a new **JGoLink** specifying both the “from” and “to” ports, and adding it to a document. Delete a link by calling the **unlink** method, which automatically removes the link from the document as well as disconnecting the link from its ports.

Link Path

The default link stroke will consist of three straight segments (four points in the stroke). The end segments, at the ports, will be relatively short. The middle segment will be just a straight line connecting the two short segments at the ports. There is no short end segment if the corresponding port does not have a link port spot (i.e., the value is **NoSpot**). **JGoPort.getLinkDir** gives the direction. You can control the length of this short end segment by setting the **JGoPort**'s **EndSegmentLength** property.

If both ports have link port spots that are **NoSpot**, then the default link stroke consists of only a single segment (two points in the stroke), unless it is **Cubic**, when it will have four points and the **Curviness** property governs the path of the curve.

If you set the **Orthogonal** property to **true**, the default link stroke will have five segments instead of three, and all segments will be either horizontal or vertical. You can also have the corners of orthogonal links be rounded by setting the **RoundedCorners** property. When **isRoundedCorners()** is true, **getCurviness()** controls the diameter of the corner curve.

If the position of one or both of its **JGoPorts** changes, the **JGoLink** redraws itself to connect the new positions. When either port changes it calls the **portChange** method, which by default just calls **calculateStroke**. For complete control over the points in the stroke, override the **calculateStroke** method to define the points used by the link's stroke. However, you can set the **AdjustingStyle** and **AvoidsNodes** properties to control how the intermediate points are plotted, excluding the end segments, if any.

When the link's from and to ports are the same port, the default **calculateStroke** method produces a little “loop” connecting the port with itself.

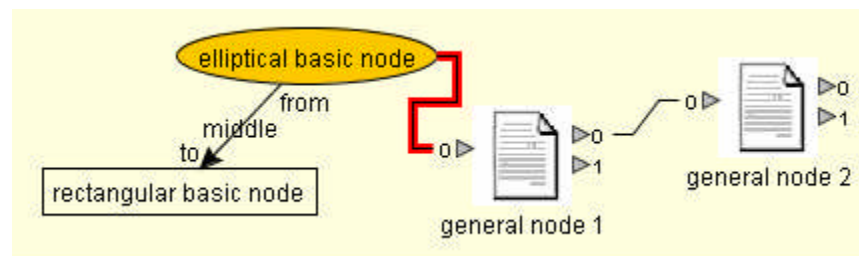
If you programmatically create a link between two nodes, the initial route for the link may cross over some nodes until the **calculateStroke** method has a chance to adjust. You may want to call **calculateStroke** explicitly on such newly created links.

Appearance and Behavior

Many attributes of links can easily be customized through the properties and methods of **JGoStroke** and **JGoDrawable**, such as:

- line color, thickness, and style (**JGoDrawable.setPen**)
- arrowheads (**JGoStroke** arrowheads)
- number, location, and size of line segments (**JGoStroke** points and **calculateStroke**; for curved links, **JGoStroke.setCubic**)
- number, style, and behavior of resize handles (pick points and **handleResize**)
- highlighting (**JGoStroke.setHighlight**)
- jumping-over of orthogonal segments (**JGoLink.setJumpsOver**)
- curviness of cubic non-orthogonal links and rounded corners of orthogonal links

The following screen shot displays a **JGoLabeledLink** that has an arrowhead at the “to” end and only a single segment connecting two styles of **JGoBasicNode**, an **Orthogonal** link to a **GeneralNode**, and a standard three-segment link between the two general nodes. The orthogonal link also has a thick red highlight pen in addition to the standard black pen of width 1.

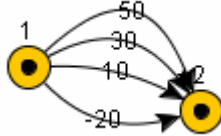


When a **JGoLink** is cubic, and the link connects two **JGoPorts** that have **NoSpot**, **calculateStroke** automatically places the points of the stroke in a curve. The **Curviness** property controls how far off a straight line the control points are for the cubic stroke. A positive value produces a clockwise curve; a negative value produces a counter-clockwise curve, and a value of zero produces a straight line.

The following picture shows two cubic links connecting two **JGoBasicNodes**. Each link has the default **Curviness**.



You can distinguish between multiple links between the same ports by assigning different values to **Curviness**:



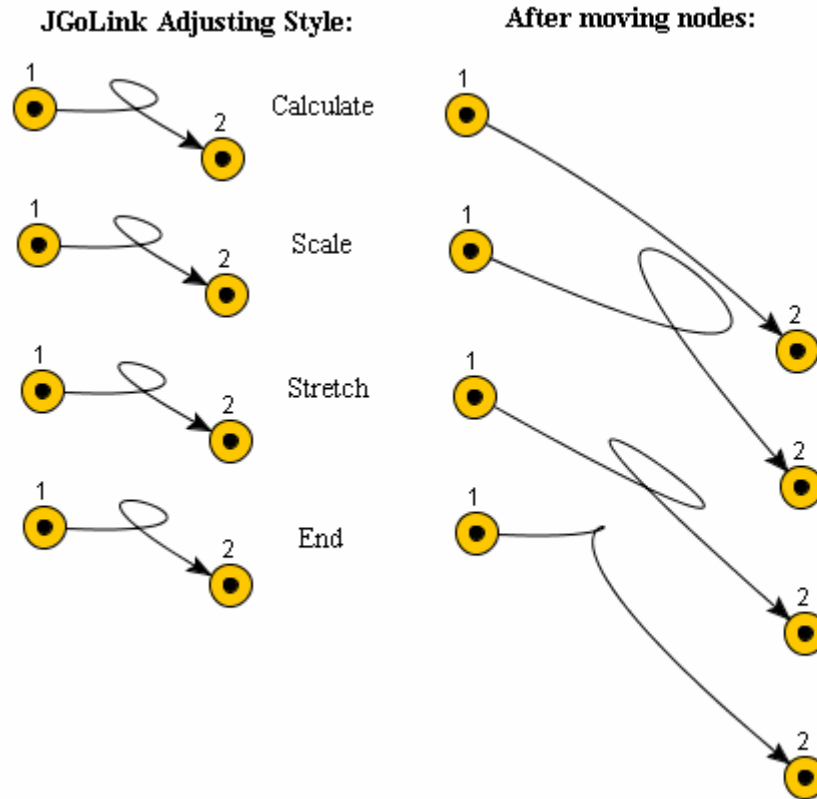
As mentioned earlier, the **JGoLink.calculateStroke** method is called when one of the ports is moved and the stroke points need to be replotted in order to maintain the appearance of a connection between the link's two ports. The standard behavior, depending on various properties of the link and of the ports, was also described above.

However, a link may very well have a non-standard path, either because you have programmatically modified the points, or because the user has "resized" some of the link points by hand. The behavior of **calculateStroke** depends on the link's **AdjustingStyle** property. The default value is **JGoLink.AdjustingStyleCalculate**, which produces the standard link path. Thus if the user has manually moved some of the intermediate points of a link, and then moves one of the connected nodes, the manual customization is lost because the link's path is restored to the standard route.

However, you can set the **AdjustingStyle** property to other values:

- **JGoLink.AdjustingStyleScale** – scale and rotate the intermediate points of the link so as to maintain the appearance of the shape of the link, but at a different size and angle to accommodate the new relative positions of the ports
- **JGoLink.AdjustingStyleStretch** – interpolate the intermediate points of the link along the X and Y dimensions between the ports
- **JGoLink.AdjustingStyleEnd** – just modify the end point(s), leaving the other points of the link stroke unchanged

For an example of what occurs with a cubic link, before and after one node is moved:



When the link is **Orthogonal**, an **AdjustingStyle** value of **AdjustingStyleScale** is treated as if it were just **AdjustingStyleCalculate**, since one cannot maintain orthogonality and similarity of shape. Also, an **AdjustingStyle** value of **AdjustingStyleStretch** is treated as if it were just **AdjustingStyleEnd**, again because orthogonality cannot be maintained. Furthermore, the **AvoidsNodes** property makes the **AdjustingStyle** property moot—the path is always recalculated.

When the **AvoidsNodes** property is set to true, the **calculateStroke** method computes a short path between the nodes that tries not to cross any nodes. Some customization of this method can be achieved by overriding the **JGoDocument.isAvoidable** method to control which top-level objects it tries to avoid, and by overriding the **JGoDocument.getAvoidableRectangle** method to specify how much of each avoidable node to consider avoiding.

A selected link will not have selection handles at the very end points, unless there is only one segment in the stroke. Resizing a link by dragging an end selection handle causes the link to be reconnected. The existing link is disconnected from one port. When the link gesture is completed the port is set again. You can disable this user-relinking behavior by setting the **Relinkable** property to false.

The default resize behavior for interior stroke points simply moves that point, rather than cause the link to be deleted and a new one started. If the link is orthogonal, the resizing moves that middle segment to maintain orthogonality. The sample applications include code to let the user insert new points and remove segments.

JGo User Guide

Labeled Links

This subclass of **JGoLink** that supports managing up to three additional objects located near either end and near the middle of the link. The **JGoLabeledLink** class has three properties: **FromLabel**, **MidLabel**, and **ToLabel**, which can be **null** or any **JGoObject**. **JGoLabeledLink** overrides **calculateStroke** to perform the default stroke calculation and then position each of its (non-null) labels to be near their respective points of the link. The methods **positionEndLabel** and **positionMidLabel** try to be smart about placing the labels where they do not overlap the link stroke too much, but you can override these methods to implement your own positioning policies.

The labels can be any object but are usually instances of **JGoLinkLabel**, a subclass of **JGoText**. A **JGoLinkLabel** has a transparent background by default, and is not resizable or draggable.

A **JGoLabeledLink** is not a **JGoArea**, even though it might appear that it has parts as an area does. The labels are inserted in the link's parent collection (normally a document layer, but perhaps an area or a view) right after the link, so that they appear just on top of the link. But one property of areas has been reused for labeled links:

GrabChildSelection. If a **JGoLinkLabel** is not selectable, the **JGoLabeledLink** will be selected if the **GrabChildSelection** property is **true**, which it is by default.

4. JGoVIEW DETAILS

JGoView is a lightweight component that supports the display and editing of graphical objects such as nodes and links.

JGoView supports the model-view-controller architecture. **JGoDocument** is the model for **JGoView**.

JGoView supports many basic features:

- displaying a **JGoDocument** and its **JGoObjects**
- displaying its own view-specific objects
- scrolling and scroll bars
- autoscrolling
- scaling (zooming)
- printing
- clipboard transfer
- drag-and-drop
- default keyboard commands
- view events and listeners
- selection
- creating links between ports
- resizing objects
- handling single clicks, double clicks
- handling mouse move and tool tips
- in-place text editing
- default cursor
- painting a background color and a background image
- displaying a grid
- constraining object moves and resizes to a grid

Display

The primary purpose of **JGoView** is to display a **JGoDocument** and its **JGoObjects**. You can use the default **JGoDocument** that is created for the default **JGoView** constructor, or you can supply your own, either at construction time or later by calling **setDocument**. It is also common to override **createDefaultModel** so that the default constructor for your view subclass will automatically create your own document class too.

A **JGoView** is just a regular **JComponent**. The part of a **JGoView** that shows the document is called the canvas. A view can also have scroll bars. A view, like any **JComponent**, can have a border, but the canvas does not support one.

JGoView also supports the display of its own view-specific objects. Thus each view on the same document can have its own set of **JGoObjects**. These view objects will appear in front of all document objects. The most common example of a view object is a selection handle (a **JGoHandle**).

Scrolling

JGoView has support for scrolling and scroll bars built in; the use of **JScrollPane** and **JViewport** is not necessary. By default there will be both a horizontal and a vertical scroll bar, but you can remove one or both of them by setting the respective properties to **null**. There is also a separate corner component, where the two scroll bars meet, that is visible when both scroll bars are visible.

Of course users can scroll the view by manipulating the scroll bars. When drag-and-drop is enabled, users can also cause automatic scrolling when they are dragging near the edge of the canvas. This autoscroll region is specified by the **AutoscrollInsets** property.

Because a view does not necessarily show the whole document, the **ViewPosition** property indicates where the view's top-left corner is in the document. The **ExtentSize** property indicates the size of the view's canvas in the document.

Each view also provides **getDocumentSize** and **getDocumentTopLeft** methods, which allow each view to have a potentially different notion of the document it is looking at. In particular, the **includingNegativeCoords** property affects the behavior of both of these methods. A **false** value prevents users from scrolling to parts of the document at negative coordinates. Alternatively, a **true** value allows the objects of the document to be placed anywhere, which can be convenient when additional objects need to be added to the left of the existing ones, and you don't want to shift the existing ones rightwards.

Scaling and Coordinate Systems

JGoView also supports zooming, to change the scale at which the objects are drawn. The **Scale** property is normally 1.0; smaller values make objects appear smaller on the screen; larger values correspond to zooming into the diagram.

The ability to scroll and zoom the view means that the coordinate system used in a view is different from that used in the document. The **convertDocToView** and **convertViewToDoc** methods perform the basic transformations of **Points**, **Dimensions**, and **Rectangles**. The **docToViewCoords** and **viewToDocCoords** methods are retained

from earlier versions, but can be less efficient to use because they allocate new objects for the return values.

Painting

As a **JComponent**, **JGoView**'s canvas overrides **paintComponent** in order to render the view. This just calls **JGoView.onPaintComponent**, which is responsible for scaling and translating the **Graphics2D** and getting a document-coordinates clipping rectangle. It then calls **paintView**, which calls methods to fill in the paper color, to draw any additional background, to draw all of the document objects (layer by layer), and then to draw any view objects. You can override **paintView** or any of the four methods called by **paintView** in order to get different effects; overriding **paintPaperColor** and **paintBackgroundDecoration** are the most common. Note that setting the **BackgroundImage** property affects the standard implementation of **paintBackgroundDecoration**.

The **paintView** method enables anti-aliasing for all non-text painting of objects in the document or in the view. You may want to override **paintView** or **paintDocumentObjects** to change the defaults.

Printing

JGoView also provides support for printing. The **print** method brings up the print dialog and then starts a **PrinterJob**. You can easily override **getPrintDocumentSize**, **getPrintPageRect**, and **getPrintScale** to customize how much is printed, on how much of the page, and at what scale. Override **printDecoration** to add headers and/or footers or any other decoration on each page. Override **printView**, like **paintView**, to change what things get printed--by default the paper color and the view objects are not printed.

The **JGoView.printPreview** methods, along with the **JGoPrintPreview** class provide a simple print-preview facility for the user.

The Demo1 example view, Demo1View.java, includes example code for several different common possibilities.

Selection

Each **JGoView** has a **JGoSelection** that holds the currently selected document objects for that view. The default selection object is an instance of **JGoSelection**, but you can override **createDefaultSelection**. The selection object is also responsible for managing selection handles in the view. Many methods in **JGoView** deal with the current selection, either by changing it, or by operating on its collection of objects. Important examples include: **cut**, **copy**, **deleteSelection**, **moveSelection**, **copySelection**, and **selectAll**.

You can also control the color of the primary selection object's handles as well as the color of the selection handles of all other selected objects by setting the **PrimarySelectionColor** and **SecondarySelectionColor** properties of **JGoView**.

Events

JGoView is a **JGoDocumentListener**, which is how it can keep its display up-to-date with changes to the document and its objects.

JGo User Guide

If you do nothing to override the input handling of a **JGoView**, the default behavior gives you input handling that anyone familiar with a graphical object editor would expect. Objects can be selected, moved, and resized using the left mouse button. Multiple selections can be made using control-left button or with rubber-band selection. Links can be created by left button down-drag over a **JGoPort**.

By default most events are ignored if the view does not have focus. A mouse pressed event will try to acquire focus.

The document property **Modifiable** affects view behavior. When the view's document **isModifiable()** is **false** (the default value for the property is **true**), **JGoView** disables the user's ability to move and resize objects, to link ports, to drop objects, and to edit text. Selection and scrolling and other event handling continue to operate normally if they do not normally modify the document. Furthermore, this property is available on **JGoLayer**, and **JGoLayer.isModifiable()** is only true when both the layer and the document are modifiable. Thus you can easily turn off user modification of a whole set of objects, if they all belong to one layer, without disabling user modification of all the other objects in the document.

View Events

JGoView also has view-specific state and general actions that other objects may care about tracking. Thus a **JGoView** will notify its **JGoViewListeners** about any **JGoViewEvents**. Such events include:

- inserting, changing, and removing view objects (but not document objects)
- adding and removing objects from the view's selection
- single and double clicking on objects or in the background
- moving, copying, or deleting the selection
- drawing a new link or reconnecting an existing link
- finishing in-place editing of text
- pasting from the clipboard or dropping objects from another window
- changing the view's position and scale or other view properties

A listener can call **JGoViewEvent.getHint()** to distinguish between the different kinds of events. The following table lists the standard abstract events that a view will fire.

JGoViewEvent hint	Method that fires the event
CLICKED (single click on a document object)	JGoView.doMouseClicked
DOUBLE_CLICKED (on a document object)	JGoView.doMouseDbClick
BACKGROUND_CLICKED	JGoView.doBackgroundClick
BACKGROUND_DOUBLE_CLICKED	JGoView.doMouseDbClick
SELECTION_GAINED (object added to selection)	JGoSelection methods

SELECTION_LOST (object removed from selection)	JGoSelection methods
SELECTION_STARTING (before a possibly big selection change)	JGoSelection.clearSelection, JGoView.selectAll, copySelection, deleteSelection, paste, doDrop
SELECTION_FINISHED (after a possibly big selection change)	JGoSelection.clearSelection, JGoView.selectAll, copySelection, deleteSelection, paste, doDrop
SELECTION_MOVED	JGoView.doMoveSelection
SELECTION_COPIED	JGoView.doMoveSelection
SELECTION_DELETING (before actually removing objects from document; call consume() to cancel)	JGoView.deleteSelection and JGoView.noReLink
SELECTION_DELETED (after objects are removed from document and from selection)	JGoView.deleteSelection and JGoView.noReLink
OBJECT_RESIZED	JGoView.handleResizing
LINK_CREATED	JGoView.newLink
LINK_RELINKED	JGoView.reLink
OBJECT_EDITED	JGoText.doEndEdit
CLIPBOARD_PASTED	JGoView.paste
CLIPBOARD_COPIED	JGoView.copy and JGoView.cut
EXTERNAL_OBJECTS_DROPPED	JGoView.drop

You can add a **JGoViewListener** to respond to any of these events. For example, you can bring up a confirmation dialog when the user tries to delete something.

```
myView.addViewListener(new JGoViewListener() {
    public void viewChanged(JGoViewEvent e) {
        if (e.getHint() == JGoViewEvent.SELECTION_DELETING) {
            String msg = "Really delete ";
            msg += Integer.toString(myView.getSelection().getNumObjects());
            msg += " objects?";
            if (JOptionPane.showConfirmDialog(myView.getFrame(), msg,
                "Confirm Delete", JOptionPane.YES_NO_OPTION,
                JOptionPane.QUESTION_MESSAGE)
                == JOptionPane.NO_OPTION) {
                e.consume();
            }
        }
    }
});
```

High Level Mouse Events

One of the more important functions of **JGoView** is the ability to handle mouse clicks. The selection may change or a click will be passed on to any visible object on top at that point. This will cause **JGoViewEvents** to be fired off to any interested listeners, and will call **doMouseClicked** or **doMouseDownClick**. This method then calls the method of the same name on the object and on its parents up to the top-level object until a call returns **true**, indicating that it completely handled the single-click or double-click. If there is no object at the mouse point, **doBackgroundClick** is called, or **doMouseDownClick** returns false.

Similarly, when the mouse moves without any mouse button being held down, **doUncapturedMouseMove** is called, which in turn calls the same-named method on the object underneath that point, and on up the parent chain until a call returns **true**. Getting a tool tip text is also similar, in that the view passes the request down to a particular object at that point, and to its parents, until it gets a non-**null** string.

For convenience the parameters on the **do...Mouse...** methods take an integer (the event-modifiers) and two **Points** (mouse event location in both document and view coordinates). You can get the original **MouseEvent** via the **getCurrentMouseEvent** method. If you use **getCurrentMouseEvent**, be aware that this method may return **null** if not invoked from a mouse handler method.

To implement popup menus, you should include the following overrides in your view subclass:

```
// invoke popup menu if needed
public boolean doMouseDown(int modifiers, Point dc, Point vc)
{
    if (getCurrentMouseEvent() != null &&
        getCurrentMouseEvent().isPopupTrigger()) {
        if (doPopupMenu(modifiers, dc, vc))
            return true;
    }
    // otherwise implement the default behavior
    return super.doMouseDown(modifiers, dc, vc);
}

public boolean doMouseUp(int modifiers, Point dc, Point vc)
{
    if (getCurrentMouseEvent() != null &&
        getCurrentMouseEvent().isPopupTrigger()) {
        if (doPopupMenu(modifiers, dc, vc))
            return true;
    }
    // otherwise implement the default behavior
    return super.doMouseUp(modifiers, dc, vc);
}
```


Both overrides are needed because the **isPopupTrigger** predicate will be true at different times on different platforms. The **doPopupMenu** method might be implemented in different manners depending on your needs. One possibility, supporting popup menus both in the background and for objects that are instances of **MyNode**, follows:

```
public boolean doPopupMenu(int modifiers, Point dc, Point vc)
{
    JGoObject obj = pickDocObject(dc, true);
    if (obj == null) {
        getBackgroundPopupMenu().show(this, vc.x, vc.y);
        return true;
    } else if (obj.getTopLevelObject() instanceof MyNode) {
        selectObject(obj);
        getMyNodePopupMenu().show(this, vc.x, vc.y);
        return true;
    }
    return false;
}
```

We may add support for something like **doPopupMenu** in a future version of **JGoView**.

Resizing

Views also have default behavior for resizing objects. When the user does a mouse down on a selection handle, the view goes into resizing mode. This causes the **handleResizing** method to be called while the mouse is dragging the selection handle. This method in turn calls the **handleResize** method on the selected object, assuming it **isResizable**. The object can then decide how to interpret the resize request.

JGoView's default behavior is to draw an XOR box during the resizing, and to reshape the object when the resizing is done. You can easily override this behavior to redraw the object continuously with the resizing, instead of drawing the XOR box. If the user cancels the resizing with the Escape key, the object is restored to its original size and position.

When the object is finally resized, there will be a **JGoDocumentEvent.CHANGED** event for that object, because its geometry will have been changed.

Drag and Drop

JGoView has a default behavior for drag-and-drop. Each view has a drag gesture recognizer and acts as both a drag source and as a drop target. Within a view, a drag moves the selected objects; between views a drag and drop copies the selected objects, and from another drag source the view can decide to accept the drop and to handle it in an application specific manner. If the user cancels a drag from a **JGoView**, the selected objects are restored to their original locations.

To customize a view as a drop target from other components, you'll want to override several methods: **isDropFlavorAcceptable**, **computeAcceptableDrop**, and **drop**. If you want to make use of the default copy behavior, you can call **doDrop** from your override

of **drop**. If you need somewhat more extensive customization, you can just override all the standard **DropTargetListener** methods.

The implementation of drag-and-drop does not actually make the **JGoView** the listener for drag gestures, drag source events, or drop target events. It is actually the view's canvas, which delegates to internal methods of the view, which in turn call the respective standard listener methods. The reason for this additional step is to allow default support for dragging selection handles, drawing a rubber-band selection box, or starting a link between ports. If you really need complete control of mouse-related operations for a view, you'll need to override the internal handler methods, whose names are prefixed with “on”, rather than overriding the “do” methods.

You can control whether a view handles any mouse events or any drag-and-drop behavior by setting the **MouseEnabled**, **DragEnabled**, and **DropEnabled** properties. If you turn off drag-and-drop, of course the user will not be able to perform any standard drag-and-drop between windows. However, much of the behavior within a **JGoView** will continue to function, perhaps with slightly different behavior or appearance.

Remember that all selected **JGoObjects** that are dragged and dropped need to be serializable. If serialization fails, perhaps because your object has a reference to an object that is not itself serializable, the serialization failure exception will be caught by the drag-and-drop system, resulting in a drag-and-drop failure rather than the behavior you expect.

Views have additional default behavior for drags within a view: the user can either move or copy the selection, using the CTRL key as a modifier to indicate copy rather than move. The copying is indicated with an outline of the selected objects that follows the mouse while the CTRL key is down. The actual copying of the selection and addition to the document is performed only if the CTRL key is still down at the time of the drop. The original selection remains at its original location; the newly copied objects become the new selection at the drop location.

The behavior for view-internal drags is controlled by the **InternalMouseActions** property. The default value is **DnDConstants.ACTION_COPY_OR_MOVE**. Set it to **DnDConstants.ACTION_MOVE** to get move-only behavior, or set it to **DnDConstants.ACTION_NONE** to disable all internal selected object drags, without disabling selection, resizing, linking, or other default view behaviors.

You can further customize drags within a view by setting the **DragsRealtime** and **DragsSelectionImage** properties. By setting the **DragsRealtime** property to false the user's moving the selection will not actually cause those objects to be continuously moved; instead the user will move an image of the selection, and the objects are actually moved only upon a successful drop. The **DragsSelectionImage** property (which defaults to true) controls whether a user's drag will drag an image of the selection or an outline of the selection.

Customizing the Mouse Behavior

You may want additional behaviors or “modes” of operation for the user. For example, you may want to allow the user to draw a stroke by specifying the points of the stroke by clicking. You can accomplish this by overriding the **JGoView** methods **doMouseDown**, **doMouseMove**, **doMouseUp** and **doCancelMouse**.

The current mouse state of the view is accessible as the **State** property of **JGoView**; predefined values include the **JGoView** constants starting with the “**MouseState**” prefix:

- **MouseStateNone**
- **MouseStateSelection**
- **MouseStateMove**
- **MouseStateCreateLink**
- **MouseStateCreateLinkFrom**
- **MouseStateResize**
- **MouseStateDragBoxSelection**
- **MouseStateLast**
- **MouseStateAction**

You can define your own modes or states by using values larger than **MouseStateLast**. Each mode has its own specific prerequisites in order to operate properly, so you should be careful about interacting with the existing behavior. For convenience there is a **CurrentObject** property that normally holds the current **JGoObject** relevant to the current view state.

The **Demo1View** class of the **Demo1** sample application implements a user drawing stroke mode. The **Processor** sample application implements a custom user-specified orthogonal linking operation.

Clipboard

JGoView supports copying the selection to and from the system clipboard; use the **copy**, **cut**, and **paste** methods. These methods, like the default drag-and-drop implementation, depend on the document's **DataFlavor**, the document's and selection's implementation of **Transferable**, and the document's **copyFromCollection** method.

Keyboard Commands

A view can accept keyboard focus and can respond to several keyboard commands by default. Override **onKeyEvent** to change or augment the default commands. You can control whether there is any default key event handling by setting the **KeyEnabled** property.

User Editing

Creating Links

Another important **JGoView** feature is the support for the user creating **JGoLinks** between ports by "dragging" from a **JGoPort** to another one. The **startNewLink** method uses **validSourcePort** and **validDestinationPort** to see if the port under the mouse point will permit the user's starting a new link. If so, the view creates a temporary port and a temporary link from the port to the temporary port. While the user remains in this

JGo User Guide

creating-a-new-link mode, the temporary port is continuously moved to follow the mouse.

Furthermore the view checks to see which ports to which it could make a valid new link, by calling **validLink** for all potential pairs of ports involving the original one. The default implementation of **validLink** just asks the "from" port if it can be linked to the "to" port; this allows the behavior to be overridden either in the port class or in the view.

To make drawing links easier for the user, there is also the notion of "port gravity", a distance. The temporary port automatically snaps to the location of the closest valid port within the port gravity distance.

Finally, when the user releases the mouse to create the link, the **newLink** method is called. This method is responsible for creating the real **JGoLink** (either that class or a user-defined subclass) in the document connecting the two ports; the temporary port and link are discarded. If for some reason the link is not made, because the attempted link was invalid or because the user cancelled the link drawing process, the **noNewLink** method is called. This allows views to clean up any other state or inform the user or do some other default failure action.

In-place Text Editing

Another handy feature that **JGoView** offers is in-place text editing. If a **JGoText** object is editable, then clicking on it may put it into editing mode, where the user can change the string. This is accomplished by creating a temporary **JGoTextEdit** object in this view and having it be responsible for actually creating and displaying a **JTextComponent** and handling its editing completion or cancellation. The **JGoTextEdit** object is held as a property of the view (**JGoView.getEditControl()**). Use **doEndEdit** to stop any in-place text editing in progress.

You can control how the user can enter text-editing mode by setting the **Editable** property of the **JGoText** object. By default this happens when the user double-clicks on the text. By setting the **EditOnSingleClick** property, the user can just click on the text to start editing it.

If you want to detect when the user has edited a **JGoText** object:

```
myView.addViewListener(new JGoViewListener() {
    public void viewChanged(JGoViewEvent e) {
        if (e.getHint() == JGoViewEvent.OBJECT_EDITED) {
            JGoText text = (JGoText)e.getObject();
            JGoObject node = text.getParentNode();
            if (node instanceof MyNode) {
                MyNode m = (MyNode)node;
                String s = text.getText();
                . . . update my database for node m to have value s
            }
        }
    }
});
```

Alternatively, you could implement a **JGoDocumentListener** (or, more efficiently, override **JGoView.documentChanged**) to do something similar:

```
myView.getDocument().addDocumentListener(new JGoDocumentListener() {
```

```

public void documentChanged(JGoDocumentEvent e) {
    if (e.getHint() == JGoDocumentEvent.CHANGED &&
        e.getFlags() == JGoText.ChangedText) {
        JGoText text = (JGoText)e.getObject();
        JGoObject node = text.getParentNode();
        if (node instanceof MyNode) {
            MyNode m = (MyNode)node;
            String s = text.getText();
            . . . update my database for node m to have value s
        }
    }
}
});

```

The difference is that the document listener will be invoked whenever the **JGoText** label's **Text** string is modified, for any reason. The view listener is called only after the user edits the label interactively.

If you want to do some validation of the user's text entry, you can override **JGoText.doEdit**. Here is an example:

```

// do some validation--don't allow integers larger than 1000
public boolean doEdit(JGoView view, String oldtext, String newtext) {
    try {
        int i = Integer.parseInt(newtext);
        if (i > 1000) {
            view.doCancelMouse();
            javax.swing.JOptionPane.showMessageDialog(view, Integer.toString(i)
                + " is too big!");
            return false;
        }
    } catch (NumberFormatException ex) {
        // allow non-integers to pass validation
    }
    return super.doEdit(view, oldtext, newtext);
}

```

Returning false will leave the text editing component up; returning true will cause **doEndEdit** to be called to remove the text editing component.

Note the call to **JGoView.doCancelMouse**, to make sure no mouse operation is ongoing in the view during or immediately after the presentation of the dialog.

Focus and JInternalFrame

If you want to use **JGoView** inside **JInternalFrame** there are some complications you will need to consider due to differences between, and bugs in, the 1.2, 1.2.2, 1.3+, and 1.4+ versions of Java.

In JDK version 1.2[.0] component focus and internal frame selection are completely independent of each other. Creating an internal frame makes it visible but does not assign focus to a component in the internal frame nor does it activate (select) it.

When the user performs a mouse pressed action on an unselected **JGoView** without focus, the internal frame is activated but focus does not change. The view does get a mouse pressed event, but unfortunately there is a bug that prevents the view from getting

JGo User Guide

a mouse released event. This can cause undesirable behavior when the **JGoView** is unselected but has kept the focus and the user clicks on the view.

To avoid this problem make sure that activating any **JInternalFrame** is accompanied by setting the focus on one of its components. **JGoView** itself tries to acquire focus on a mouse pressed event. It avoids most of the problems by basically ignoring the user's first mouse press.

In JDK 1.2.2 this bug was fixed, but a different one took its place. When an unselected **JGoView** is activated the view starts getting drag events, even though the user really hadn't started a drag. **JGoView** tries to avoid this problem by temporarily disabling drag-and-drop behavior. This means that any initial dragging gesture by the user will be ignored.

JDK 1.3 introduced much new behavior with internal frames. You will need to explicitly call **setVisible(true)** in order to show the internal frame. **JInternalFrames** keep track of their last focused components. The problems with 1.3 include the spurious drag events from 1.2.2 and a new one. The problem is that clicking on a component in a **JInternalFrame** causes the component's layer to be reset. This means that the **JGoView** will get the focus and then lose it because of the view being removed and then added back into the **LayeredPane** of the **JInternalFrame**. **JGoView** tries to ameliorate the situation by explicitly making the enclosing **JInternalFrame** unselected and then selected.

JDK 1.4 changed the focus architecture yet again. The operations are clearer, and the work-around of unselecting and then reselecting the frame is not needed. But gaining focus can happen later than it did in earlier versions. We have adapted **JGoView** accordingly, particularly for in-place text editing.

5. NODES

As noted previously, sets of JGo primitive objects can be combined into higher-level grouped objects. One of the most common applications of this technique is in creating a “node” for a diagram. A node is a **JGoArea** that contains some **JGoPorts**, thus allowing the nodes to be connected to each other with **JGoLinks**.

The **JGoNode** class extends **JGoArea** to provide several useful features that practically all “nodes” have. Each node has several properties:

- **Label**, providing access to the principal **JGoText** object in the area
- **Text**, which by default accesses the string in the **Label**
- **ToolTipText**, which if non-null, is a string to be displayed in a tooltip when the user hovers over the node
- **PartID**, a unique integer identifying this node, if the node’s document’s **MaintainsPartID** property is true
- **UserObject**, an arbitrary object that programmers can use to associate their own information with the node (i.e., this property is not used by JGo)
- **Flags**, an arbitrary integer that programmers can use for application-specific purposes (i.e., this property, which is actually on **JGoObject**, is not used by JGo)

All of the classes in JGo whose names end in “Node”, whether part of the JGo package or as examples, extend **JGoNode**. We recommend that you extend the **JGoNode** class or one of its subclasses when you want to define your own area containing ports.

Three very commonly used kinds of nodes are included in the JGo package, along with one kind that can hold nested graphs:

- **JGoBasicNode**, an elliptical or rectangular node with a single port and a single optional text label that can be positioned at different spots relative to the drawable shape
- **JGoIconicNode**, the simplest node with an image for the icon, a text label, and a single port
- **JGoTextNode**, a node displaying text with a background shape and four ports, one at the center of each side
- **JGoSubGraph**, a node that can hold a graph within the area, with optional label, background color and border, and that can be collapsed/expanded by the user

JGo User Guide

But many useful examples are provided for you in the examples directory. These include:

- **SimpleNode**, a node with an icon, a label, and two ports
- **GeneralNode**, a node with an icon, labels at the top and bottom, and variable numbers of labeled ports on the left and right sides
- **MultiPortNode**, a node like **JGoIconicNode**, but with a variable number of ports that can be positioned arbitrarily on the node
- **MultiTextNode**, a node displaying a list of objects (typically **JGoTexts**) with a pair of objects (typically **JGoPorts**) on each side of each item, separated by lines and backed by a **JGoDrawable**, with an additional pair of objects (again, typically ports) at the top and bottom of the node
- **ListArea**, an area that organizes a list of objects, much as **MultiTextNode** does, but with a scroll bar
- **RecordNode**, a complex node that uses a **ListArea** to hold a scrollable list of objects with ports, and adds header and footer objects
- **Comment**, an area that displays multi-line text with a background that looks like a notepad

Other potentially useful node, link, and drawable classes are provided in the example app subdirectories, including:

- **ClassNode** (in **Classier**)
- **Demo1RoundRect** (in **Demo1**)
- **Diamond** (in the examples directory)
- **PersonNode** (in **FamilyTree**)
- **ActivityNode** (differently both in **Flower** and in **Processor**)
- **FlowLink** (differently both in **Flower** and in **Processor**)
- **WebNode** (in **WebWalker**)

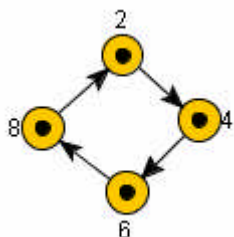
For all of these examples, be sure to look at the source code for more descriptions and details.

JGoBasicNode

The **JGoBasicNode** class is useful when you want to display some text and there is just a single port centered in an ellipse or rectangle. Use **JGoBasicNode** when you expect that there may be links coming in or going out in different directions, and you would like the link to seem to originate from the center of the ellipse or rectangle.

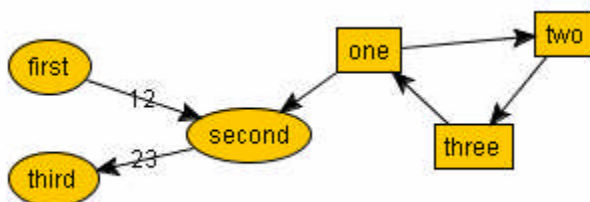
A **JGoBasicNode** has properties for its parts: **Drawable**, **Label**, and **Port**. As with all **JGoNode** subclasses, there is easy access to the text string by means of the **Text** property. **JGoBasicNode** also provides convenient access to the **Drawable**'s **Pen** and **Brush** properties.

You can control the relative position of the **JGoText** label to the **JGoDrawable** background shape by setting the **LabelSpot** property. For example, the following diagram shows four different **JGoBasicNodes**, each with a different **LabelSpot**.



Users can draw new links interactively by dragging from the ports in the center of the nodes. They can select/move/copy nodes by a mouse-down on the text or on the rest of the drawable shape.

When the **LabelSpot** is **JGoObject.Center**, then the layout of the node is altered so that the background shape is expanded to surround the text.



The port's style is changed to be **JGoPort.StyleHidden**, and its size and position are changed to match that of the background drawable shape. Thus users will be able to interactively draw links between such **JGoBasicNodes** by starting to drag in the drawable shape but outside of the text itself. Users can drag the node by dragging the text. In this configuration it is easy to disable user-drawing of new links for a particular node by making the port invisible: **aBasicNode.getPort().setVisible(false)**.

The size of the drawable's object is determined by the size of the text label. The **Insets** property specifies how much bigger to make the drawable object than the label. This is particularly useful for shapes such as **JGoEllipse** or **Diamond** that would otherwise cut off part of the text.

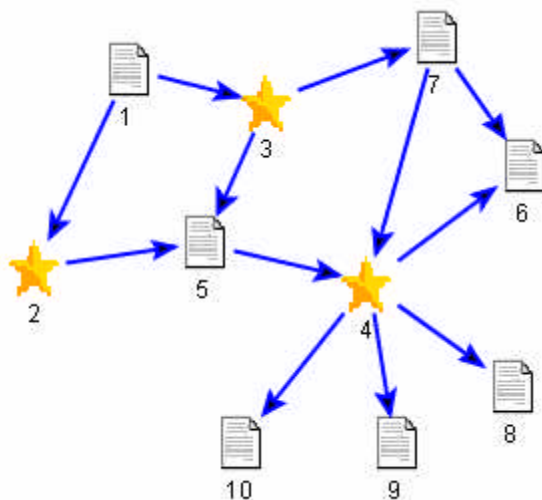
If however you want the drawable's size to remain constant even when the label's size changes due to text changes, you can call **setAutoResize(false)** to cause the **JGoText** label to be **Multiline**, **Wrapping**, and **Clipped**, with a **WrappingWidth** determined by the width of the drawable minus the **Insets**. It will display as much text as will fit. Usually you will want to call **getLabel().setAlignment(JGoText.ALIGN_MIDDLE)** to have the text be centered in the node.

If you create a **JGoBasicNode** by using the zero-argument constructor, it will not have any parts. You will need to create and assign the parts yourself. If you use the **JGoBasicNode** constructor that takes a string, the resulting node will have a **Label**, a **Port**, and a **Drawable** that is a **JGoEllipse**.

JGoIconicNode

A **JGoIconicNode** is the simplest node that has an icon. It has a text label and a single port. You'll want to use **JGoIconicNode** in the same circumstances as **JGoBasicNode**, except that **JGoIconicNode**'s primary display is an icon rather than a text object with a rectangle or ellipse. Links appear to originate from the center of the icon.

A **JGoIconicNode** has properties for its parts: **Icon**, **Label**, and **Port**. The **Text** property provides convenient access to the label's text string. The **Icon** need not be an instance of **JGoImage**, although it is by default. For convenience, the **Image** property casts the **Icon** as a **JGoImage**.



Like a **JGoBasicNode**, the port of a **JGoIconicNode** is at the center of the icon, but its style is **JGoPort.StyleHidden** so it does not obscure the icon. Users can draw links from and to the icon. Users can drag the node by dragging the text or part of the icon outside of the port. It is also easy to disable a user's drawing of new links for a particular node by making the port invisible.

If you create a **JGoIconicNode** by using the zero-argument constructor, it will not have any parts. You will need to create and assign the parts yourself. If you use the **JGoIconicNode** constructor that takes a string, the resulting node will have a **Label** and a **Port** and an **Icon** that is a "blank" **JGoImage**. You will still need to initialize the image, perhaps as follows:

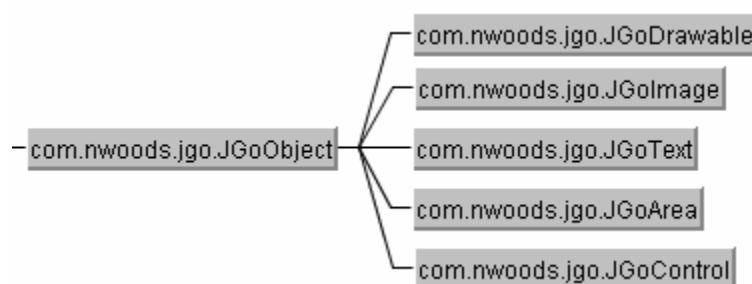
```
JGoIconicNode inode = new JGoIconicNode("an iconic node");
inode.getImage().setSize(50, 50);
inode.getImage().loadImage(Demo1.class.getResource("doc.gif"), true);
. . . other inode initialization . . .
```

If the **DraggableLabel** property is set to true, the user is able to drag the text label around. Moving the node then also drags the label, keeping it at the same position relative to the icon.

JGoTextNode

When you have a lot of textually identified objects to display, and the links between them tend to be organized in a horizontal and/or a vertical direction, then the **JGoTextNode** may be what you would prefer to use in place of a **JGoBasicNode**. Each **JGoTextNode** has four ports—one in the middle of each side of the node. When the links are connected to the appropriate ports, the links tend to be more cleanly organized, whether using the default three-segment stroke or the five-or-more-segment strokes when the **JGoLinkOrthogonal** property is true.

The **Text** property provides access to the label's string value.



By default the four ports are very small and of style **JGoPort.StyleHidden**. You may find it convenient to remove unneeded ports by setting them to null; e.g., **myTextNode.setTopPort(null)**. The **Insets** property determines how much space there is on each side of the **Label**, a **JGoText**.

The size of the background's object is determined by the size of the text label. The **Insets** property specifies how much bigger to make the background object than the label. This is particularly useful for shapes such as **JGoEllipse** or **Diamond** that would otherwise cut off part of the text.

If however you want the background's size to remain constant even when the label's size changes due to text changes, you can call **setAutoResize(false)** to cause the **JGoText** label to be **Multiline**, **Wrapping**, and **Clipped**, with a **WrappingWidth** determined by the width of the drawable minus the **Insets**. It will display as much text as will fit. Usually you will want to call **getLabel().setAlignment(JGoText.ALIGN_MIDDLE)** to have the text be centered in the node.

If you create a **JGoTextNode** by using the zero-argument constructor, it will not have any parts. You will need to create and assign the parts yourself. If you use the **JGoTextNode** constructor that takes a string, the resulting node will have a **Label**, four **JGoPorts** (**TopPort**, **RightPort**, **BottomPort**, **LeftPort**), and a **Background** that is a **JGo3DRect**.

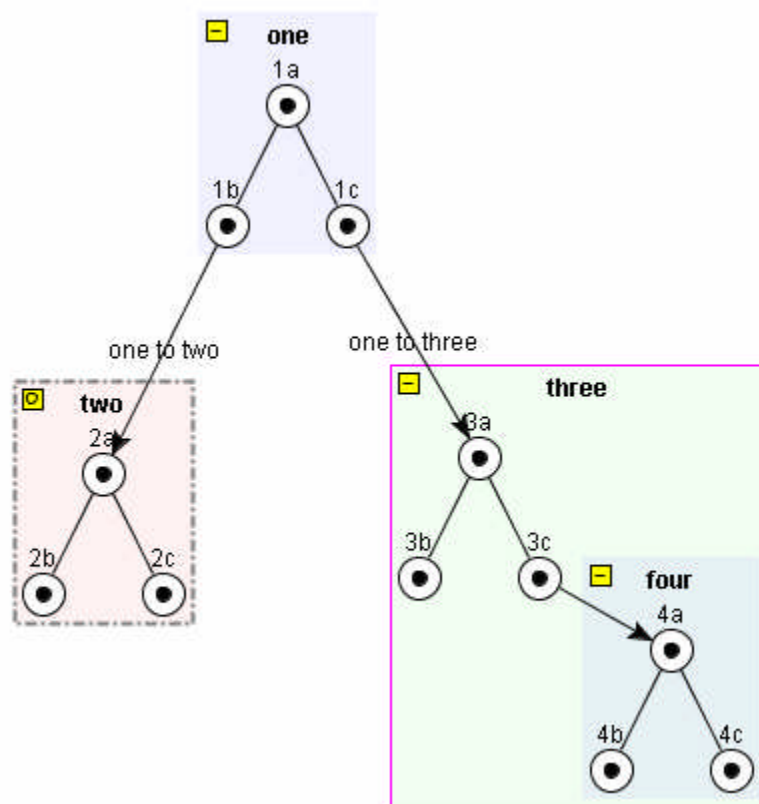
The Classifier example's **ClassNode** class, which extends **JGoTextNode**, also handles selection and double-click by changing the background color and by displaying much more information in the text label.

JGoSubGraph

Often the information model you wish to display to your user can be organized as a graph whose nodes can themselves be displayed as graphs. One way of handling this is to implement your application as a multiple-document interface application, where drilling into a node brings up a new child window showing the detail diagram. This helps keep individual graphs relatively small and simple.

But another way of displaying graphs within nodes is to show the graphs in-place. The **JGoSubGraph** class provides some useful features in this regard—it adds a label that can be positioned at various spots in the area, a background color, and a border.

A **JGoObject** property, **DragsNode**, controls whether an object that is selected can be dragged around by the user independently of the parent area. **JGoSubGraph** automatically sets that property to false for all objects that are added as immediate children of the area. This allows users to move the nodes around within the **JGoSubGraph**.



Each subgraph has a small rectangular handle at the top-left corner. This instance of **JGoSubGraphHandle** can be clicked by the user to collapse and expand the subgraph. All of the children (besides the label and the handle itself) are made not **Visible** and are moved to the top-left corner of the subgraph. The label is moved to the middle of the (now-reduced-size) subgraph node. Collapsing also remembers all of the positions of the children so that an expansion will restore the original positions of the nodes within the subgraph.

The **TestSubGraph** class in the Demo1 example demonstrates adding an input and an output port to a **JGoSubGraph**, as if the subgraph were like a **SimpleNode**.

The **TestSubGraph2** class, also in Demo1, demonstrates having a single port representing the subgraph as a whole node, in addition to whatever ports the subgraph's child nodes may have. Its **Port** is an instance of **TestSubGraph2Port**, which is a subclass in order to override **JGoObject.pick** to make the port act as if it were hollow, so that only the outer margin of the port is active to the user for initiating the drawing of a new link. The **layoutChildren** override makes sure the **Port**'s bounding rectangle covers the whole subgraph; by default the **Port** would have had the same bounds as the **Handle**.

Demo1 furthermore provides a **CollapsedObject** for the instances of **TestSubGraph** and **TestSubGraph2** that it creates. The subgraph's collapsed object is shown instead of the background and border when the subgraph is collapsed. In Demo1, the collapsed object is just a **JGoImage**.

You can specify the location of the **Label** relative to the subgraph's children by setting **JGoSubGraph**'s **LabelSpot** and **CollapsedLabelSpot** properties.

JGoSubGraphs are also resizable by the user. Resizing just modifies the **Insets** property (or the **CollapsedInsets** property if the subgraph is collapsed), so that resizing the whole subgraph does not resize (or reposition) any of the subgraph's child nodes.

Although it should be clear that a **JGoSubGraph** should contain the various child nodes that are part of it, it should also be the parent of all of the **JGoLinks** that connect the subgraph's children. This is needed so that copying a subgraph will properly copy all of the links that appear to belong to the subgraph because they connect nodes that appear to belong to the subgraph. A link between a subgraph's child node and another top-level node cannot belong to the subgraph, of course, but will need to belong to the **JGoDocument**. And similarly, a link between a subgraph's node and another subgraph's node, if one subgraph is a child of the other one, will need to belong to the "least common parent" subgraph.

Thus **JGoView.newLink** and **JGoView.reLink** automatically call **JGoSubGraphBase.reparentToCommonSubGraph** to make sure any newly drawn links or newly reconnected links belong to the proper subgraph, if any.

Although Java2D fully supports translucency via the alpha property of Colors, you may find it convenient to specify a solid/opaque **BackgroundColor** and then control the transparency by setting the **Opacity** property. The **Opacity** property is ignored when the **BackgroundColor** has an alpha value that is not completely opaque.

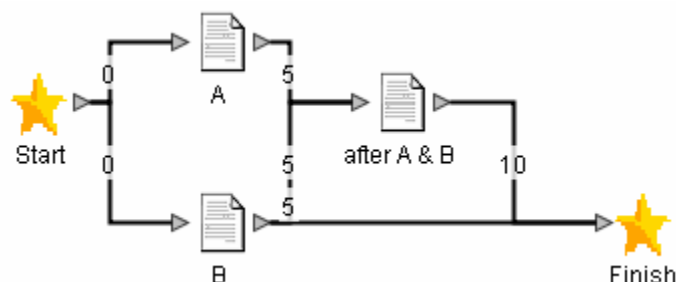
JGoSubGraph now extends the **JGoSubGraphBase** class. If you don't like the design of the **JGoSubGraph** class you can design and implement your own by inheriting from the **JGoSubGraphBase** class. **JGoSubGraphBase** is a minimal extension of **JGoNode** to support nested nodes and links. It does not implement support for collapsing/expanding, nor does it even assume there are drawn borders or a **Label** or a **Port**.

SimpleNode

A **SimpleNode** is slightly more complicated than a **JGoIconicNode**. It is designed for "flow"-like applications whose diagrams have a generally horizontal orientation. Instead

JGo User Guide

of a single port for the node, there are two distinct ones, one on each side. Often you can consider one as an “input” and the other as an “output”.



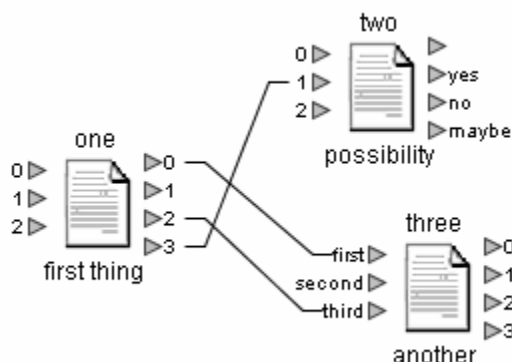
Although **SimpleNodes** are not normally resizable, if they are resized, only the icon is resized while maintaining its original aspect ratio. As with **JGoIconicNode**, the object passed to **initialize** is normally a **JGoImage**, but may actually be any **JGoObject**.

The label for a **SimpleNode** is normally editable by the user. You can turn this off by **aSimpleNode.getLabel().setEditable(false)**.

You can customize the appearance of the ports by setting **JGoPort** properties such as **PortStyle**, **Pen** and **Brush**, or **PortObject**.

GeneralNode

A **GeneralNode** is a generalization of a **SimpleNode**. It supports a variable number of ports on either side of the node. It also can have two labels for the whole node, at the top and at the bottom. Furthermore each port has its own label, to help identify the port to the user.



The **GeneralNodeLabels** and **GeneralNodePortLabels** are normally editable by the user. Each port, a **GeneralNodePort**, is indexed by position on its respective side of the node.

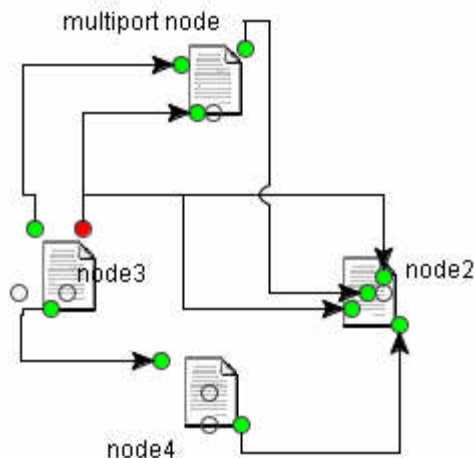
MultiPortNode

A **MultiPortNode** is just like a **JGoIconicNode**, but with a variable number of ports that can be positioned arbitrarily within the node. When you create a **MultiPortNode** you will need to add **JGoPorts** to the area and position them appropriately.

The user can interactively reposition the label (a **MultiPortNodeLabel**) relative to the icon. This is useful in allowing the user to position the label so that fewer links cross it. If you make this label not **Selectable**, the user will not be able to select and drag the label independently of the rest of the node.

The **MultiPortNodePort** class automatically changes its **Brush** upon a change in the number of links that are connected to the port. You may wish to modify or override the **linkChange** method to suit your application's needs.

MultiPortNodePort is also interesting in that it implements a **MaxLinks** property—you can specify the maximum number of links that the user can connect to the port. For demonstration purposes this value defaults to three, but you can set it to any value.

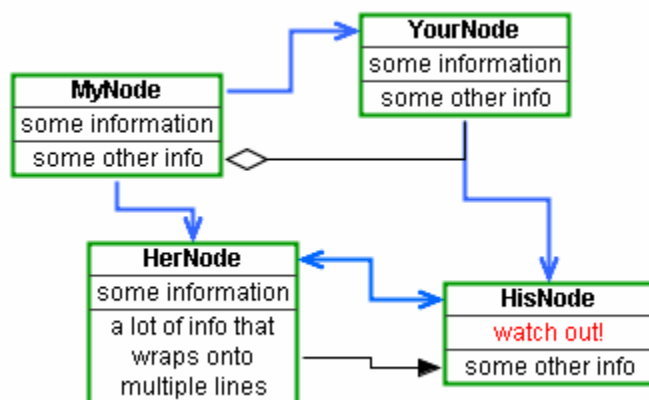


MultiTextNode

When you want to display several objects vertically in an area, perhaps separated by lines and with a drawable shape as the background, the **MultiTextNode** is appropriate to use. Each item in the node can have its own ports on each side, and there are ports at the top and at the bottom of the whole node.

As the name implies, it is typically used to display text strings. The **addString** method is a convenient way of setting up a **MultiTextNode**.

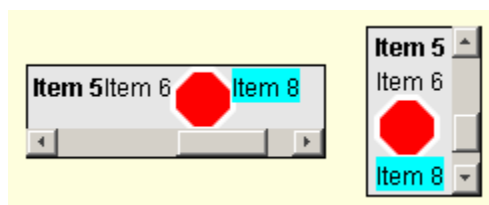
Each **MultiTextNode** has its own **ItemWidth** property, which is used to specify the width of all of the item objects. Calling **setItemWidth** automatically resets the width of all of the items present in the node. For convenience, if an item is an instance of **JGoText**, the method also sets the text object's **WrappingWidth**. However, if you do not want the text to wrap automatically, you can set the text's **Wrapping** property to false. Because the width of the text is then forced to be the node's **ItemWidth**, you will want to make sure the text's **Clipping** property is true, so that text on long lines doesn't spill over beyond the bounds of the node.



ListArea

A **ListArea** is similar to a **MultiTextNode** in that they both display an ordered list of objects. However, **ListArea** supports the use of scroll bars to be able to display many items within a relatively small area. Furthermore, **ListArea** is designed to be oriented horizontally as well as vertically, with the scroll bar on either side.

The following screen shot shows two **ListAreas**, one horizontal and one vertical, each showing three text objects and one polygon object. The text object containing the string “Item 8” is selected in both areas.



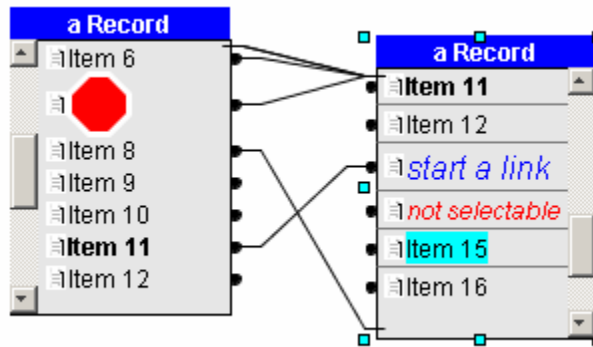
ListAreas can be resized, both programmatically and interactively. The minimum size is constrained to be large enough to show both the tallest item and the widest item.

Just as with **MultiTextNode**, each item can have two objects on each side, typically instances of **JGoPort**. But **ListArea** also supports a fourth object for each item, which is typically used for an icon (see the **RecordNode** screen shot, below).

You can specify the spacing between the items, the pen used to draw lines between the items, the rectangle used as the background, and the insets or margin around the list of items but inside the background. Unlike **MultiTextNode**, the item objects are not resized to fit to a particular width. In fact, the whole area will grow to accommodate an item that has grown in size.

RecordNode

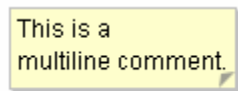
A **RecordNode** is an area that includes a **ListArea** and a header and a footer object. It assumes that each item object really is accompanied by an optional **JGoPort**, thus making this area a true node.



Of course all of the real flexibility comes from the **ListArea** class.

Comment

A **Comment** is a simple **JGoArea** that just has a **JGoText** object with a **JGo3DNoteRect** as a background object. As the size of the text changes, the bounds of the comment adjust correspondingly.



For convenience you can access the text string by the **Text** property. You can also change whether users can interactively edit the text in a comment by setting the **Editable** property.

General Concepts When Defining Nodes

You will probably want to add graphics that are specific to the real object the node represents. For example, if the node is a shop floor manufacturing machine, there might be a “Stopped” state that might change the appearance of the node so the operator could tell at a glance.

Another common addition is a property-editing dialog for each kind of node. Not all of the interesting information can or should be shown as **JGoObjects**; for example, additional status and a lot of controls for that shop floor manufacturing machine probably belong in a dialog.

When defining your own class derived from **JGoArea** or **JGoNode**, you may find it useful to examine the classes in the examples directory.

Depending on the desired functionality, there are several things that are commonly done in custom node classes:

- When adding settable fields, consider adding a new Changed hint and overriding **copyNewValueForRedo** and **changeValue** to handle updates with that new hint, to support undo and redo. (More information is available in the chapter about Undo and Redo.) You’ll also want to override **copyObject**.

JGo User Guide

- When some of your fields refer to children of the area, you will need to override **copyChildren** to make sure that each field is referring to the corresponding newly copied child object. And you should override **removeObjectAtPos** to check for when a child is removed from the area, to make the field reference invalid.
- If you want to support the standard persistence using SVG XML, be sure to override **SVGReadObject**, **SVGUpdateReference** and **SVGWriteObject** to read and write your class's attributes and references.
- To support drag-and-drop and cut/copy/paste, you should make sure your class is serializable. Fields that cannot be serialized (or that you don't want to be serialized, such as cached information) you should declare **transient** and make sure your code can reconstruct the needed information when the field is null in the copied object.
- You will need to override **layoutChildren** in order to reposition and/or resize some or all of the area's child objects when the area or some of its children change position or size. The **layoutChildren** method is called after the area is resized to do the actual work.
- You may wish to override **getLocation** and **setLocation**, if the natural position of the node isn't the top-left corner
- You may wish to override **setBoundingRect** to prevent the object from being moved to certain positions or from being sized to certain dimensions
- Similarly you may wish to override **handleResize** to constrain the user's interactive resizing of the object, if it is **Resizable**

Please examine the source code for the example nodes for a better understanding of the many features that JGo makes possible.

6. UNDO AND REDO

JGo makes it easy for programmers to build graphical applications that display relationships between objects and that allow users to change those relationships with little effort. Because users can make massive changes so easily, a well-designed application should also allow users to reverse the consequences of unintended changes.

Swing offers a standard framework, the **javax.swing.undo** package, for supporting undo and redo in any applet or application. JGo uses the standard mechanisms to implement support for undo and redo. But if you want to support undo and redo in your application, you will need to do five things:

- Signal any change to any application-specific document state.
- Perform the undo and redo for any such change.
- Set the undo manager for the document.
- Declare groups of changes that the user will want to consider a single logical edit.
- Implement the user-interface commands to allow users to perform an undo or a redo, with the appropriate appearance.

The built-in support for undo in JGo only applies to documents and document objects. Changes to views, such as selection and view position, are not considered to be edits to the document, and therefore are not tracked for undo and redo.

UndoableEdit and JGoDocumentChangedEdit

The basic concept is the **javax.swing.undo.UndoableEdit**, an interface that describes an object that represents a change to a document and the ability to undo and redo that change.

A change to a document means that some part of the document's state has been altered. This includes changing the values of any properties of a document, adding **JGoObjects** to a document, removing them, and changing any properties or parts of any document objects.

If you want to add undo and redo functionality to your application, you must make sure that your **JGoDocument** and **JGoObject** extensions faithfully signal any state changes by calling **JGoDocument.fireUpdate** or **JGoObject.update** respectively, and that your extensions correspondingly implement the **copyOldValueForUndo**, **copyNewValueForRedo**, and **changeValue** methods.

JGo User Guide

Not all document state need participate in this undo framework. However, you and your users must be willing to live with the inconsistencies that might result when the user makes a change and a later undo does not restore the state faithfully. You may find that some state currently associated with a document really belongs in the app or in the view.

Extending JGoDocument

The Flower example includes a representative document extension: adding a **name** property in the **ProcessDocument** class. The class, with parts elided for clarity, looks like the following code:

```
public class ProcessDocument extends JGoDocument
{
    public ProcessDocument()
    { // enable undo/redo memory for this document
        setUndoManager(new JGoUndoManager());
    }

    // Name property
    public String getName()
    {
        return myName;
    }

    public void setName(String newname)
    {
        String oldName = getName();
        if (!oldName.equals(newname)) {
            myName = newname;
            fireUpdate(NAME_CHANGED, 0, null, 0, oldName);
        }
    }

    // copy current state
    public void copyNewValueForRedo(JGoDocumentChangedEdit e)
    {
        switch (e.getHint()) {
            case NAME_CHANGED:
                e.setNewValue(getName());
                return;
            default:
                super.copyNewValueForRedo(e);
                return;
        }
    }
}
```

```

// actually perform the undo or redo
public void changeValue(JGoDocumentChangedEdit e, boolean undo)
{
    switch (e.getHint()) {
        case NAME_CHANGED:
            setName((String)e.getValue(undo));
            return;
        default:
            super.changeValue(e, undo);
            return;
    }
}

// Event hints
public static final int NAME_CHANGED = JGoDocumentEvent.LAST+1;

// State
private String myName = "";
}

```

The **getName** and **setName** methods of course define the **name** property for the document. What is noteworthy is that **setName** makes sure that there really is a change before setting the internal **myName** field and then calling **fireUpdate**.

The call to **fireUpdate** passes a hint, **NAME_CHANGED**, and the old value, held in **oldName**. It is important that the hint be unique within the class and all of its superclasses.

It is also required that the updating occur *after* the change has happened, and that the update event listener is able to retrieve the previous value. Normally the previous value is passed along as part of the document event. The reason for the requirement that the previous value be accessible is that the document listener responsible for undo/redo needs to record the values both before and after an edit. These values are used to construct a **JGoDocumentChangedEdit**, which implements **UndoableEdit**.

JGoDocumentChangedEdit gets the before and after values from the **JGoDocumentEvent** that is generated from a change's call to **fireUpdate**. In most cases the previous value is just fine; however if the value is a reference to an object that might be modified by further edits, it is important that the **JGoDocumentChangedEdit** keeps a true copy of the old value, rather than just a reference to something whose relevant state may have changed. Thus the **JGoDocumentChangedEdit** constructor calls the **copyOldValueForUndo** method, which allows the class to decide whether the previous value needs to be copied for safekeeping. Many classes do not have any kinds of changes where the previous value will need to be copied, so they do not bother to override **copyOldValueForUndo**.

JGoDocumentChangedEdit gets the new value by calling the **copyNewValueForRedo** method. Each class that extends the undoable document state must override this method to handle the class-specific changes to get the new (current) values. In the example

JGo User Guide

above, it just needs to remember the value of **getName()**. For change hints that don't belong to this class, the method should call the super method.

Finally, each class must override **changeValue** in order to perform the undo or redo, depending on the value of the boolean argument. For convenience the **JGoDocumentChangedEdit.getValue** method also takes the same **undo** parameter to decide whether to return the old/before value or the new/after value. In the example above, the method just needs to call **setName** to effect the change. Again, for change hints not belonging to this class, the method calls the super method.

For efficiency and for convenience the previous value of a **JGoDocumentEvent**, and the old and new values of a **JGoDocumentChangedEdit** are not simply **Objects**, but a pairing of an **int** and an **Object**. For those properties that can be represented efficiently by an **int**, you can use that instead of boxing the integer by creating an **Integer**. (For the common case of boolean values, boolean versions of these methods are provided by **JGoDocumentChangedEdit**.) For those properties that can be conveniently represented by an integer and an object (for example a change to an element of a vector), you can use both.

Extending JGoObject subclasses

For a change to an object, the hint is **JGoDocumentEvent.CHANGED**. However, there is no way for the document to know how to remember the old or new values for any particular sub-hint, nor how to perform that particular state transition. Instead those responsibilities are transferred to **JGoObject**, which has the same **copyOldValueForUndo**, **copyNewValueForRedo**, and **changeValue** methods.

The implementation is very similar to that for adding properties to documents. What follows is the definition of the **ListArea** example class, stripped down to essentials regarding the **insets** property.

```
public class ListArea extends JGoArea
{
    public JGoObject copyObject(JGoCopyEnvironment env)
    {
        ListArea newobj = (ListArea)super.copyObject(env);
        if (newobj != null) {
            . . .
            newobj.myInsets.top = myInsets.top;
            newobj.myInsets.left = myInsets.left;
            newobj.myInsets.bottom = myInsets.bottom;
            newobj.myInsets.right = myInsets.right;
        }
        return newobj;
    }
    // extra space around the edges
    // the space should include room for the scroll bar
    public Insets getInsets()
    {
```

```

        return myInsets;
    }

    public void setInsets(Insets x)
    {
        Insets s = getInsets();
        if (!s.equals(x)) {
            Insets oldInsets=new Insets(s.top,s.left,s.bottom,s.right);
            myInsets.top = x.top;
            myInsets.left = x.left;
            myInsets.bottom = x.bottom;
            myInsets.right = x.right;
            update(InsetsChanged, 0, oldInsets);
            layoutChildren();
        }
    }

    public void copyNewValueForRedo(JGoDocumentChangedEdit e)
    {
        switch (e.getFlags()) {
            . . .
            case InsetsChanged: {
                // copy value so it doesn't get clobbered later
                Insets s = getInsets();
                e.setNewValue(new Insets(s.top,s.left,s.bottom,s.right));
                return; }
            default:
                super.copyNewValueForRedo(e);
                return;
        }
    }

    public void changeValue(JGoDocumentChangedEdit e, boolean undo)
    {
        switch (e.getFlags()) {
            . . .
            case InsetsChanged:
                setInsets((Insets)e.getValue(undo));
                return;
            default:
                super.changeValue(e, undo);
                return;
        }
    }
}

```

JGo User Guide

```
// Event hints
public static final int InsetsChanged = JGoDocumentEvent.LAST +
10033;

// State
private Insets myInsets = new Insets(1, 4, 1, myBarSize + 4);
}
```

Note that the **setInsets** method makes a copy of the old value before remembering the new value (also by copying, just in case the argument **Insets** value were to be modified independently). This copy is needed in order to pass the previous value to the **JGoObject.update** method.

Remember that the properties should also be copied in the **copyObject** method.

Handling Big Changes

Keeping track of all these edits is simple enough, but incurs a lot of overhead for detecting the change and constructing the edit. What should you do when you know you might be making a lot of changes and don't want the repeated overhead?

In the past the only such mechanism was to suspend updates. Calling **setSuspendUpdates(true)** would turn off all event notification. After all of the batched changes were done, you would call **setSuspendUpdates(false)** to re-enable event notification, and listeners would have to assume anything and everything had possibly changed. This was true both at the **JGoDocument** level as well as the **JGoObject** level.

Suspending updates is still possible, but with the introduction of undo managers, this is more complicated. The problem is that implementing undo requires getting the state *before* the changes. Turning off event notification means that there's no way to keep track of any changes that are going on. Trying to save all state at the time of the call to **setSuspendUpdates(true)** would be horribly inefficient, particularly for documents. Instead we need to save very targeted state, depending on the kinds of changes that are expected to occur during the update suspension.

The mechanism that JGo supports is analogous to the **fireUpdate/update** mechanism used for notification after a change. The **fireForedate/foredate** methods are exactly like **fireUpdate/update** except they should be called just before a change. For obvious reasons, the foredate methods don't need any previous value parameters.

Here is an example of how foredating can be done:

```
// care about undo, so need to call fireForedate here,
// so that the before-layout geometries of all top-level
// objects can be remembered
fireForedate(JGoDocumentEvent.ARRANGED, 0, null);
setSuspendUpdates(true);
layoutWholeDiagram();
setSuspendUpdates(false);
// care about undo/redo, so need to call fireUpdate here;
// don't need to pass previous arrangement here
```



```
fireUpdate(JGoDocumentEvent.ARRANGED, 0, null, 0, null);
```

The foredate methods create **JGoDocumentEvents** whose **isBeforeChanging** predicate returns **true**. Listeners that don't care about notification before a change should ignore these events; for example, **JGoView** ignores these events. But **JGoUndoManager**, described below, uses them to remember the state before events.

The **copyOldValueForUndo** method, when invoked for the ARRANGED update, is responsible for getting the old/previous state. Since that state is not passed in via the previous value parameters, it must get it from the edit produced by the foredate event. It can do that by calling the **findBeforeChangingEdit** method on **JGoDocumentChangedEdit**.

```
public void copyOldValueForUndo(JGoDocumentChangedEdit e)
{
    switch (e.getHint()) {
        . . .
        case JGoDocumentEvent.ARRANGED:
            // For an update, there's no previous value info passed in.
            // However, that information is instead available in the
            // earlier JGoDocumentChangedEdit created by the foredate.
            // In the after/update case, we want to move the previous
            // state information from the BeforeChanging Edit to this
            // Edit.
            if (!e.isBeforeChanging()) {
                JGoDocumentChangedEdit before =
                    e.findBeforeChangingEdit();
                if (before != null) {
                    e.setOldValue(before.getNewValue());
                }
            }
            return;
    }
}
```

The **copyNewValueForRedo** and **changeValue** methods are implemented normally for the ARRANGED case. The **copyNewValueForRedo** method copies all the current node and link geometries into a vector. The **changeValue** method sets all the node and link geometries given the information in a vector.

JGoUndoManager, CompoundEdits and Transactions

The edits implemented by **JGoDocumentChangedEdit** are very detailed, specific changes that can be undone and redone. But when a user drags a selection, the user is changing the positions of possibly thousands of objects. Clearly the user will not expect that an undo command only move one of those objects back to its earlier location.

The Swing package offers the **CompoundEdit** class for keeping track of an ordered list of **UndoableEdits**. Each compound edit is composed of all the edits that occur due to a

JGo User Guide

particular user gesture or command. The compound edits in turn are managed by the undo manager.

JGoUndoManager is an extension of **javax.swing.undo.UndoManager**. It implements **JGoDocumentListener** so that it can detect all of the changes that happen to a document, and then record them by producing and collecting **JGoDocumentChangedEdits** in the **JGoUndoManager**'s current **CompoundEdit**.

To control when a compound edit is finished and another one should be started, **JGoDocuments** support the notion of a transaction. Call **startTransaction** before any changes occur and call **endTransaction** afterwards. The first detected document change will open up a new compound edit. All succeeding edits are added to this current compound edit. A call to **endTransaction** will close up the current compound edit and add it to the undo manager's list of undoable edits.

Generally it is the view that is naturally responsible for detecting the start of a user action or command and knowing when it is finished. Thus the default implementations of many commands in **JGoView** start and end transactions. These methods include:

- **copy** (start and end)
- **cut** (start and end)
- **paste** (start and end)
- **drop** (start and end)
- **doMoveSelection** (start and end)
- **deleteSelection** (start and end)
- **startNewLink** and **startReLink** (start)
- **newLink**, **noNewLink**, **reLink**, and **noReLink** (end)
- **startResizing** (start)
- **handleResizing** (end)
- **doCancelMouse** and other cancel methods (end)

In addition, some methods such as **JGoText.doStartEdit** and **doEndEdit** enclose editing activity within a transaction. However, any code anywhere can start and end transactions on a document. When you add your own commands to your application, you will probably want to wrap any document changing code with a transaction.

Transactions may be nested (e.g. start, start, end, end). Only the final transaction end causes the compound edit to be closed and added to the undo manager's list. Beware calling **startTransaction** without a corresponding call to **endTransaction**, perhaps due to an exception.

A call to **endTransaction** requires a **String** argument that describes that particular transaction to the user. This is the "presentation name". **JGoUndoManager** provides default presentation names for the predefined transactions. These are the only strings in the **JGo** package that should be localized for international applications.

Each document that supports undo must have a **JGoUndoManager**. Normally each document will have its own undo manager, but when there are interrelated documents where one change affects other documents, you may want to share one undo manager amongst several documents. Calling **JGoDocument.setUndoManager** automatically makes the manager a listener on that document.

A call of **endTransaction(false)** will discard the current compound edit, rather than adding it to the undo manager. Unlike a transactional database system, aborting a transaction in JGo does *not* automatically undo all of the changes that may have happened since the transaction start. This is because there might not be an undo manager, or because not all changes are being recorded.

Another difference between transactions with JGo documents and database systems is that there is no prohibition on examining or even modifying documents or their objects without a preceding call to **startTransaction**. There is no practical way to enforce the prohibition of reading the data structures.

Defining Menu Commands

JGoUndoManager provides implementations of **undo**, **redo**, **canUndo**, **canRedo**, and **discardAllEdits**, that user interface implementations should call.

JGoDocument provides these same methods by delegating to the document's undo manager, if one exists.

The following code is taken from the Flower example. Adding user-interface support for undo entails calling **canUndo** to enable/disable the command and calling **undo** to perform the action. In addition, you may wish to customize the menu item text with the presentation name.

```
JMenuItem UndoMenuItem = null;

AppAction UndoAction = new AppAction("Undo", this) {
    public void actionPerformed(ActionEvent e)
    {
        getView().getDocument().undo();
        AppAction.updateAllActions();
    }
    public boolean canAct()
    {
        return super.canAct() &&
            (getView().getDocument().canUndo());
    }
    public void updateEnabled()
    {
        super.updateEnabled();
        if (UndoMenuItem != null && getView() != null)
            UndoMenuItem.setText(
```

JGo User Guide

```
        getView().getDocument().  
        getUndoManager().getUndoPresentationName());  
    }  
};
```

This code calls **getView()** to get the currently open child window.

7. PERFORMANCE HINTS

When there are only tens or hundreds of objects in a document, performance is rarely a problem. However, when dealing with many hundreds or thousands of objects, the programmer should be aware of performance issues.

Don't add an area (node) to the document until the last possible moment—as objects are added to the area and as they are modified, no document listeners will be notified until after the area is added to the document.

Another way to avoid a lot of updates temporarily is to use

JGoObject.setSuspendUpdates(true) or **JGoDocument.setSuspendUpdates(true)**.

These calls can temporarily avoid notifying listeners about change events, for an individual object or for a whole document, respectively. Be sure to re-enable listener notification by calling the method again with a **false** value.

Support for undo and redo slows down editing because the undo manager must listen for document events and construct edits for each change. By default a document does not have an undo manager, so you should call **JGoDocument.setUndoManager** only when needed. Alternatively you can set the document's **SkipsUndoManager** property, assuming it is not confusing for the user to do so.

Those undo edits can take up a lot of memory. Depending on your application design, sometimes you may wish to call **discardAllEdits** to save on virtual memory occupied by all of the edits. This is commonly done when the document is saved. You can also change how much is saved by overriding **skipEvent** on **JGoUndoManager**, or by calling **setLimit**.

Try to avoid allocating many **Points**, **Dimensions**, and **Rectangles** that just get thrown away. For example, to get an object's X position, call **getLeft()** instead of **getTopLeft().x**.

Interactively dragging many objects together can be sluggish if there are a lot of links connected to the nodes being moved. This is particularly true when the calculation of new strokes for all those links is expensive, such as when the **JGoLink** properties **AvoidsNodes** and **JumpsOver** and **Orthogonal** are true. It can also be true for very complex nodes, such as **RecordNodes**. You can avoid this continuous overhead during dragging by setting the **JGoView** property **DragsRealtime** to false. The move, and thus the recomputation of all attached links, is only done when the user finishes the drag.

Don't use Sun's JVM/JRE 1.4.0 on Windows. See the ...README.txt file for details. We recommend using 1.4.1 or later instead.

8. JGO SUPPORT FOR XML AND SVG

JGo provides support for Extensible Meta-Language (XML) and Scalable Vector Graphics (SVG) in several ways:

- Write-only support of SVG using the Batik libraries and **com.nwoods.jgo.examples.SVGGoView**
- Write-only support of SVG using the JAXP libraries and **com.nwoods.jgo.svg**
- Read/write support of JGo XML using the JAXP libraries and **com.nwoods.jgo.svg**
- Read/write support of extended SVG (SVG with JGo XML extensions) using the JAXP libraries and **com.nwoods.jgo.svg**
- Read/write support of your own custom XML format using the JAXP libraries as demonstrated in the **com.nwoods.jgo.example.Flower** sample application

Each of the above approaches is demonstrated in the **com.nwoods.jgo.Flower** sample application and each will be discussed further in the following sections.

SVG Support using Batik and SVGGoView

The generation of SVG using **com.nwoods.jgo.examples.SVGGoView** relies on the Batik library to reproduce everything drawn to the **Graphics2D** object associated with a **JGoView** as SVG. This technique captures all graphical output at a very low level.

The advantages of this technique include its simplicity and visual accuracy.

The disadvantages of this technique include the inability to read the generated SVG to reproduce the original **JGoDocument**, and inability to extend the generated SVG to easily include your own elements, attributes, scripts, etc.

To create an SVG document using **SVGGoView**, simply create a new instance of **SVGGoView** and set its **Document** property to the **JGoDocument** to be generated as SVG. Then call the **generateSVG** method to perform the output. The following method from the **ProcessDocument** class in the Flower sample application illustrates this process:

```
public void storeSVG2(OutputStream outs)
    throws IOException, UnsupportedOperationException
{
    SVGGoView svgView = new SVGGoView();
    svgView.setDocument(this);
}
```

```

        svgView.generateSVG(outs);
    }

```

If your principal requirement is the write-only creation of SVG documents from JGo for viewing by other applications or browsers, **SVGGGoView** may be your best solution.

XML and SVG Support Using JAXP and the JGo SVG Package

The generation of SVG using the JGo SVG package (**com.nwoods.jgo.svg**) uses the JGo infrastructure to generate SVG and XML elements closely associated with high level JGo object classes.

The advantages of this technique include its extensibility and its ability to allow output files to be read back into JGo to precisely reproduce the original **JGoDocuments**.

The disadvantages of this technique include a small additional complexity and a slightly less accurate rendering of the **JGoDocument** as SVG.

The Java API for XML Parsing (JAXP) version 1.2 or later, available from Sun Microsystems (java.sun.com), is a prerequisite for the use of the JGo SVG Package. The libraries are built into the 1.4 and later releases.

To create an SVG document using the JGo SVG package, simply create a new instance of **DefaultDocument** and call the **SVGWriteDoc** method to perform the output. The following method from the **ProcessDocument** class in the Flower sample application illustrates this process:

```

public void storeSVG1(OutputStream outs,
                     boolean genXMLExtensions,
                     boolean genSVG)
{
    DefaultDocument svgDomDoc = new DefaultDocument();
    svgDomDoc.setGenerateJGoXML(genXMLExtensions);
    svgDomDoc.setGenerateSVG(genSVG);
    svgDomDoc.SVGWriteDoc(outs, this);
}

```

The two properties (**GenerateJGoXML** and **GenerateSVG**) shown in the above example control the classes of elements generated by the JGo SVG package. JGo XML elements are used to describe the JGo classes and their properties. SVG elements are used to render the JGo graphical elements as SVG. When using the JGo SVG package, you can determine whether to generate either or both of these element classes by specifying boolean values for these properties. If only **isGenerateJGoXML()** is true, an XML document will be generated that faithfully serializes a **JGoDocument** for read/write purposes, but no SVG elements will be generated and the document will not be viewable by an SVG viewer. If only **isGenerateSVG()** is true, an SVG document will be generated that allows the **JGoDocument** to be viewed by SVG viewers, but the document will not be able to be read back into JGo to recreate the original **JGoDocument**. If both **isGenerateJGoXML()** and **isGenerateSVG()** are true, an SVG document will be created which is both viewable by SVG viewers and is able to be read back into JGo to reproduce the original **JGoDocument**.

As mentioned earlier, one of the principal advantages of this technique is the ability to extend both the generated SVG and the JGo XML extensions used for serialization purposes. In order to create these extensions, you typically need only to override the **SVGWriteDoc** and **SVGReadObject** methods on any subclass of **JGoObject** you create. When **DefaultDocument.SVGWriteDoc** is invoked, the **SVGWriteObject** method is automatically invoked on the **JGoDocument** and each of the **JGoObjects** contained in that document. By overriding the **SVGWriteObject** method, you can easily add your own information. Similarly, when reading an SVG or XML file via **DefaultDocument.SVGReadDoc**, you can override **SVGReadObject** to look for your extensions and recreate the information in your own objects.

The **SVGReadObject** and **SVGWriteObject** methods deal with **DomDoc** and **DomElement** parameters. **DomDoc** is an interface similar to **org.w3c.dom.Document**. **DomElement** is an interface similar to **org.w3c.dom.Element**. These interfaces allows the **com.nwoods.jgo** package to provide methods that manipulate **org.w3c.dom** objects while not requiring the **org.w3c.dom** package to be present in order to build or use **com.nwoods.jgo**. Methods such as **DomDoc.createElement**, **DomElement.setAttributes** and **DomElement.appendChild** allow you to easily create new XML or SVG elements and attributes and add them to your document.

The implementation of these interfaces is provided by the **com.nwoods.jgo.svg** package. The default implementation of **com.nwoods.jgo.DomDoc** is **com.nwoods.jgo.svg.DefaultDocument**. The default implementation of **com.nwoods.jgo.DomElement** is **com.nwoods.jgo.DefaultElement**. You should be able to accomplish whatever you need through methods of the **DomDoc** and **DomElement** interfaces, however if you require access to the **org.w3c.com** objects themselves, that can be accomplished by casting your **DomDoc** or **DomElement** object to **DefaultDocument** or **DefaultElement** and then invoking the **DefaultDocument.getDocument** or **DefaultElement.getElement** methods to get the actual **org.w3c.dom** objects.

The following code shows the implementation of **JGoRectangle.SVGWriteObject** which is responsible for generating both the XML and SVG elements necessary to represent a **JGoRectangle** object.

```
public void SVGWriteObject(DomDoc svgDoc,
                           DomElement jGoElementGroup)
{
    // Add JGoRect element
    if (svgDoc.JGoXMLOutputEnabled()) {
        DomElement jGoRect = svgDoc.createJGoClassElement(
            "com.nwoods.jgo.JGoRectangle", jGoElementGroup);
    }
    // Add SVG rect element
    if (svgDoc.SVGOutputEnabled()) {
        DomElement element =
            (DomElement)svgDoc.createElement("rect");
        // Add attributes to SVG <rect> element
```



```

        SVGWriteAttributes(element);
        jGoElementGroup.appendChild(element);
    }

    // Have superclass add to the JGoObject group
    super.SVGWriteObject(svgDoc, jGoElementGroup);
}

```

Note the use of the convenience method **DomDoc.createJGoClassElement**. This method will create a **DomElement** with the tag “JGoClass” and a “class” attribute specifying the class name and append it as the next child node of the specified **DomElement**. The class name supplied must be accurate and complete as it will be used to create an object of the correct type when the SVG XML file is read back in. Although the example shown above does not apply any attributes to the created <JGoClass> element, attributes could easily have been added via the **DomElement.setAttribute** method. Finally, note that this method calls its superclass so that the superclass can add its own elements and attributes.

When reading an SVG or XML file using **DefaultDocument.SVGReadDoc** method, any <JGoClass> element encountered will be automatically recognized by the **DefaultDocument.SVGReadElement** method. The class attribute of this element will be read and used to create a new instance of this object class from the class name. The **SVGReadObject** method will then be invoked on the newly created instance. Attributes of this element should be read by the **DomElement.getAttribute** method. Finally, this method should call its superclass so that the superclass can read its own elements and attributes.

Typically, the attribute values specified in **SVGReadObject** and **SVGWriteObject** can be stored as String values. Occasionally, however, the attribute value may need to be a reference to another **JGoObject** specified in the SVG or XML output. This can be difficult due to the fact that the referenced **JGoObject** may or may not have yet been written out to the XML or SVG document when the referencing object is written. The **DomDoc.registerReferencingNode** method has been created as a convenience for this situation. In **SVGWriteObject** when writing such an attribute value, call **registerReferencingNode** specifying the referencing **DomElement**, the **Object** being referenced, and the attribute name to use to hold the reference. The **DomDoc** will maintain a table of these references and update the **DomElements** after all the objects have been created in the **DomDoc** but before **DomDoc** has been rendered as SVG or XML.

When reading an object reference attribute in **SVGReadObject**, simply call **DomDoc.registerReferencingObject** specifying the referencing **Object** and the name and value of reference attribute. The **DomDoc** will maintain a table of these references and will invoke **JGoObject.SVGUpdateReference** passing a string identifying the reference attribute name and the referenced **Object** once all objects have been created. You must override **SVGUpdateReference** if your subclass has reference to other **Objects** that are to be saved and restored from SVG or XML.

JGo User Guide

The following example uses a **JGoRectangle** to illustrate the format of a generated SVG **JGoObject**. Note that all of the output shown below is automatically generated by JGo. Your application need only be concerned with your own extensions.

```
<g>
<JGoClass class="com.nwoods.jgo.JGoRectangle">
<rect height="75" style="stroke:black;stroke-
width:1;fill:rgb(255,0,0);" width="75" x="65" y="71"/>
<JGoClass class="com.nwoods.jgo.JGoDrawable"
drawablebrush="jgoid1" drawablepen="jgoid2"
embeddedpenbrush="false"/>
<JGoClass class="com.nwoods.jgo.JGoObject" obj_flags="1054"/>
</g>
```

Note that the entire **JGoObject** is enclosed in a group (<g>). Each subclass of the **JGoObject** is described by a <JGoClass> element, starting with the most specific class and moving to the more general. Each <JGoClass> element has a "class" attribute that defines the class name. Each <JGoClass> element may also have several other attributes that uniquely describe that state of that class. Following the <JGoClass> element, each class may also generate any other elements that are required, including representations of contained objects and standard SVG elements such as the <rect> element shown in the above example. The information contained in the <JGoClass> elements allows us to accurately save and restore all the information in a particular **JGoObject** subclass.

For a working example of using the JGo SVG package to read and write XML or SVG files, refer to the com.nwoods.jgo.examples.Flower sample application. Look for sections of code in **ProcessDocument** commented with /*SVG ...*/. Remove the comments to activate the contained code sections and rebuild the Flower sample application. The “file/save as” and “file/open” menu items illustrate saving and restoring the application data in a variety of formats.

For a formal description of the XML elements and attributes written by the JGo SVG package, refer to the file com.nwoods.jgo.svg/xsvg.dtd.

Custom XML Support Using JAXP

The generation of your own custom XML using only the Java API for XML Parsing (JAXP) available from Sun Microsystems (java.sun.com) is another viable alternative.

The advantages of this technique include its conciseness, as well as the ability to exercise complete control over the XML content.

The disadvantages of this technique include the difficulty of generating all your own SVG elements (if SVG output is necessary) and a slight additional complexity.

It is often unnecessary to save all the state information of every object in JGo (as is done by the JGo SVG package). Typically, one needs only to save enough information to allow your application to recreate the JGo objects and other application information.

To create an XML document using JAXP, you must first create an **org.w3c.Document** object. Typically, you would next traverse all of the top-level **JGoObjects** in your **JGoDocument** and populate the **Document** with **org.w3c.Element** objects, with appropriate properties set on these objects to represent those **JGoObjects**. Finally, you

would create a **javax.xml.transform.Transformer** object to transform the **Document** to XML output. The following method from the **ProcessDocument** class in the Flower sample application illustrates this process:

```
public void storeXML(OutputStream outs)
    throws IOException, UnsupportedOperationException
{
    Document document = null;
    try {
        DocumentBuilderFactory factory =
            DocumentBuilderFactory.newInstance();
        DocumentBuilder builder = factory.newDocumentBuilder();
        document = builder.newDocument();

        Element process =
            (Element)document.createElement(processTag);
        process.setAttribute("name", getName());
        process.setAttribute("location", getLocation());
        process.setAttribute("lastnodeid",
            Integer.toString(myLastNodeID));
        process.setAttribute("ortholinks", isOrthogonalFlows()
            ? "1" : "0");
        document.appendChild(process);

        // first produce all of the nodes
        JGoListPosition pos = getFirstObjectPos();
        while (pos != null) {
            JGoObject obj = getObjectAtPos(pos);
            pos = getNextObjectPosAtTop(pos);

            if (obj instanceof ActivityNode) {
                ActivityNode node = (ActivityNode)obj;
                Element act = document.createElement(activityTag);
                act.setAttribute("id", Integer.toString(node.getID()));
                act.setAttribute("type",
                    Integer.toString(node.getActivityType()));
                act.setAttribute("x",
                    Integer.toString(node.getLeft()));
                act.setAttribute("y", Integer.toString(node.getTop()));
                act.setAttribute("text", node.getText());
                process.appendChild(act);
            }
        }

        // then produce all of the links
    }
}
```

```

pos = getFirstObjectPos();
while (pos != null) {
    JGoObject obj = getObjectAtPos(pos);
    pos = getNextObjectPosAtTop(pos);

    if (obj instanceof FlowLink) {
        FlowLink link = (FlowLink)obj;
        Element flow = document.createElement(flowTag);
        flow.setAttribute("from",
            Integer.toString(link.getFromNode().getID()));
        flow.setAttribute("to",
            Integer.toString(link.getToNode().getID()));
        flow.setAttribute("text", link.getText());
        process.appendChild(flow);
    }
}
} catch (ParserConfigurationException pce) {
    // Parser with specified options can't be built
    pce.printStackTrace();
}
}
if (document != null) {
    try {
        TransformerFactory transformerFactory =
            TransformerFactory.newInstance();
        Transformer serializer =
            transformerFactory.newTransformer();
        serializer.setOutputProperty(OutputKeys.METHOD, "xml");
        serializer.setOutputProperty(OutputKeys.INDENT, "yes");
        serializer.transform(new DOMSource(document),
            new StreamResult(out));
    } catch (Exception x) {
        x.printStackTrace();
    }
}
}
}

```

A similar set of operation are required to read back in the generated custom XML. Refer to the **ProcessDocument.loadXML** method for a detailed example.

The **com.nwoods.jgo.examples.Flower** sample application demonstrates this technique (as well as all the other techniques described in this chapter). Look for sections of code in **ProcessDocument** commented with `/*XML ...*/`. Remove the comments to activate the contained code sections and rebuild the Flower sample application. The “file/save as” and “file/open” menu items illustrate saving and restoring the application data in a variety of formats.

9. BUILDING A SAMPLE APPLICATION USING JGO BEANS

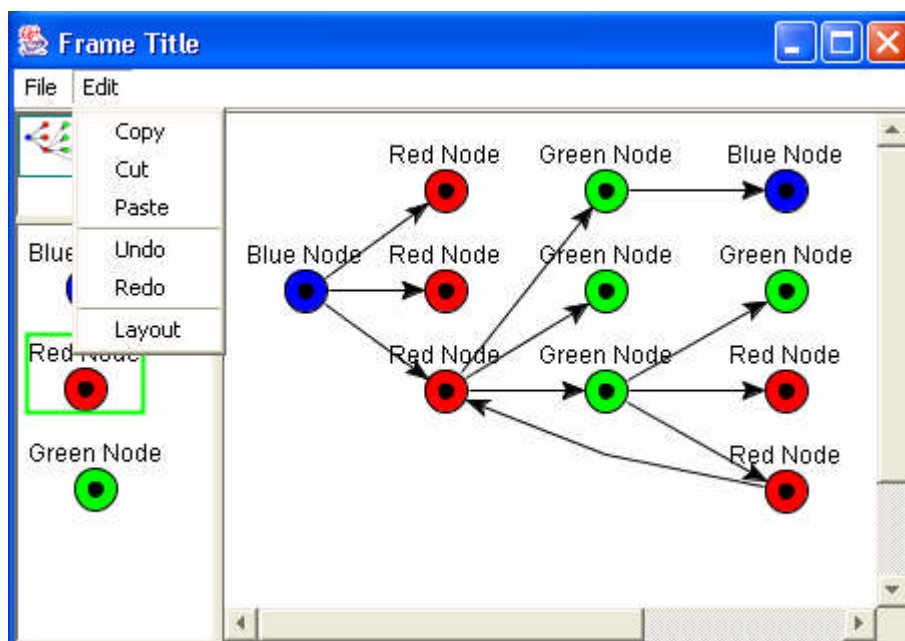
Building a JGo application using a standard Java development environment and the JGo Java beans (**JGoView**, **JGoPalette**, and **JGoOverview**) is quite simple. Although the precise steps required differ according to your specific development environment, the general concepts remain the same:

- Register the JGo beans with the development environment
- Create a new project
- Visually edit the user interface as required
 - Drag & drop **JGoView**, **JGoPalette**, and/or **JGoOverview** beans to your user interface
 - Edit bean properties
- Add event handlers to customize user interface behavior
 - Add JGo **documentChanged** listeners
 - Add JGo **viewChanged** event listeners

In this chapter we will describe the general process of creating a simple JGo application. Specific examples will refer to the Borland JBuilder development environment although the general concepts should apply equally well to any Java development environment with support for Java beans.

The sample application will provide a palette of objects (**JGoPalette**), including JGo sample objects as well as objects customized according to the needs of our application. The sample will support drag and drop of these objects from the palette to a scrolled, scalable window (**JGoView**). A third pane of the application will provide a miniature overview window showing the entire canvas and the region currently visible in the **JGoView** window. The application will provide cut, copy, and paste clipboard support, undo and redo, serialization to and from an extended Scalable Vector Graphics (SVG) XML format file, and automatic layout of the graph produced by our users. Double-clicking on any item in the **JGoView** window will cause a dialog to appear displaying more detailed properties for that object. Naturally, this is only one instance of the kinds of applications that can be created using JGo and the JGo beans, but hopefully this one instance will provide some insight into development commonly done for many JGo applications.

An example use of the finished application follows:



Register the JGo Beans with the Development Environment

We start by registering the JGo beans with the development environment. To register beans with a development environment we must first define the library containing the beans. JGo provides three different bean classes, **JGoView**, **JGoPalette**, and **JGoOverview**. All three of these beans are defined in the **com.nwoods.jgo** package and are packaged in the **JGo.jar** file. In addition, this particular sample application will utilize the **com.nwoods.jgo.layout** and **com.nwoods.jgo.svg** packages. These packages do not define any additional beans, but provide the auto-layout and XML/SVG serialization capabilities demonstrated in the later steps of the sample application. These packages are packaged in the **JGoLayout.jar** and **JGoSVG.jar** files, respectively. In JBuilder, libraries are defined under the “tools, configure libraries...” menu entry.

Once the library has been defined, the beans it contains can be placed on one of the development environment’s component palettes for use in the future visual construction and editing of user interfaces. Java beans can be identified to a Java development environment by being defined in the jar manifest file, by having **BeanInfo** classes associated with them, or simply by selecting them by their class name. All the JGo bean classes can be identified by any of the above means. The simplest approach is to simply add all the beans defined in the **JGo.jar** manifest file. This will result in the three JGo bean classes, and only those classes, being added to the component palette. In JBuilder, component palettes are defined under the “tools, configure palette...” menu entry.

Visually Construct the User Interface

Once the JGo beans have been added to the Java development environment’s component palette, we can begin creating our JGo sample application. Our application may be a Java application or an applet. JGo can also be used in Java servlets, but servlet creation

Building a Sample Application Using JGo Beans

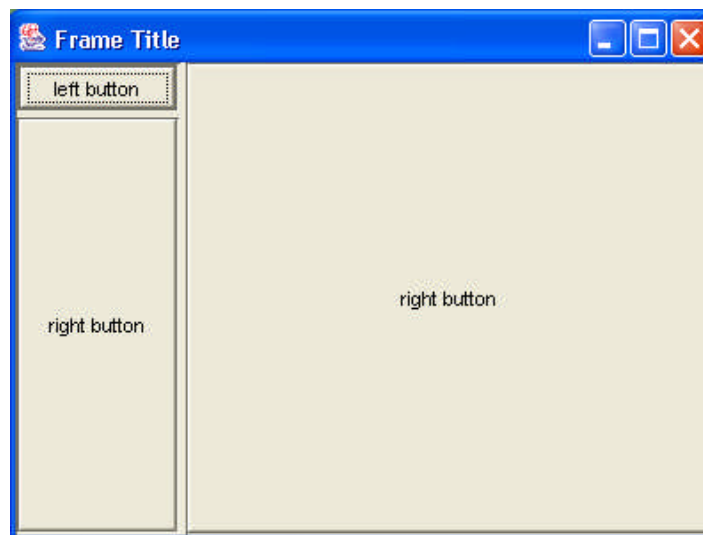
is not discussed in this section. Refer to the `com.nwoods.jgo.examples.imager` sample application for an example of JGo use in a servlet.

Java applications will contain one or more top-level containers, typically either **JFrames** or **JApplets**. A hierarchy of visual components will be added to these top-level containers to create the user interface that our end-users will see. Because user interfaces are inherently visual, most Java development environments enable the developer to construct the user interface visually rather than by directly writing code. Typically, the components of the interface are dragged from component palette and dropped on the user interface under construction.

To build our sample application we will create a new project with a single **JFrame** component. In JBuilder, this is accomplished by first selecting the “File/New Project...” menu entry and then selecting the “File/New...” menu entry and selecting “Application” as the item to create. A subclass of **JFrame** named **Frame1** is created. Select **Frame1** for editing and click on the Design tab to enable visual editing. Make sure that the “layout” property for the **JFrame contentPane** is set to be an instance of **java.awt.BorderLayout**.

We now divide the top-level frame into two separate panes by dragging and dropping a **JSplitPane** onto the very center of the top-level **JFrame**. In JBuilder, the default name of this **JSplitPane** is `jSplitPane1`. Make sure that the `jSplitPane1` is added at the **BorderLayout.CENTER** location of **Frame1**.

Next, we sub-divide the left pane of the **JSplitPane** into two panes by dragging and dropping another **JSplitPane** onto the very center of the leftmost **JSplitPane**. In JBuilder, the default name of this **JSplit** pane is `jSplitPane2`. Make sure this **JSplitPane** has been added to `jSplitPane1` at the **JSplitPane.LEFT** location. Modify the “orientation” property of this **JSplitPane** to be **JSplitPane.VERTICAL_SPLIT**. The resulting window should appear as follows:



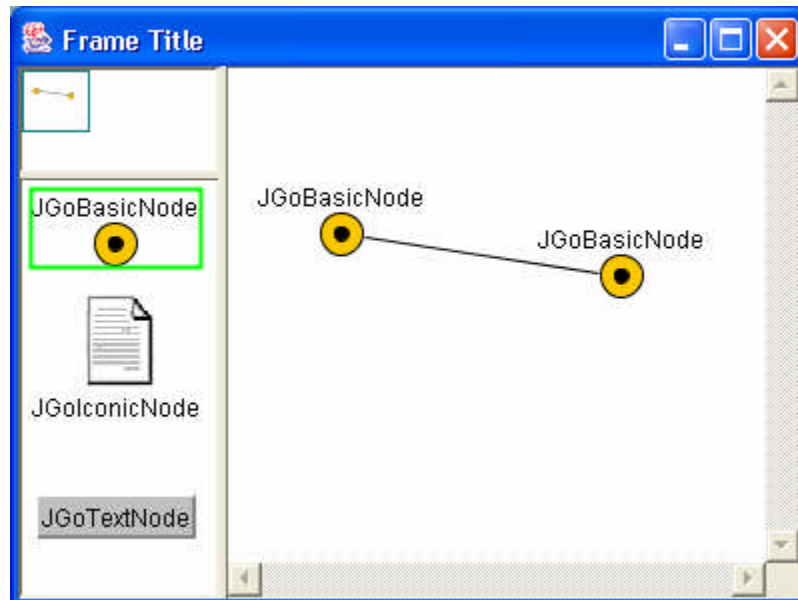
JGo User Guide

Next, we will drag and drop the JGo beans into these panes. Place the **JGoOverview** bean in the upper-left pane. Place the **JGoPalette** bean in the lower-left pane. Place the **JGoView** bean in the right pane. Verify that the location of the **JGoView** bean is **JSplitPane.RIGHT** in **jSplitPane1**, the location of the **JGoOverview** bean is **JSplitPane.TOP** in **jSplitPane2**, and the location of the **JGoPalette** bean is **JSplitPane.BOTTOM** in **jSplitPane2**.

Modify the “showSampleItems” property of the **JGoPalette** to be “true”. This will cause a small variety of sample nodes to be shown in the **JGoPalette**.

Modify the “observed” property of the **JGoOverview** to be the name of the **JGoView** component. If you are using JBuilder, the default name given to the **JGoView** is “jGoView1”. This will cause the **JGoOverview** window to display a miniaturized overview of the **jGoView**.

Build and run the program. Drag and drop two **JGoBasicNodes** onto the **JGoView**. Create a new link by dragging a link from the port of one **JGoBasicNode** to the port of the other. The results should appear as follows:



At this point we have already created a simple application illustrating some of the default behaviors supported by JGo, including drag and drop, selection, and multiple selection. Nodes can be created by dragging from the palette and dropping onto the main view, or by control-copying selected nodes. Links can be created by dragging a link between ports on the objects.

Add Event Listeners

By adding event listeners, we can further modify the default behavior of the application and react to end-user interaction.

We'll begin by adding a **documentChanged** event listener on the **JGoView**. This event listener will be called in response to any modification of the **JGoDocument** or any of the

Building a Sample Application Using JGo Beans

JGoObjects contained in the document, including creation and deletion of **JGoObjects**. We will modify any link as it is created to add an arrowhead.

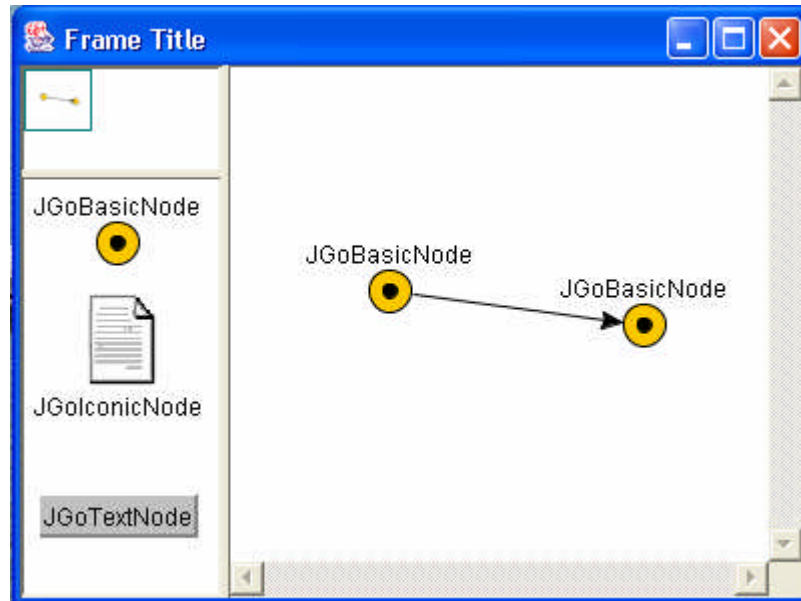
In JBuilder, click on `jGoView1` to select it for editing. Click on the Events tab to view all the available event listeners. Click on **documentChanged** and add an event listener. A **documentChangedListener** class will be automatically created and the following empty method will be added to the **Frame1** class:

```
void jGoView1_documentChanged(JGoDocumentEvent e) {  
}
```

We will add code to this method to look for new **JGoLink** objects being created, and to modify the appearance of those links to include an arrowhead on the “to” end of the link as follows:

```
void jGoView1_documentChanged(JGoDocumentEvent e) {  
    switch (e.getHint()) {  
        case JGoDocumentEvent.INSERTED:  
            if (e.getJGoObject() instanceof JGoLink) {  
                JGoLink link = (JGoLink)e.getJGoObject();  
                link.setArrowHeads(false, true);  
            }  
            break;  
    }  
}
```

Build and run the program again. Drag and drop two **JGoBasicNodes** onto the **JGoView**. Create a new link by dragging a link from the port of one **JGoBasicNode** to the port of the other. The results should appear as follows:



JGo User Guide

Next we'll add a **viewChanged** event listener on the **JGoView**. This event listener will be called in response to any modification of the **JGoView**, including any user interaction with the objects shown in the view, such as selection, click, or double-click. We will react to any double click on an object by displaying a message dialog identifying the class of the object upon which the user double-clicked.

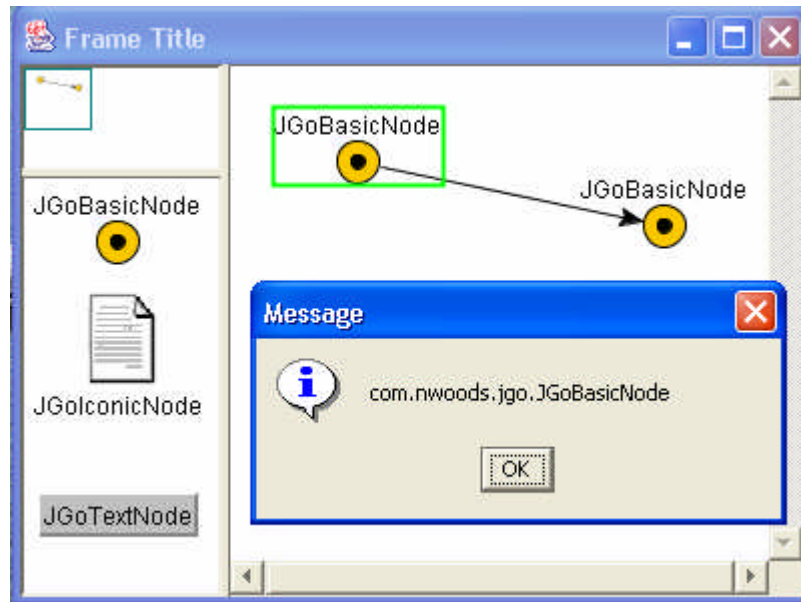
In JBuilder, click on `jGoView1` to select it for editing. Click on the Events tab to view all the available event listeners. Click on **viewChanged** and add an event listener. A **viewChangedListener** class will be automatically created and the following empty method will be added to the **Frame1** class:

```
void jGoView1_viewChanged(JGoDocumentEvent e) {  
}
```

We will add code to this method to look for double-click events on **JGoObjects** or on the background and display an appropriate message dialog as follows:

```
void jGoView1_viewChanged(JGoDocumentEvent e) {  
    switch (e.getHint()) {  
        case JGoViewEvent.DOUBLE_CLICKED:  
            JOptionPane.showMessageDialog(null,  
                e.getJGoObject().getTopLevelObject()  
                    .getClass().getName());  
            break;  
        case JGoViewEvent.BACKGROUND_DOUBLE_CLICKED:  
            JOptionPane.showMessageDialog(null,  
                "Double-clicked on background");  
            break;  
    }  
}
```

Build and run the program again. Double-click on one of the nodes dragged to the **JGoView**. The resulting window should appear as follows:



Next we'll add a `keyPressed` event listener on the **JGoView**. This event listener will be called in response to any key being pressed while the **JGoView** has focus. We will look for the delete key being pressed and react by deleting the currently selected objects, if any.

In JBuilder, click on `jGoView1` to select it for editing. Click on the Events tab to view all the available event listeners. Click on `keyPressed` and add an event listener. A **keyAdapter** class will be automatically created and the following empty method will be added to the **Frame1** class:

```
jGoView1_keyPressed(KeyEvent e) {
}
```

We then add code to this method to look for a Delete key being pressed and then remove the currently selected objects from the document:

```
jGoView1_keyPressed(KeyEvent e) {
    switch (e.getKeyCode()) {
        case KeyEvent.VK_DELETE:
            jGoView1.deleteSelection();
            break;
    }
}
```

Build and run the program again. Select several nodes and links. Use shift-click to extend the selection, ctrl-click to toggle the selection, or use rubber band selection (mouse down and drag to select all objects in enclosed rectangle). Press the Delete key to delete all the selected objects.

Customize the Palette

Naturally, we'll want our own set of nodes to appear in the palette. The nodes displayed when the `showSampleItems` property is set to true are primarily useful for boot strapping and testing purposes when first creating an application. At this point we're ready to set this property to false and add our own instances of nodes.

The nodes we create will typically be subclasses of **JGoNode** or **JGoArea**. The **JGoArea** class collects the various individual parts of our object (images, text, ports, etc.) so that they behave as a single entity. We may wish to find an object class defined in the **com.nwoods.jgo** or **com.nwoods.jgo.examples** package that has similar appearance and behavior to that which we are attempting to create and either make a new similar class or subclass in order to create our new node class. Looking at the **com.nwoods.jgo.examples.demo1** sample application can be helpful in order to explore the various different node types and their behavior.

For simplicity in this example, we will create a subclass of **JGoBasicNode** that adds a single additional integer field as follows:

```
public class SampleNode extends JGoBasicNode {
    public SampleNode() {
    }
    public SampleNode(String label) {
        super(label);
    }
    public int getIntVal(){
        return myInt;
    }
    public void setIntVal(int iVal){
        myInt = iVal;
    }
    private int myInt;
}
```

In the initialization code for the **JGoPalette** we can now replace the line:

```
jGoPalettel.setShowSampleItems(true);
```

with the following code to create three different colored instances of our new **SampleNode** class:

```
// jGoPalettel.setShowSampleItems(true);
SampleNode node1 = new SampleNode("Blue Node");
node1.setBrush(new JGoBrush(Color.blue));
node1.setIntVal(1);
SampleNode node2 = new SampleNode("Red Node");
node2.setBrush(new JGoBrush(Color.red));
node2.setIntVal(2);
SampleNode node3 = new SampleNode("Green Node");
node3.setBrush(new JGoBrush(Color.green));
```

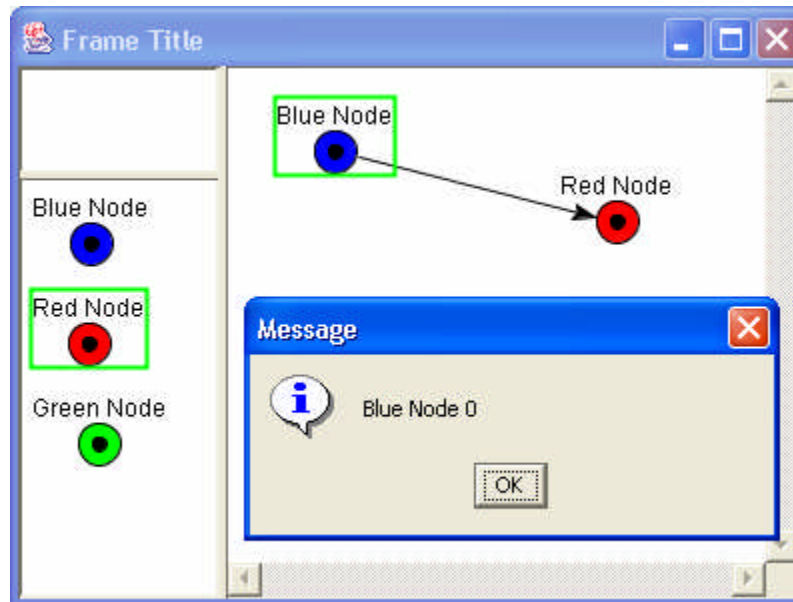
Building a Sample Application Using JGo Beans

```
node3.setIntVal(3);
jGoPalettel.getDocument().addObjectAtTail(node1);
jGoPalettel.getDocument().addObjectAtTail(node2);
jGoPalettel.getDocument().addObjectAtTail(node3);
jGoPalettel.layoutItems();
```

We'll also modify the message that's displayed when our user double-clicks on a node to display the "myInt" value added by our **SampleNode** subclass by modifying the **DOUBLE_CLICKED** event handler as follows:

```
void jGoView1_viewChanged(JGoViewEvent e) {
    switch (e.getHint()) {
        case JGoViewEvent.DOUBLE_CLICKED:
            JGoObject obj = (JGoObject)e.getJGoObject()
                                .getTopLevelObject();
            if (obj instanceof SampleNode) {
                SampleNode node = (SampleNode)obj;
                JOptionPane.showMessageDialog(null, node.getText() + " "
                    + Integer.toString(node.getIntVal()));
            }
            else {
                JOptionPane.showMessageDialog(null,
                    e.getJGoObject().getTopLevelObject()
                        .getClass().getName());
            }
            break;
        case JGoViewEvent.BACKGROUND_DOUBLE_CLICKED:
            JOptionPane.showMessageDialog(null,
                "Double-clicked on background");
            break;
    }
}
```

Build and run the program again. Double-click on one of the nodes dragged to the **JGoView**. The resulting window should appear as follows:



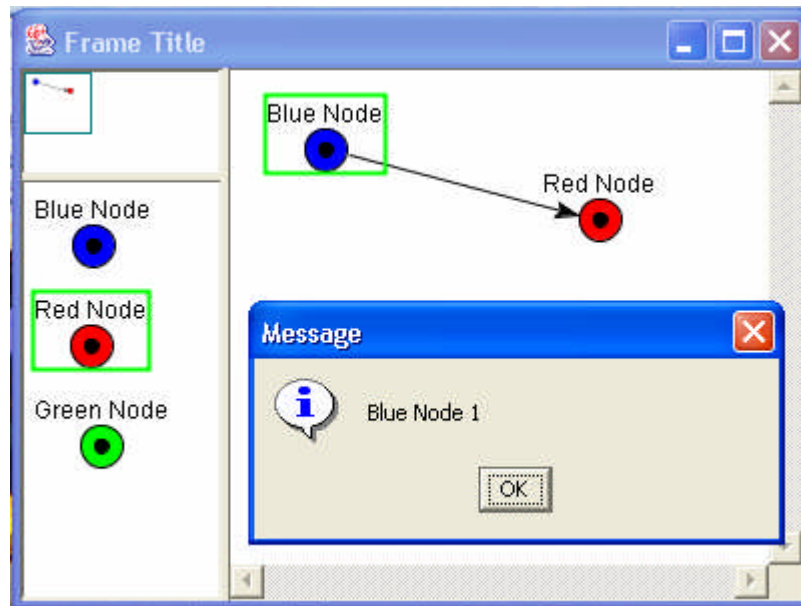
Note that the integer value displayed for the blue node is 0 rather than 1. This is because a new copy of the **SampleNode** is created when it is dropped in the **JGoView** window. By default, JGo will make a copy of an object whenever that object is:

- involved in a drag and drop operation
- involved in a clipboard operation

The objects are copied by invoking the virtual **JGoObject.copyObject** method. Because we have created our own **SampleNode** class and have added a data member to that class, we must override **SampleNode.copyObject** to copy the additional data as follows:

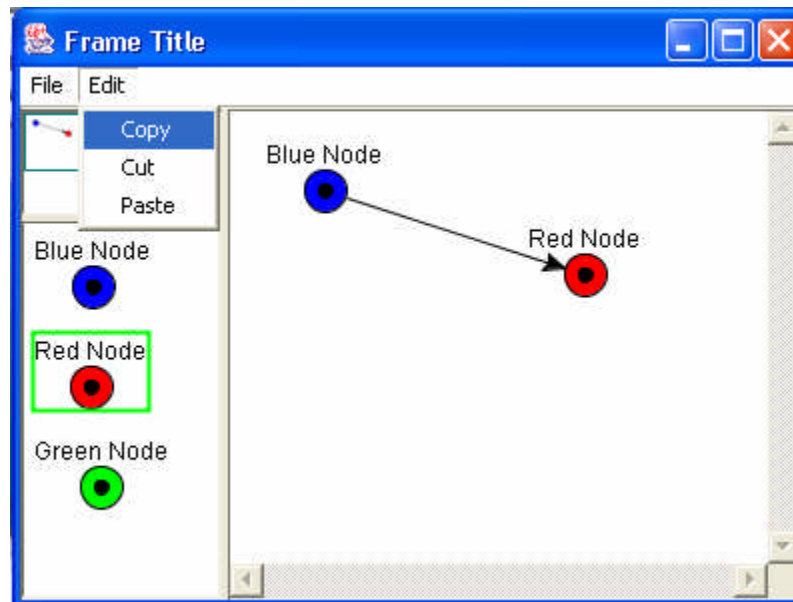
```
public JGoObject copyObject(JGoCopyEnvironment env) {  
    SampleNode newObj = (SampleNode)super.copyObject(env);  
    newObj.myInt = myInt;  
    return newObj;  
}
```

Build and run the program again. Double-click on one of the nodes dragged to the **JGoView**. The resulting window should appear as follows:



Add Clipboard Support

Next we'll add a menu bar to the sample application. We'll create a **JMenuBar** called `jMenuBar1` and populate it with clipboard commands. Name the new menu items `CopyItem`, `CutItem`, and `PasteItem`. Make sure that the **JMenuBar** property of **Frame1** is set to `jMenuBar1`. The resulting window should appear as follows:



Next we'll add an **actionPerformed** event listener on the "Copy" menu item. This event listener will be called in response to the user selecting the "Copy" command from the menu.

In JBuilder, click on the "Copy" menu item (`CopyItem`) in `jMenuBar1` to select it for editing. Click on the Events tab to view all the available event listeners. Click on

JGo User Guide

actionPerformed and add an event listener. An **actionListener** class will be automatically created and an empty method will be added to the **Frame1** class. Repeat the above operations to create **actionListeners** and methods for the other menu items. The following empty methods will be created for the “Copy”, “Cut”, and “Paste” menu items:

```
void CopyItem_actionPerformed(ActionEvent e) {  
}  
void CutItem_actionPerformed(ActionEvent e) {  
}  
void PasteItem_actionPerformed(ActionEvent e) {  
}
```

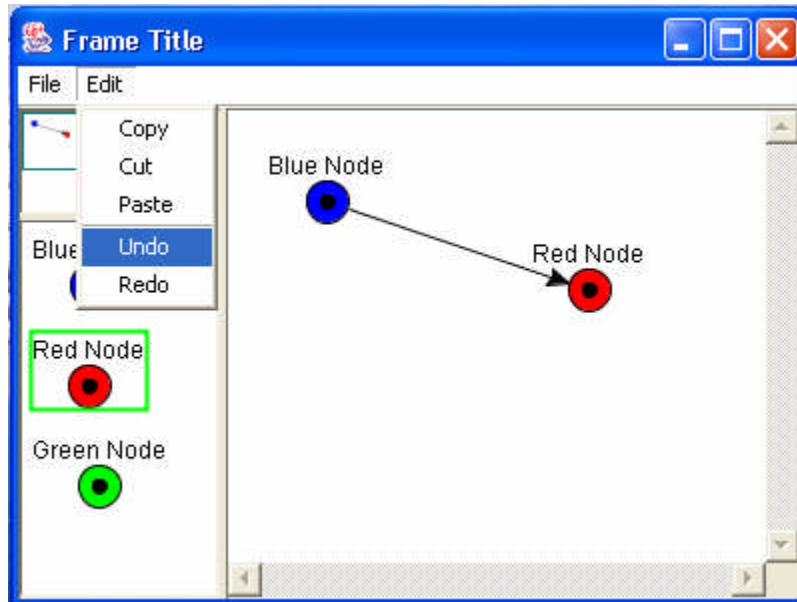
We will add code to these methods to simply invoke the standard cut, copy, and paste operations on the **JGoView** as follows:

```
void CopyItem_actionPerformed(ActionEvent e) {  
    jGoView1.copy();  
}  
void CutItem_actionPerformed(ActionEvent e) {  
    jGoView1.cut();  
}  
void PasteItem_actionPerformed(ActionEvent e) {  
    jGoView1.paste();  
}
```

Build and run the program again. Select several nodes and links. Select the “Copy” or “Cut” command to move the selected items to the clipboard. Select the “Paste” operation to create new copies of the items on the clipboard. Note that the pasted items are initially created in the same relative location they were in when copied to the clipboard. Also note that when you double-click on the pasted objects, the integer value associated with the **SampleNode** objects are preserved. The same **copyObject** method that we implemented to support copying the additional data members in our **SampleNode** class for drag and drop operations also works for clipboard operations.

Add Undo/Redo Support

We’ll start by adding the menu items for undo and redo to the menu bar. The updated menu bar should appear as follows:



Just as we did before when defining the clipboard operations, click on the “Undo” menu item (UndoItem) in jMenuBar1 to select it for editing. Click on the Events tab to view all the available event listeners. Click on **actionPerformed** and add an event listener. An **actionListener** class will be automatically created and an empty method will be added to the **Frame1** class. Repeat the above operations to create **actionListeners** and methods for the redo menu items. The following empty methods will be created for the “Undo” and “Redo” menu items:

```
void UndoItem_actionPerformed(ActionEvent e) {
}
void RedoItem_actionPerformed(ActionEvent e) {
}
```

We will add code to these methods to simply invoke the standard undo and redo operations on the **JGoView** as follows:

```
void UndoItem_actionPerformed(ActionEvent e) {
    jGoView1.getDocument().undo();
}
void RedoItem_actionPerformed(ActionEvent e) {
    jGoView1.getDocument().redo();
}
```

We must also add a **JGoUndoManager** to the **JGoDocument** in order to control the undo and redo operations. We will simply add a default **JGoUndoManager** in the initialization code for **Frame1** as follows:

```
JGoUndoManager undoManager = new JGoUndoManager();
jGoView1.getDocument().setUndoManager(undoManager);
```

Build and run the program again. Perform several operations, such as drag and drop, link nodes, drag nodes from one location to another. Verify that the “Undo” and “Redo” menu items faithfully undo and redo these operations.

JGo User Guide

Because our sample application does not yet support any modifications to our **SampleNode** data member (`myInt`) after the object has been added to the **JGoDocument**, there is no need to track changes to this item or save or restore its previous values. However, in the interests of a more robust example, let us assume that in the future we wish to allow changes to this value and that all such changes will occur as a result of calling **SampleNode.setIntVal(int iVal)**. We would then need to define the following methods in **SampleNode** to track changes to this value and modify the value during undo and redo operations:

```
public void setIntVal(int iVal){
    int oldVal = myInt;
    if (oldVal != iVal) {
        myInt = iVal;
        // Signal state change to support undo/redo
        update(IntValChanged, oldVal, null);
    }
}

public void copyNewValueForRedo(JGoDocumentChangedEdit e)
{
    // Copy the current state before doing the undo so it can
    // be reset in a future redo operation
    switch (e.getFlags()) {
        case IntValChanged:
            e.setNewValueInt(myInt);
            return;
        default:
            super.copyNewValueForRedo(e);
            return;
    }
}

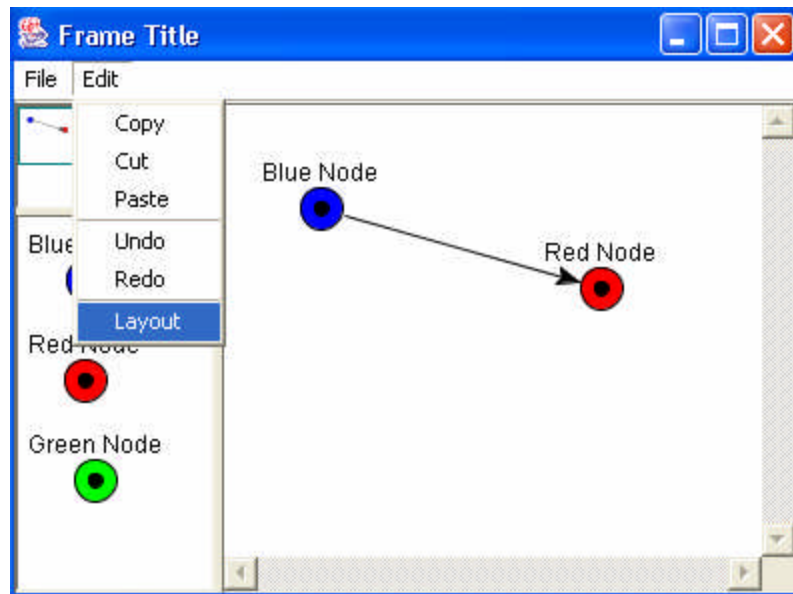
public void changeValue(JGoDocumentChangedEdit e, boolean undo)
{
    // Actually perform the undo or redo operation
    switch (e.getFlags()) {
        case IntValChanged:
            setIntVal(e.getValueInt(undo));
            return;
        default:
            super.changeValue(e, undo);
            return;
    }
}

public static final int IntValChanged = JGoDocumentEvent.LAST +
10000;
```

Add Auto-layout Support

In the section entitled “Register the JGo Beans with the Development Environment”, we added the `com.nwoods.jgo.layout` package to the libraries required by this sample application. This optional package is required for auto-layout support. For more details on the auto-layout package and different layout options, refer to the “JGo Layout User Guide”.

We’ll start by adding the menu items for auto-layout to the menu bar. The updated menu bar should appear as follows:



Just as we did before, click on the “Layout” menu item (`LayoutItem`) in `jMenuBar1` to select it for editing. Click on the Events tab to view all the available event listeners. Click on **actionPerformed** and add an event listener. An **actionListener** class will be automatically created and an empty method will be added to the **Frame1** class. Repeat the above operations to create **actionListeners** and methods for the redo menu items. The following empty method will be created for the “Layout” menu item:

```
void LayoutItem_actionPerformed(ActionEvent e) {
}
```

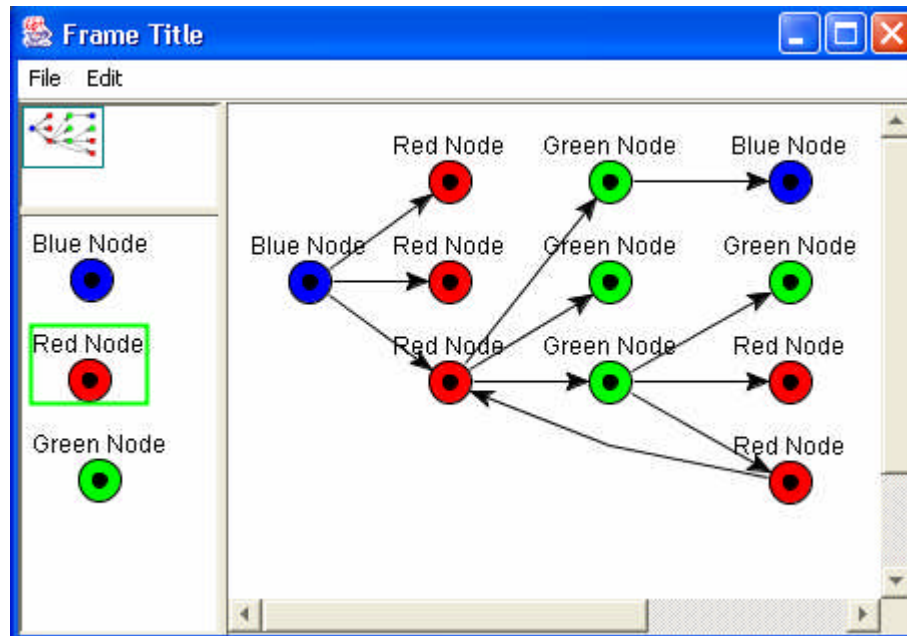
We will add code to this method to perform a layered, directed graph layout of the nodes displayed in the **JGoView**. We accomplish this by first constructing a **com.nwoods.jgo.layout.JGoLayeredDigraphAutoLayout** object, passing the **JGoDocument** to the constructor so that a **JGoNetwork** of nodes and links can be automatically created for us. We then specify property values for layout direction, layer spacing, and column spacing. Finally, we call **performLayout** to cause the layout operation to actually take place as follows:

```
void LayoutItem_actionPerformed(ActionEvent e) {
    JGoLayeredDigraphAutoLayout layout =
        new JGoLayeredDigraphAutoLayout(jGoView1.getDocument());
```

JGo User Guide

```
layout.setDirectionOption(  
    JGoLayeredDigraphAutoLayout.LD_DIRECTION_RIGHT);  
layout.setLayerSpacing(10);  
layout.setColumnSpacing(10);  
layout.performLayout();  
}
```

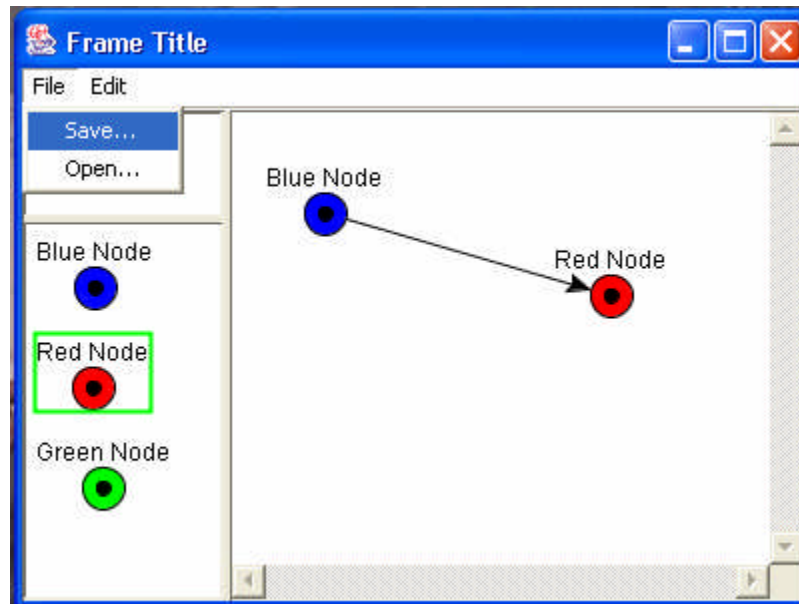
Build and run the program again. Create a network of nodes and links and select the “Edit/Layout” option from the menu. The resulting window may appear as follows:



Add XML/SVG Serialization Support

In the section entitled “Register the JGo Beans with the Development Environment”, we added the **com.nwoods.jgo.svg** package to the libraries required by this sample application. This optional package is required for serialization to and from the extended SVG XML document type. Refer to the “Serialization” section of “JGoDocument” in the “JGoDocument and JGoObject Details” chapter for more information on this topic.

We’ll start by adding the menu items for saving and opening documents to the menu bar. The updated menu bar should appear as follows:



Just as we did before, click on the “Save...” menu item (SaveItem) in jMenuBar1 to select it for editing. Click on the Events tab to view all the available event listeners. Click on **actionPerformed** and add an event listener. An **actionListener** class will be automatically created and an empty method will be added to the **Frame1** class. Repeat the above operations to create **actionListeners** and methods for the “Open...” menu item. The following empty methods will be created for the “Save...” and “Open...” menu items:

```
void SaveItem_actionPerformed(ActionEvent e) {
}
void OpenItem_actionPerformed(ActionEvent e) {
}
```

We will add code to these methods to save and restore the graph shown in the **JGoView** to and from XML documents. The XML document format used is an extension of the SVG (Scalable Vector Graphics) XML document type. We will use a **JFileChooser** object to identify the path of the file to be read or written, and create a **FileInputStream** or **FileOutputStream** associated with that file. Finally, we will create an instance of **com.nwoods.jgo.svg.DefaultDocument** and call the **SVGReadDoc** or **SVGWriteDoc** methods to read or write the extended SVG document. The resulting code is as follows:

```
void SaveItem_actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showSaveDialog(null);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
        try {
            String loc = chooser.getSelectedFile().getAbsolutePath();
            FileOutputStream fstream = new FileOutputStream(loc);
            DefaultDocument svgDomDoc = new DefaultDocument();
            svgDomDoc.SVGWriteDoc(fstream, jGoView1.getDocument());
        } catch (Exception ex) {
```

JGo User Guide

```
        ex.printStackTrace();
    }
}

void OpenItem_actionPerformed(ActionEvent e) {
    JFileChooser chooser = new JFileChooser();
    int returnVal = chooser.showOpenDialog(this);
    if (returnVal == JFileChooser.APPROVE_OPTION) {
        try {
            String loc = chooser.getSelectedFile().getAbsolutePath();
            FileInputStream fstream = new FileInputStream(loc);
            DefaultDocument svgDomDoc = new DefaultDocument();
            svgDomDoc.SVGReadDoc(fstream, jGoView1.getDocument());
        } catch (Exception ex) {
            ex.printStackTrace();
        }
    }
}
```

Build and run the program again. Create a network of nodes and links and select the “File/Save...” option from the menu. Verify that the output file can be read back into the sample application using the “File/Open...” option from the menu.

At this point, the JGo graph is being successfully serialized to and from a file on disk, but our extensions to JGo are not. In particular, our **SampleNode** subclass of **JGoBasicNode** is not being saved correctly. When the file is read back into our sample application, all the nodes in our graph have been recreated as **JGoBasicNodes** rather than our **SampleNode** subclass. We can see this by double clicking on any node in the diagram. Instead of seeing the message pane for a **SampleNode** object (“Blue Node 1”, for example) we see the generic message used for all other object types that simply shows the class name of the double clicked object (“com.nwoods.jgo.JGoBasicNode” for example).

We need to override **JGoObject.SVGWriteObject** and **JGoObject.SVGReadObject** in order to cause our subclass to be saved, and restored properly. In **SVGWriteObject** we call **createJGoClassElement** to create an XML element corresponding to our subclass. The class name passed to this element must be a fully qualified class name. This class name will be used to recreate an instance of our subclass when reading the file back in, so it is important to make sure this name is entered correctly. We use the **setAttribute** method to add an attribute called “inval” to this element and use it to save the value of our only data member.

When reading the file back in with **SVGReadObject**, use the **getAttribute** method to retrieve our data member value.

In both **SVGReadObject** and **SVGWriteObject**, we must remember to call `super()` to allow our superclasses to save and restore their data. The resulting code is as follows:

```
public void SVGWriteObject(DomDoc svgDoc, DomElement
```

Building a Sample Application Using JGo Beans

```
jGoElementGroup)
{
    // Add SampleNode element
    DomElement sampleNode = svgDoc.createJGoClassElement(
        "com.nwoods.jgo.examples.sampleapp.SampleNode", jGoElementGroup);
    sampleNode.setAttribute("intval", Integer.toString(myInt));
    // Have superclass add to the JGoObject group
    super.SVGWriteObject(svgDoc, jGoElementGroup);
}

public DomNode SVGReadObject(DomDoc svgDoc, JGoDocument jGoDoc,
    DomElement svgElement, DomElement jGoChildElement)
{
    if (jGoChildElement != null) {
        // This is a SampleNode element
        myInt = Integer.parseInt(jGoChildElement.getAttribute("intval"));
        super.SVGReadObject(svgDoc, jGoDoc, svgElement,
            jGoChildElement.getNextSiblingJGoClassElement());
    }
    return svgElement.getNextSibling();
}
```

Build and run the program again. Create a network of nodes and links and select the “File/Save...” option from the menu. Verify that the output file can be read back into the sample application using the “File/Open...” option from the menu. Also verify that double-clicking on the restored nodes displays the correctly restored value for the `intval` property of each **SampleNode**.

The output file can be viewed as using an SVG viewer, an XML viewer, or simply using a standard text editor. The portion of the extended SVG XML output relating to our **SampleNode** subclass of **JGoBasicNode** is as follows:

```
<JGoClass class="com.nwoods.jgo.examples.SampleNode" intval="1"/>
```