GLG Programming Tutorial for Java

GLG Toolkit
Version 3.7

Generic Logic, Inc.
6 University Drive 206-125
Amherst, MA 01002
USA
Telephone: (413) 253-7491
FAX: (413) 241-6107
email: support@genlogic.com
web: www.genlogic.com

November 25, 2017
Software Release Version 3.7

GLG Programming Tutorial for Java

Table of Contents

GLG Programming Tutorial for Java

T

Overview

The GLG Toolkit provides a 100% pure Java class library that allows the user to load, display and animate GLG drawings in a Java application. A GLG drawing can be created in the GLG Graphics Builder or programmatically. It may contain predefined GLG components from the GLG Widgets catalog, such as graphs, gauges, meters, or custom graphics created by the user.

A GLG Java application can be written as a stand-alone Java program or as an applet. To display GLG graphics, an application creates a GLG bean that can be used either as a top level applet or as a Java bean inside another Java component. Examples of using the GLG bean in both cases appear later in this tutorial, as well as in the examples directory *<glg_dir>/examples_java*.

The GLG Toolkit for Java supports Swing and Java2D.

Types of Glg beans

There are several types of the GLG Bean class provided.

GlgJBean
Heavyweight Swing-based Java bean and applet, subclassed from *JApplet*.

GlgJLWBean
Lightweight Swing-based Java bean for use with *JDesktopPane* and *JInternalFrame*.

GlgBean
An AWT-based Java bean and applet, subclassed from *Applet*.

All GLG Bean classes have the same GLG-inherited interface and differ only in the methods inherited from their respective Swing or AWT superclasses. The Swing and AWT-based GLG beans should not be intermixed in one application.

All GLG Bean components are recognized by Java IDEs. They may be directly inserted into an application or subclassed (as seen in our demos and examples of this tutorial) to create custom components with additional functionality.

Since all GLG Bean classes have the same functionality, this tutorial uses the generic term GlgBean for all of them, regardless of the actual flavor of the bean used.

Classes and Packages

The Toolkit provides GLG jar files containing the **com.genlogic** package which includes the entire GLG programming API for GLG Java classes. The main GLG classes are the ***GlgObject*** class and the **GLG Bean** classes (*GlgJBean, GlgJLWBean* or *GlgBean*). Both classes provide GLG API and

GLG Extended API methods. A few utility classes, such as *GlgPoint*, *GlgCube*, *GlgTraceData*, provide some convenient functionality. The rest of the classes (*GlgArc*, *GlgPolygon*, etc.) extend the generic *GlgObject* class, providing constructors for creating GLG objects of different types.

The *GlgObject* class is a generic GLG superclass that provides API methods for all GLG objects. The GLG bean classes (*GlgJBean*, *GlgJLWBean*, *GlgBean*) provide the same set of methods as those for the *GlgObject* class. While the *GlgObject* methods have to be invoked for individual objects, the bean provides a central location for invoking methods involving any object or resource.

The standard API provides methods for loading, displaying and animating the drawing, handling user interaction, and setting and querying resource values. The Extended API provides methods for creating and adding objects to the drawing programmatically, as well as programmatic access to object manipulation.

Jar File Usage

The GLG API package must be included in all GLG Java programs as follows:

```
import com.genlogic.*;
```

To compile GLG Java programs, the **CLASSPATH** environment variable should include one of the following GLG jar files:

For the evaluation version of the Toolkit

GlgCE.jar	Standard, Intermediate and Extended API for the GLG Bean
GlgServerCE.jar	Standard, Intermediate and Extended API for the JSP Servlet

For the production version of the Toolkit

Glg2.jar	Standard API for the GLG Bean
GlgInt2.jar	Standard and Intermediate API for the GLG Bean
GlgExt2.jar	Standard, Intermediate and Extended API for the GLG Bean
GlgServer.jar	Standard, Intermediate and Extended API for the JSP Servlet

The GLG jar files for the evaluation version are located in the *<glg_dir>/eval/lib* directory, and for the production version in the *<glg_dir>/lib* directory. For the Community Edition, the jar files are also located in the *<glg_dir>/lib* directory.

For example, to use the evaluation version of the Toolkit, the **CLASSPATH** environment variable should be set as follows:

on Windows in the MSDOS prompt,

```
set CLASSPATH=.;<glg_dir>\eval\lib\GlgCE.jar
```

on Unix,

```
export CLASSPATH=.:<glg_dir>/eval/lib/GlgCE.jar
```

To run GLG Java demos, the **GlgDemo.jar** file should also be added to the CLASSPATH environment variable. GlgDemo.jar contains the classes for all GLG Java demos and is located in the *<glg_dir>/DEMOS/java_demos/java2* directory. This directory also contains the GlgCE.jar file and the demo drawings.

For example, to run the GLG Java demos on Windows, set the CLASSPATH environment variable as follows:

```
cd <glg_dir>\DEMOS\java_demos\java2
set CLASSPATH=.;.\GlgDemo.jar;.\GlgCE.jar
```

To run the demos on Unix/Linux:

```
cd <glg_dir>/DEMOS/java_demos/java2
set CLASSPATH=.:./GlgDemo.jar:./GlgCE.jar
```

Another example:

If the production version of the Toolkit is being used with the GLG Standard API, set the CLASSPATH environment variable as follows.

on Windows in the MSDOS prompt,

```
set CLASSPATH=.;<glg_dir>\lib\Glg2.jar
```

on Unix,

```
export CLASSPATH=.:<glg_dir>/lib/Glg2.jar
```

1. Structure of Typical GLG Java Application

The structure of a typical GLG Java application proceeds as follows.

- Adding the GlgBean to an application interface.
- Loading and displaying a drawing in the GlgBean.
- Initializing the drawing.
- Handling dynamic updates.
- Handling user interaction, including the selection of objects and interacting with GLG control widgets.

Each of these steps, including code samples and detailed explanation, is described in detail later in this tutorial.

Before plunging into the technical details of these steps, let us examine the general model of a GLG application. A typical GLG application loads and displays one or more GLG drawings, animates the drawings with dynamic data and responds to user interaction. The Java version of the GLG Toolkit is both data and event driven.

When the application's data changes, the Toolkit's resource mechanism is used to set new values for the drawing's resources. New data values may be used to control the attributes or dynamics of the drawing's objects, or the values may be displayed in a GLG graph widget. The Toolkit automatically updates the drawing to reflect the new data values, handling all aspects of attribute changes, dynamics recalculation, damage repair and double buffering.

The Toolkit automatically handles most low-level Java events. When a user interacts with a GLG drawing, the low-level events are converted into high-level GLG events, such as input or selection events, and then passed to the application for processing.

There are several types of user input events processed via the *Input* listener. Low-level input events are generated when the user interacts with input objects, such as buttons, toggles, knobs and sliders. *Command* and *Custom* events are triggered by a specific user interaction with input objects that have input actions attached. The action's parameters specify when the action is triggered and define a command or a custom event to execute.

The *Input* listener is also used to process object selection events. If a drawing's *ProcessMouse* attribute includes *Click* and/or *Move* masks, the low-level *Object Selection* events are generated on MouseClick and/or MouseOver. For objects that have mouse actions attached, *Command* and *Custom* events are generated when the user selects objects with mouse button or moves the mouse over them. The action's parameters specify when the action is triggered (on a *MouseClick* or *MouseOver*) and define a command or a custom event to execute.

The *Select* listener provides a simplified object selection interface based on object names.

Adding GlgBean to an Application Interface

A GlgBean may be used in various ways in a Java application.

- as a Java bean component in a stand-alone Java application.
- as a top level applet;
- as a Java bean component inside another applet;

Using GlgBean in a Stand-alone Java Program

To use a GlgBean in a stand-alone Java program, the GlgBean should be created and added to the application interface. Once a GlgBean object has been created, the ***SetDrawingName*** method can be used to specify a drawing filename to be displayed in the GlgBean.

Below is an example of a GlgBean being added to a *Frame* and a drawing name, "bar1.g," set to be displayed in the GlgBean:

```
public static void main( String arg[] )
{
   // Use invokeLater to create all interface components synchronously
   // with the event thread.
   SwingUtilities.invokeLater(
      new Runnable(){ public void run() { Main( arg ); } } );
}

public static void Main( final String arg[] )
{
   //....
   JFrame frame = new JFrame();
   frame.setResizable( true );
   frame.setSize( 600, 400 );

   // Create a GlgBean component
   GlgJBean glg_bean = new GlgJBean();

   // Set a GLG drawing name to be displayed in a GLG bean
   glg_bean.SetDrawingName( "bar1.g" );

   // Add glg_bean component to a frame
   frame.getContentPane().add( glg_bean );
   frame.show();
}
```

The path to the drawing filename is relative to the current directory. The details of specifying a drawing to be displayed in a GlgBean are described in the *Loading and Displaying a Drawing* section on page 13.

Examples of using a GlgBean in a stand-alone Java program may be found in the *<glg_dir>/examples_java* directory. Refer to the README file in that directory for a description of each example.

Using GlgBean as an Applet

To use a GlgBean as a top level applet, an application should subclass a GlgBean class as follows.

```
public class GlgGraphApplet extends GlgJBean implements ActionListener
{
   //...
}
```

The *ActionListener* interfaces is optional and is used to implement an update timer for animating the bean with dynamic data.

A GLG drawing displayed in the applet can be specified either as a ***DrawingURL*** parameter in the *.html* file or set directly in the code.

The following example shows the use of the DrawingURL parameter in the *.html* file:

```
<param NAME=DrawingURL VALUE=bar1.g>
```

Unless an absolute path is used, the name of the drawing URL is relative to the applet's document base directory.

Alternatively, to specify the drawing to be displayed in the Java code, the GlgBean's ***SetDrawingURL*** method can be used. The following example shows how to invoke SetDrawingURL from the applet's *start* method:

```
public class GlgGraphApplet extends GlgJBean implements ActionListener
{
   //...
   // Invoked by the browser to start the applet
   public void start()
   {
      SetDrawingURL( "bar1.g" );
      //...
   }
}
```

In both cases, unless an absolute path is used, the drawing URL is relative to the applet's document base directory.

For details, refer to the *Loading and Displaying a Drawing* section on page 13.

The GlgGraphApplet.java example, located in the *<glg_dir>/examples_java* directory, demonstrates the usage of a GlgBean as a top level applet.

Using GlgBean as a Component of an Applet

The following code sample demonstrates how to add a GlgBean as a component to another applet.

```
public class GlgGraphComponent extends JApplet
{
   GlgJBean glg_bean;

   //Constructor
   public GlgGraphComponent()
   {
      setLayout( new GridLayout( 1, 1 ) );
      glg_bean = new GlgJBean();
      add( glg_bean );
   }

   // Invoked by the browser to start the applet
   public void start()
   {
      super.start();
      glg_bean.SetParentApplet( this );
      glg_bean.SetDrawingName( "bar1.g" );
   }
   //...
}
```

In the above code, the ***SetDrawingName*** method is used to specify the drawing to be displayed in the GlgBean. In this case, the *SetDrawingURL* method could have been used as well, since the GlgBean is used inside an applet, where the drawing is specified as a URL. *SetDrawingName* is a generic method which can be used in a stand-alone program as well as in a web environment. If *SetDrawingURL* is used to set the drawing, an absolute URL path can be specified instead of a local one.

If a relative path is used to set the drawing name, the GlgBean must know what its parent applet is in order to retrieve information about the applet's *DocumentBase* directory, which is used by the GlgBean to load the drawing. As shown in the code sample above, GlgBean's *SetParentApplet* method should be used to set a parent applet for the GlgBean.

For details, refer to the *Loading and Displaying a Drawing* section on page 13.

Subclassing GlgBean

In addition to the standard GlgBean methods, the GlgBean class may be subclassed to provide additional custom functionality. Subclassing the GlgBean may be very convenient for creating custom beans that display graphics and encapsulate graphical and user interaction functionality. Such beans may be used as custom components in the application.

There are two GLG beans available for Swing applications: a *heavyweight* **GlgJBean**, or a *lightweight* **GlgJLWBean**. The advantages and disadvantages of using a heavyweight GLG bean as opposed to a lightweight bean is described in the *Using GLG Components in Java* chapter. For

AWT applications, **GlgBean** class is provided. The technique of subclassing the *GlgJBean*, *GlgJLWBean* and *GlgBean* is identical. Therefore all examples in this chapter refer only to the GlgJBean class, regardless of the GLG bean flavor actually used.

The following sample code illustrates how to subclass a GlgBean and add a new *GlgGraphClass* component to an applet.

```
public class GlgSubclassExample extends JApplet
{
   // Custom component to display and update a GLG graph
   GlgGraphClass graph_component;

   // Constructor
   public GlgSubclassExample()
   {
      //...
      // Create a custom component and add it to the applet
      graph_component = new GlgGraphClass();
      add( graph_component );
   }

   // Invoked by the browser to start the applet
   public void start()
   {
      // Define parent applet for the custom graph component
      graph_component.SetParentApplet( this );
   }

   // Class definition for a custom component.
   class GlgGraphClass extends GlgJBean implements ActionListener
   {
      // Constructor
      public GlgGraphClass()
      {
         // Set the drawing by specifying its name.
         SetDrawingName( "bar1.g" );
      }

      // Ready callback to place initialization code
      public void ReadyCallback( GlgObject viewport )
      {
         // Start periodic dynamic updates
         StartUpdates();
         //...
      }

      // Input callback to handle user interaction
      public void InputCallback( GlgObject viewport,
                                 GlgObject message_obj )
      {
         //...
      }

      // Update procedure to update the graph with random data.
```

```
        // Invoked by a timer.
        void UpdateGraphProc()
        {
           // Push next data value into a graph
           SetDResource( "DataGroup/EntryPoint", GetData() );
           Update();
           //...
        }
     }
  }
```

The name of the drawing may be specified in the constructor of the inherited class, as shown in the above code sample. The drawing name is relative to the current directory, if running a stand-alone Java program, or to the document base directory if a custom GLG component is being used inside an applet. The document base directory of a parent applet must be passed to the custom component using GlgBean's ***SetParentApplet*** method. As shown in the above sample code, this can be done in the start method of the parent applet.

A custom GLG component may also handle the dynamic updates occurring in the GLG drawing so that all graphical functionality is encapsulated in one class. As shown in the above code sample, periodic dynamic updates may be started in the *Ready* callback using a *timer*. In this code sample, *StartUpdates* is a generic method used to accomplish this. Dynamic updates are handled in the *UpdateGraphProc* method.

An example of subclassing a GLG bean may be found in *GlgSubclassExample* located in the *<glg_dir>/examples_java* directory.

Loading and Displaying a Drawing

A GLG drawing to be displayed in the GlgBean may be created in the GLG Builder or programmatically using the GLG Extended API.

A GLG viewport object, which is an encapsulation of a native window object, is used as a drawing surface for drawing graphics. The viewport is a composite object that contains other GLG graphical objects, such as polygons, arcs, text and others, including embedded viewports.

To display a GLG drawing in the GlgBean, the drawing must contain a viewport at the top of the hierarchy named ***$Widget***, which is used by the GlgBean as a top level viewport. Most predefined GLG components, such as graphs and controls, contain a $*Widget* viewport, allowing them to be displayed in the GlgBean without modification. However, if these components are inserted into other GLG drawings to create custom panels, the $Widget viewport of each component should be renamed so that there is only one top level viewport in the drawing named *$Widget*. For example, if the drawing contains two controls, their viewports could be named *Control1* and *Control2*, whereas the drawing's top level viewport should be named *$Widget*.

For details on using a viewport object and creating nested viewports and custom panels, refer to the *GLG Builder and Animation Tutorial*.

A GLG Java application may need to display multiple GLG components, which can vary from individual widgets, such as graphs and controls, to complex graphical displays. An application may use two approaches.

1. An application can contain one GlgBean that displays a GLG drawing composed of multiple GLG components (graphs, controls, dynamic graphical objects, etc.). Such a GLG drawing may be created in the GLG Builder and then loaded in the program to be displayed in the GlgBean. An example may be found in the GlgControlPanel example located in the *<glg_dir>/examples_java* directory.

2. An application can create multiple GLG beans, each displaying a separate GLG drawing or component. For example, each control or graph is displayed in a separate GlgBean. An example of creating multiple GLG beans may be found in the GlgControlBean example located in the *<glg_dir>/examples_java* directory.

Certainly, an application may also use a combination of the above approaches.

To define a GLG drawing to be displayed in the GlgBean, an application may use the following methods:

- use the ***DrawingURL*** parameter in the *.html* file if the GlgBean is used as a top level applet;
- set the drawing name in the application code.

If the GlgBean is used as a top level applet, GlgBean's *DrawingURL* parameter may be used in the *.html* file to specify a GLG drawing:

```
<param NAME=DrawingURL VALUE=bar1.g>
```

If a full path is not specified, the drawing URL is assumed to be relative to the applet's document base directory.

To specify the drawing to be displayed in the Java code, one of the following GlgBean methods can be used:

void SetDrawingURL(String drawing_url)
void SetDrawingFile(String filename)
void SetDrawingName(String drawing_name)

The ***SetDrawingURL*** method takes a URL as an argument, which can be either a full URL path or a URL relative to the current document base. This method can be used if GlgBean is a top level applet or a component inside another applet.

The ***SetDrawingFile*** method assumes filename, the argument it is passed, is the name of the drawing file on the local system. The filename can be specified using an absolute or pathname relative to the current directory. The "file://..." notation may be used to specify the drawing, but in this case, the *SetDrawingURL* method should be used instead of *SetDrawingFile*. The *SetDrawingFile* method can be used only in a stand-alone Java program.

Regardless of how the GlgBean is used, ***SetDrawingName*** may be used as a generic method for setting the drawing to be displayed in the bean. The Toolkit determines how to interpret the passed drawing name depending on its usage. Unless a full URL path or filename is specified, the name is assumed to be relative to the applet's document base directory or the current directory on the local system.

For example, to set the drawing name to be displayed in the Java code, regardless of whether the GlgBean is used on the web or in a stand-alone program, the *SetDrawingName* method can be used, as shown in the code sample below.

If the GlgBean is used as a Java Bean component within another applet, the GlgBean has to know what the current document base is to determine a full path for the drawing name. To retrieve the document base, the GlgBean provides the ***SetParentApplet*** method to pass the parent applet to the GlgBean.

These methods may be placed in the applet's *start* method which is invoked by the browser when the applet is started. The drawing name should be unset in the applet's *stop* method to perform a cleanup and destroy the drawing.

```
public class GlgGraphComponent extends JApplet implements
      ActionListener
{
   GlgJBean glg_bean;
   //...
   // Invoked by the browser to start the applet
   public void start()
   {
      super.start();
      glg_bean.SetParentApplet( this );
      glg_bean.SetDrawingName( "bar1.g" );
   }

   // Invoked by the browser to stop the applet.
   public void stop()
   {
      // Unset the drawing
      glg_bean.SetDrawingName( null );
      super.stop();
   }
}
```

If the GlgBean is used in a stand-alone Java program, the drawing name may be set in the main() method. Refer to the Java examples located in the *<glg_dir>/examples_java* directory for complete source code.

When the drawing name is set using any of the above methods, a few steps take place internally in the Toolkit:

- the drawing is loaded from a URL or file and the $*Widget* viewport is extracted;
- the hierarchy setup is performed for the objects in the drawing, which includes loading sub-drawings, populating series objects with instances of their template and creation of graphs'

DataSamples

- the drawing is drawn the first time upon receiving a paint event.

A GLG Java application, like any other Java application, is event driven, which means the above steps are completed when certain events are processed. When calling the *SetDrawingName* method, the drawing name is set for the GlgBean, but the drawing may not be loaded right away. Instead, it may wait for particular events to occur. An application should wait for certain events and not expect a drawing and its contents to be set up and available immediately after the call to *SetDrawingName*. To be notified when these events occur, the GlgBean provides various interfaces for event listeners. For example, to set its initial drawing parameters, an application may add a listener to be invoked after the drawing is loaded but before it gets drawn for the first time.

Various GlgBean listeners are discussed in the *Initializing the Drawing* section and the *Handling User Interaction* section.

The methods described above rely on event processing. If complete control of loading is required, for accessibility without waiting for events to be processed, the drawing can be loaded directly using the ***GlgObject.LoadWidget*** method. This method returns the drawing's top level viewport which can be attached to the GlgBean using the ***SetDrawingObject*** method, which can also be used if the drawing is to be created programmatically.

The following code sample illustrates the usage of these methods:

```
public class GlgGraphComponent extends JApplet implements
      ActionListener
{
   GlgJBean glg_bean;
   GlgObject viewport;
   //...
   // Invoked by the browser to start the applet
   public void start()
   {
      super.start();

      // Load GLG drawing from a URL and attach its viewport to the
      glg_bean.
      viewport = GlgObject.LoadWidget( drawing_url, GlgObject.URL);
      glg_bean.SetDrawingObject( viewport );
   }

   public void stop()
   {
      // Unset the drawing.
      glg_bean.SetDrawingObject( null );
      super.stop();
   }
}
```

The *GlgObject.LoadWidget* method loads a GLG drawing and returns the object ID of the viewport named $*Widget*. The *SetDrawingObject* method attaches the viewport to the GlgBean. Upon execution of these methods, the drawing is loaded and the hierarchy setup is complete. Its contents become accessible and it will be painted upon the reception of a paint event.

Initializing the Drawing

Often, an application needs to set or modify its objects' attributes before the drawing is drawn for the first time. The initial attributes of a drawing may be set after the drawing has been painted for the first time, but it would cause flickering since the drawing has to be repainted to reflect the new settings. To avoid this, the initial parameters may be set after the drawing is loaded but before it is initially drawn.

When a GLG drawing is loaded, the hierarchy of the GLG objects is set up before the drawing is displayed. GLG objects can contain any number of subobjects. For example, a GLG graph object has a particular number of datasamples and labels, as defined by the corresponding graph object's parameters. A hierarchy setup of a drawing containing a graph causes the graph's datasample and label instances to be created.

If the initial number of datasamples and labels needs to be set at run time, it must be done *before* the datasample and label instances are created, to avoid destruction and recreation of the new number of instances. On the other hand, filling the graph with initial data values and labels must be done *after* the hierarchy setup, when the datasample and label instances have already been created.

The number of datasamples and labels can also be changed dynamically at run time after the graph has been drawn. The Toolkit will automatically recreate the datasample and label instances to reflect the new settings.

To access GLG objects and their attributes at various stages of drawing initialization, an application may use the following **listeners** of the GlgBean**:**

H listener (***GlgHListener*** interface), invoked upon hierarchy setup, *after* the drawing is loaded, but *before* the hierarchy setup has occurred;

V listener (***GlgVListener*** interface), invoked *after* the drawing is loaded and *after* the hierarchy setup has occurred, but *before* the initial drawing;

Ready listener (***GlgReadyListener*** interface), invoked *after* the drawing is ready, i.e. it is loaded, set up and painted for the first time;

The GlgBean also provides listeners for handling user interaction and system events:

Input listener (***GlgInputListener*** interface), invoked upon user interaction, such as object selection and interaction with the GLG control widgets;

Select listener (***GlgSelectListener*** interface), invoked when named objects in the drawing are selected with the mouse;

Trace listener (***GlgTraceListener*** interface), invoked for all system events occurring in a GlgBean and commonly used as an escape mechanism for handling low-level events.

A **Hierarchy** listener is also provided for handling complex subdrawing and subwindow logic.

All listeners should be added to a GlgBean *before* a drawing name is set. ***H, V*** and ***Ready*** listeners are used to set the drawing's initial parameters and discussed in this chapter. The use of the ***Input***, ***Select*** and ***Trace*** listeners is discussed in the *Handling User Interaction* chapter.

To add a listener to a GlgBean, the GlgBean's ***AddListener*** method should be used:

```
glg_bean.AddListener( listener_type, Object listener );
```

The second argument to the *AddListener* method is a class which implements the corresponding interface and overrides the interface's respective method to create custom application code. For example, the listener class *HListener* implements the *GlgHListener* interface and provides an implementation of the *H* callback method to set a drawing's initial attributes.

The following code sample illustrates the addition of *H*, *V* and *Ready* listeners to a GlgBean and definition of corresponding interfaces:

```
public class GlgGraphComponent extends JApplet implements
      ActionListener
{
   GlgJBean glg_bean;
   boolean IsReady = false;

   //Constructor
   public GlgGraphComponent()
   {
      setLayout( new GridLayout( 1, 1 ) );
      glg_bean = new GlgJBean();
      add( glg_bean );

      // Add Hierarchy Setup, Value and Ready Listeners.
      glg_bean.AddListener( GlgObject.H_CB, new HListener() );
      glg_bean.AddListener( GlgObject.V_CB, new VListener() );
      glg_bean.AddListener( GlgObject.READY_CB, new ReadyListener() );
   }

   // Invoked by the browser to start the applet
   public void start()
   {
      super.start();
      glg_bean.SetParentApplet( this );
      glg_bean.SetDrawingName( "bar1.g" );
      //...
   }

   // Invoked by the browser to stop the applet.
   public void stop()
   {
      // Unset the drawing
      glg_bean.SetDrawingName( null );
      IsReady = false;
      super.stop();
   }

   // Define a class HListener which implements GlgHListener interface.
   // This class should provide an implementation of the HCallback
   // method.
   class HListener implements GlgHListener
   {
```

```
            public void HCallback( GlgObject viewport )
            {
               // Set initial graph parameters, including titles, datasample
               // and label parameters.

               viewport.SetSResource( "Title/String", "Bar Graph" );
               viewport.SetSResource( "XAxisLabel/String", "Sample" );
               viewport.SetSResource( "YAxisLabel/String", "Value" );

               // Set initial DataSample value
               viewport.SetDResource( "DataGroup/DataSample/Value", 0. );

               // Set number of data samples
               viewport.SetDResource( "DataGroup/Factor", 20. );
            }
         }

         // Define a class VListener which implements GlgVListener interface
         // and its VCallback() method.
         class VListener implements GlgVListener
         {
            public void VCallback( GlgObject viewport )
            {
               // This callback may be used to fill graph with data for the
               // initial appearance.
            }
         }

         // Define a class ReadyListener which implements GlgReadyListener
         // interface. This class should provide an implementation of the
         //  ReadyCallback method.
         class ReadyListener implements GlgReadyListener
         {
            public void ReadyCallback( GlgObject viewport )
            {
               // Enable dynamic updates
               IsReady = true;

               //Place custom initialization code here
            }
         }
      }
```

Refer to the Java examples in <*glg_dir*>*/examples_java* directory for the complete source code which illustrates the usage of the GlgBean's listeners.

During loading time, listeners, such as the *H* and *V* listeners, are invoked if they exist. Therefore, as a rule of thumb, all listeners should be added to a GlgBean *before* a drawing name is set, because this causes the drawing to be loaded.

The GlgBean implements all GLG listener interfaces and installs itself as a default listener. Therefore, when a GlgBean is subclassed in the application, its callback methods (*H* callback, *V* callback, *Ready* callback, etc.) may be overridden and used directly without creating separate

classes for the listeners.The GlgGraphApplet and GlgSubclassExample examples, located in the *<glg_dir>/examples_java* directory, illustrate how to use callbacks when the GlgBean is subclassed.

The *H* listener, *V* listener and *Ready* listener can be added only to the GlgBean, while the *Input*, *Select* and *Trace* listeners may also be added to the individual children viewports of the drawing. Only one listener of each type may be installed for each bean or viewport. Installation of a second listener of the same type will replace the first one.

Handling Dynamic Updates

Dynamic attributes of GLG objects are updated via a unified resource mechanism, which allows any parameter of the drawing to be accessed as a resource name containing the full path of the attribute in the drawing's object hierarchy. Therefore, all dynamic updates are handled by setting the drawing's resources using a few basic GLG methods:

SetDResource to set a double type resource;

SetSResource to set a string type resource;

SetGResource to set a geometric type resource, such as *FillColor*, or *EdgeColor* or the coordinates of a point;

Update to update the drawing to reflect new resource values.

For example, to push a new data value into a graph, the *SetDResource* method is used:

```
SetDResource( "DataGroup/EntryPoint", data_value );
```

where

data_value - a double value for the current iteration.

To set a label string for the graph's axis labels, *SetSResource* is used:

```
SetSResource( "XLabelGroup/EntryPoint", new_label_string );
```

where

new_label_string - a String object whose value is used to set a label string for the current iteration.

To set the EdgeColor of the graph's title, *SetGResource* is used:

```
// Set color of a title string to red
SetGResource( "Title/EdgeColor", 1., 0., 0. );
```

For details regarding objects' resources and their usage, refer to the *Objects* chapter on page 31 of the *GLG User's Guide and Builder Reference Manual.*

To update a GLG drawing with new data values, an application should periodically supply new values for dynamic resources of the drawing and repaint the drawing to reflect the changes.

Below is an example of an *UpdateGraphProc* method which updates a GLG graph to reflect new data values. In this method, the *SetDResource* and *SetSResource* methods are used to set new values for the graph's data samples and labels, and *Update* method is used to repaint the drawing to make new resource settings visible.

```
void UpdateGraphProc()
{
   if( !IsReady )
       return;

   // Push next data value
   glg_bean.SetDResource( "DataGroup/EntryPoint", GetData() );

   // Push next label glg_bean.
   SetSResource( "XLabelGroup/EntryPoint", GetLabel() );

   // Make changes visible
   glg_bean.Update();
}
```

In this example, the *GetData* method returns a random data value, and the *GetLabel* method returns the next label for the X axis. In a real world application, a user would provide custom methods to supply data values, label strings and attribute values that need to be dynamically updated. The data may come from a file, database, socket or any other source. The Toolkit provides an open data API which works equally well with a variety of data sources.

An update procedure is invoked periodically by an application via a timer. Swing does not support asynchronous updates and is *not* thread-safe, and the use of timers ensures synchronous updates from the event thread.

Example of Handling Dynamic Updates

The following code uses a timer to invoke the *UpdateGraphProc* method periodically to handle data updates:

```
public class GlgGraphComponent extends JApplet implements
      ActionListener
{
   GlgJLWBean glg_bean;
   Timer timer = null;
   boolean IsReady = false;

   // Constructor, where Glg bean component is created and added to a
   // native Java container, an Applet in this case.
   public GlgGraphComponent()
   {
      getContentPane().setLayout( new GridLayout( 1, 1 ) );
      glg_bean = new GlgJLWBean();
      getContentPane().add( glg_bean );
```

```
   // Add Ready Listener.
   glg_bean.AddListener( GlgObject.READY_CB, new ReadyListener() );
}

// Invoked by the browser to start the applet
public void start()
{
   super.start();
   glg_bean.SetParentApplet( this );
   glg_bean.SetDrawingName( "bar1.g" );
   StartUpdates();
}

// Invoked by the browser to stop the applet.
public void stop()
{
   // Unset the drawing
   glg_bean.SetDrawingName( null );
   StopUpdates();
   IsReady = false;
   super.stop();
}

// Define a class ReadyListener which implements GlgReadyListener
// interface. This class should provide an implementation of the
//  ReadyCallback method.
class ReadyListener implements GlgReadyListener
{
   public void ReadyCallback( GlgObject viewport )
   {
      // Enable dynamic updates
      IsReady = true;

      //Place custom initialization code here
   }
}

// Start periodic updates.
void StartUpdates()
{
   if( timer == null )
   {
       Timer timer = new Timer( 30, this );
       timer.setRepeats( true );  timer.start();
   }
}

// Stop updates.
void StopUpdates()
{
   if( timer != null )
   {
      timer.stop();
      timer = null;
   }
```

```
    }

    public void actionPerformed( ActionEvent e )
    {
       UpdateGraphProc();
    }

    // UpdateGraphProc mthod is invoked periodically by a timer.
    void UpdateGraphProc()
    {
       if( IsReady )
          return;

       // Push next data value
       glg_bean.SetDResource( "DataGroup/EntryPoint", GetData() );

       // Push next label glg_bean.
       SetSResource( "XLabelGroup/EntryPoint", GetLabel() );

       // Make changes visible
       glg_bean.Update();
    }
}
```

In this example, the *StartUpdates* method creates a *timer* object with a time interval of 30 milliseconds. The *GlgGraphComponent* class implements *ActionListener* interface and is passed to the timer as an *ActionListener* parameter. When the given time interval elapses, the timer invokes the *actionPerformed* method of the applet, invoking the *UpdateGraphProc* method and causing the drawing or be updated every 30 msec.

As mentioned earlier, Swing is not thread-safe, which means that GLG dynamic updates must be performed in the main thread. Using timer objects is the preferred method of implementing periodic updates. If an application is threaded, the threads should not invoke any GLG methods.

Complete source code for all GLG Java examples can be found in the *<glg_dir>/examples_java* directory. The source code of all GLG Java demos is located in the *<glg_dir>/DEMOS/java_demos/java2* directory.

Handling User Interaction

Overview

User interaction in the GlgBean, including mouse clicks, mouse motion and keyboard events, causes input events to be generated. These events are handled by *listeners* added to the GlgBean:

- **Input** listener, invoked upon any interaction with the GLG objects
- **Select** listener, invoked when a named GLG object is selected with the mouse

- **Trace** listener, invoked for all system events occurring in the GlgBean, including mouse clicks and mouse motion.

The **Input** listener is used to handle user interaction with input objects, such as buttons and sliders, process actions attached to objects in the drawing, as well as to process object selection.

The **Select** listener provides a simplified interface for processing the selection of named objects.

The **Trace** listener is used to trace low level system events, such as mouse motion events and locating the cursor for drag and drop functionality in the application.

To following sample code illustrates how to add *Select*, *Input* and *Trace* listeners to a GlgBean:

```
glg_bean.AddListener( GlgObject.INPUT_CB, new InputListener() );
glg_bean.AddListener( GlgObject.SELECT_CB, new SelectListener() );
glg_bean.AddListener( GlgObject.TRACE_CB, new TraceListener() );
```

Listener objects must implement the corresponding GLG interfaces: **GlgInputListener**, **GlgSelectListener** and **GlgTraceListener**. Each listener provides a callback method which will be invoked when the corresponding event occurs in a GlgBean. The following code shows stubs of the listener objects and their callback methods:

```
class InputListener implements GlgInputListener
{
   public void InputCallback( GlgObject viewport, GlgObject
      message_obj )
   {
   }
}

class SelectListener implements GlgSelectListener
{
   public void SelectCallback( GlgObject vp, Object[] name_array, int
      button )
   {
   }
}

class TraceListener implements GlgTraceListener
{
   public void TraceCallback( GlgObject viewport, GlgTraceData
      trace_info )
   {
   }
}
```

Input, *Select* and *Trace* listeners may be added to the GlgBean, or individual child viewports of the drawing.

The GlgBean provides a default implementation for all GLG listener interfaces and installs itself as a default listener. Therefore, when a GlgBean is subclassed in the application, its callback methods may be overridden and used directly without listener registration. Refer to the GlgSubclassExample located in the *<glg_dir>/examples_java* directory

This chapter covers the details of using *Select* and *Input* callbacks. For examples of using a *Trace* callback, refer to the GlgDiagramDemo and GlgMapDemo demos in the *<glg_dir>/DEMOS/java_demos* directory.

The **Input** callback is invoked with the following arguments:

```
public void InputCallback( GlgObject viewport, GlgObject message_obj )
```

where

viewport - a GLG viewport object to which an *Input* listener is attached. If an *Input* listener is added to a GlgBean, the viewport parameter will be the bean's top level viewport named $*Widget*.

message_obj - a GLG *message* object containing detailed information regarding the occurred event.

Resources of the **message object** may be used to retrieve information about the occurred event. The following resources are always present in the *message* object regardless of the type of event that triggered it:

Format - Defines the format of the input handler which triggered the event. It may have values such as *Button*, *Slider*, *Knob*, *CustomEvent*, *ObjectSelection*, etc.

Origin - Contains the name of the viewport the event occurred in or the name of the selected object.

FullOrigin - Full pathname of the viewport or of the selected object.

Action - Describes the occurred action. Action values depend on the *Format* resource and may have values such as *ValueChanged*, *Activate*, *MouseClick*, *MouseOver* and so on.

SubAction - Provides additional information about the event.

The rest of the *message* object's resources depend on the event type and are described in the *Handling Input Events in a GLG Control Widget* section and the *Handling Object Selection* section.

The *Select* callback is invoked with the following arguments:

```
public void SelectCallback( GlgObject vp, Object[] name_array,
                            int button )
```

where

vp - the viewport object the *Select* callback is attached to;

name_array - the array of the object names of all selected objects. This array can be traversed to find a particular object name;

button - the mouse button which initiated the event

Handling Input Events in a GLG Control Widget

There are two ways to handle user interaction with a GLG input object, such as a button, a toggle or a slider:

- using low-level input events passed to the Input callback
- using integrated input actions attached to input objects at design time using the Enterprise Edition of the GLG Builder or the HIM Configurator. At run time, the actions are triggered by a specific input activity specified by the action's parameters.

Using Low-Level Input Events

To handle input events occurred in a GLG control widget, a *message* object passed as a parameter to the *Input* callback should be used to retrieve the following information:

- the type of the control (*Slider*, *Knob, Button*, etc.), defined by the *Format* resource
- the name of the widget, defined by the *Origin* resource
- the event action type, which depends on the *Format* resource and is defined by the *Action* resource
- the resources relevant to a particular control type with a particular *Interaction Handler*, such as the *Value* resource for a dial, *ValueY* or *ValueX* resources for a slider and so on.

The following Swing based sample code illustrates how to handle occurred input events in a *Slider* and *Button* controls. A GlgJLWBean is added to the *JApplet* and displays the GLG drawing named "*controls.g*", which is composed of a dial, slider and button GLG controls. The events coming from the controls are handled in the *Input* callback.

```
public class GlgControlPanel extends JApplet
{
   GlgJLWBean glg_bean;
   static boolean IsStandalone = false;

   // Constructor.
   public GlgControlPanel()
   {
      getContentPane().setLayout( new GridLayout( 1, 1 ) );

      glg_bean = new GlgJLWBean();
      getContentPane().add( glg_bean );

      // Add InputListener to handle user interaction.
      glg_bean.AddListener( GlgObject.INPUT_CB, new
      InputListener(this) );
   }

   //Invoked by a browser to start the applet
   public void start()
   {
      super.start();

      // Set parent applet for the GlgBean and set GLG
```

```
   // drawing to be displayed in the bean.
   glg_bean.SetParentApplet( this );
   glg_bean.SetDrawingName( "controls.g" );
}

// Invoked by the browser to stop the applet
public void stop()
{
   // Unset the drawing.
   glg_bean.SetDrawingName( null );
   super.stop();
}

 // Define a class InputListener.
class InputListener implements GlgInputListener
{
   GlgControlPanel parent;

   public InputListener( GlgControlPanel parent_p )
   {
      parent = parent_p;
   }

   public void InputCallback( GlgObject viewport,
                              GlgObject message_obj )
   {
      String origin, format, action, subaction;

      origin = message_obj.GetSResource( "Origin" );
      format = message_obj.GetSResource( "Format" );
      action = message_obj.GetSResource( "Action" );
      subaction = message_obj.GetSResource( "SubAction" );

      // Input event occurred in a button
      if( format.equals( "Button" ) )
      {
         if( !action.equals( "Activate" ) )
            return;

        if( origin.equals( "QuitButton" ) )
        {
           if( parent.IsStandalone )
             System.exit( 0 );
           else
             parent.stop();
         }
      }

      // Input event occurred in a slider.
      if( format.equals( "Slider" ) )
      {
         double slider_value;

         // Retreive a current slider value from a message object
         slider_value =
```

```
                message_obj.GetDResource( "ValueY" ).doubleValue();

                // Set a data value for a dial control
                viewport.SetDResource( "Dial/Value", slider_value );
             }

             // Update the viewport to reflect new resource settings.
             viewport.Update();
          }
       }
    }
```

In the above example, the *Format* resource retrieved from the *message* object is used to determine the type of control the event occurred in. The *Format* represents the type of control, such as *Slider* for a slider control, and *Button* for a button control. The type of the *Interaction Handler* of a control may be found in the GLG Builder in the *Properties* dialog for that control in the *Handler* field. A complete list of the various GLG *Interaction Handlers* may be found in the *Message Object* section on page 116 of the *GLG Programming Reference Manual*.

The *Origin* resource is used to determine the name of the control which caused an input event to occur. If necessary, the *FullOrigin* resource may be used to retrieve the full pathname.

In this example, if the event occurred in a slider, the slider's value retrieved from the *message* object is reflected in a dial. For a vertical slider, the slider's value is represented by the *ValueY* resource. The value of a dial is represented by the *Dial*/*Value* resource, where *Dial* is the viewport name of the dial control.

For complete source code, refer to the GlgControlPanel and GlgControlBean examples, located in *<glg_dir>/examples_java* directory.

Using Integrated Input Actions

When an input action attached to a GLG input object is triggered at run time, the *Input* callback is invoked with a *message* object that contains information about the input object and the action that triggered the callback. The code sample below illustrates how to process input actions of the *command* type in the *Input* callback. A more elaborate example of handling input actions with complete source code may be found in the *GlgSCADAViewer* example located in the *<glg_dir>/SCADA_Viewer/Java* directory.

The code that processes commands can process commands from both input objects as well as commands generated on object selection, and the code below is identical to the command processing code in the *Using Input Callback to Handle Object Selection via Integrated Actions* section on page 31.

```
class InputListener implements GlgInputListener
{
   public void InputCallback( GlgObject viewport,
                              GlgObject message_obj )
   {
      String origin, format, action, subaction;
      String selected_object_name;
```

```
origin = message_obj.GetSResource( "Origin" );
format = message_obj.GetSResource( "Format" );
action = message_obj.GetSResource( "Action" );
subaction = message_obj.GetSResource( "SubAction" );

if( format.equals( "Command" ) )
{
   // This code handles command actions attached to anobject.

   // Retrive command information using the Standard API.
   String command_type = message_obj.
      GetSResource( "ActionObject/Command/CommandType" );
   String event_label =
      message_obj.GetSResource( "EventLabel" );
   String selected_object_name =
      message_obj.GetSResource( "Object/Name" );

   /* Retrieve the object IDs of the command and the selected
      object using the Intermediate API. */
   GlgObject selected_object =
      message_obj.GetResourceObject( "Object" );
   GlgObject command = message_obj.
      GetResourceObject( "ActionObject/Command" );

   ExecuteCommand( selected_object, command_type, command );
}
```

Handling Object Selection

The selection of an object occurs when the user clicks on a GLG object with the mouse or moves the mouse over it. There are several ways to handle object selection in a GLG Java application:

- **Select** callback may only be used to process object selection of *named* objects clicked with the mouse. The *Select* callback provides a simplified interface for handling object selection and is invoked with a list of names of all objects selected. This list can be traversed to retrieve individual object names.

- **Input** callback may be used to handle the **low-level object selection** on either mouse move or mouse click events using **object IDs**. This method requires the use of the GLG Intermediate API.

  A *message* object passed to the *Input* callback will contain an array of all objects selected with the mouse. This array can be traversed to retrieve individual object IDs. The array contains objects on the lowest level of the object hierarchy. For example, if an object group is selected, subobjects at the cursor position will be reported instead of the group object itself. The *GetParent* method can be used to obtain the group's ID.

- **Input** callback may be used to process **commands** or **custom events** added to objects in a drawing using integrated actions. A command or custom event action may be added to an object using the Enterprise Edition of the Builder or the HMI Configurator. The action parameters specify whether the action is activated on *MouseClick* or *MouseOver* and supply additional data to execute associated command or to process the custom event.

  When an object's action is triggered by *MouseClick* or *MouseOver* at run-time, the *Input* callback will be invoked with a *message* object containing the action and event information, including the object selected by the mouse event and an command associated with the action.

  This method is convenient for handling selection of complex objects. For example, an application may display node objects implemented as a group containing a text label and an icon. The application might need to handle a selection event when the user selects either the label or the icon object with the mouse. This can be easily achieved by adding a *MouseClick* action for the node object in the Builder and processing this event in the *Input* callback.

  An application may also need to differentiate between various types of selection, such as node and link selection. This may be achieved by using an action's event label set to "NodeSelection" or "LinkSelection" depending on the object type. In the *Input* callback, an application would retrieve the *EventLabel* resource of the *message* object and perform different actions depending on the *EventLabel* value. Alternatively, an command action with a custom command may also be used, with *CommandType* set to "NodeSelection" or "LinkSelection".The *GlgMapDemo* located in the *DEMOS/java_demos/java2* directory of the GLG installation provides an elaborate example of handling node and link selection using custom selection events.

  C*ustom events* and *command* actions may be added to objects in a drawing using either the Enterprise Edition of the GLG Builder or the HMI Configurator. Refer to the *Custom Object Selection Events and Commands* section on page 59 of the *GLG User's Guide and Builder Reference Manual* for details.

In order to receive selection events in the input listener, the **ProcessMouse** attribute of a viewport must include one of both of the *Move* and *Click* masks to enable selection events on *MouseOver* or *MouseClick*, correspondingly.

Details and code samples of different methods of handling object selection are provided below.

Using Select Callback

The following code sample illustrates the use of the *Select* callback to handle the selection of objects named *SolventValve* and *SteamValve*. The complete source code may be found in the *GlgProcessDemo* demo located in the *<glg_dir>/DEMOS/java_demos/java2* directory.

```
public void SelectCallback( GlgObject vp, Object[] name_array,
                            int button )
{
   //...
   if( name_array == null )
      return;
```

```
      for( int i=0; ( name = (String) name_array[i] ) != null; ++i )
      {
         if( name.equals( "SolventValve" ) ||
             name.equals( "SteamValve") )
         {
            if( button == 1 )
               OpenValve( name );
            else
               CloseValve( name );
         }
      }
   }
```

One advantage of using the *Select* callback is its simple interface. However, when dealing with complex objects such as nodes which contain a collection of objects, using the *Select* callback may become tedious, since it involves string parsing. It also requires unique object names.

To handle the selection of complex objects, lack of or collision of object names, or when complete control over object selection is required, the Input callback should be used.

Using Input Callback to Handle Object Selection via Integrated Actions

The following code sample illustrates the use of integrated actions attached to an object to handle the object selection. In order for action events to be received in the *Input* callback, either the viewport containing these objects or any parent viewport must have the *ProcessMouse* flag set to process mouse click and/or mouse move events.

When an action event occurs, the *Input* callback is invoked with the *message* object parameter that contains information about the selected object and the action that triggered the callback. The code sample below illustrates how to process both *command* and *custom event* actions in the *Input* callback. A more elaborate example of custom event actions with complete source code may be found in the *GlgObjectSelection* example located in the *<glg_dir>/examples_java/ObjectSelection* directory. The *GlgSCADAViewer* example located in the *<glg_dir>/SCADA_Viewer/Java* directory provides an example of handling *command* actions attached to objects.

```
public class GlgMouseActionsExample extends JApplet
{
   GlgJLWBean glg_bean;

   // Constructor.
   public GlgMouseActionsExample()
   {
      glg_bean = new GlgJLWBean();
      getContentPane().add( glg_bean );

      // Add InputListener to the GlgBean.
      glg_bean.AddListener( GlgObject.INPUT_CB, new InputListener() );
   }

   // Invoked by a browser to start the applet
   public void start()
   {
      super.start();
```

```
      glg_bean.SetParentApplet( this );
      glg_bean.SetDrawingName( "obj_selection.g" );
   }

   // Define a class InputListener.
   class InputListener implements GlgInputListener
   {
      public void InputCallback( GlgObject viewport,
                                 GlgObject message_obj )
      {
         String origin, format, action, subaction;
         String selected_object_name;

         origin = message_obj.GetSResource( "Origin" );
         format = message_obj.GetSResource( "Format" );
         action = message_obj.GetSResource( "Action" );
         subaction = message_obj.GetSResource( "SubAction" );

         if( format.equals( "Command" ) )
         {
            /* This code handles command actions attached to an
               object. */

            // Retrive command information using the Standard API.
            String command_type = message_obj.
               GetSResource( "ActionObject/Command/CommandType" );
            String event_label =
               message_obj.GetSResource( "EventLabel" );
            String selected_object_name =
               message_obj.GetSResource( "Object/Name" );

            /* Retrieve the object IDs of the command and the selected
               object using the Intermediate API. */
            GlgObject selected_object =
               message_obj.GetResourceObject( "Object" );
            GlgObject command = message_obj.
               GetResourceObject( "ActionObject/Command" );

            ExecuteCommand( selected_object, command_type, command );
         }
         else if( format.equals( "CustomEvent" ) )
         {
            /* This code handles custom events attached toan object.
               This code handles both custom event actions and the
               old-style custom events. */

            String event_label =
               message_obj.GetSResource( "EventLabel" );

            /* Don't process events with empty EventLabel.An event
               with an empty EventLabel is generated for MouseOver
               events when the mouse moves away from the object, and
               for MouseClick events when the mouse button is released.
               */
            if( event_label.equals( "" ) )
```

```
                        return;

                    // Retrieve command and selected object (Intermediate
                    // API).
                    GlgObject selected_object =
                       message_obj.GetResourceObject( "Object" );

                    // Retrieve the selected object's name using Standard API.
                    selected_object_name =
                       message_obj.GetSResource( "Object/Name" );

                    /* For the old-style MouseClick actions, query the mouse
                       button that generated this custom selection event (0
                       for custom MouseOver events). */
                    int button_index =
                       message_obj.GetDResource( "ButtonIndex" ).intValue();

                    // Prints MouseClick or MouseOver.
                    System.out.println( "Custom event action: ", action );
                    System.out.println( "Selected object: ", name );

                    /* Retrieve ActionObject resources of the message object
                       (Intermediate API). ActionObject will be NULL for
                       custom events added prior to GLG v.3.5. */
                    GlgObject action_obj =
                       message_obj.GetResourceObject( "ActionObject" );

                    /* Retrieve the action's ActionData. If present, its
                       properties may be used for processing the event. */
                    if( action_obj != null )
                       GlgObject action_data =
                          action_obj.GetResourceObject( "ActionData" );
                }
            }
        }
    }
```

Using Input Callback to Handle Low-Level Object Selection using Object IDs

The *Input* callback may be used to handle object selection via the retrieved *SelectionArray* resource from the *message* object. *SelectionArray* contains a list of all selected objects and may be traversed to get individual object IDs. This functionality requires the use of the Extended API.

When any GLG object is selected with the mouse and the *ProcessMouse* flag of a viewport is set to *Click* or *Click & Move*, the *Input* callback is invoked with the *message* object whose *Format* resource is set to *ObjectSelection*.

If the *ProcessMouse* flag is set to *Click*, the *Input* callback is invoked only for mouse click events. If *ProcessMouse* flag is set *Click & Move*, the *Input* callback is invoked for both mouse move and mouse click events. The value of the *Action* resource of the *message* object will be set to *MouseClick* or *MouseMove* for a mouse click or a mouse move event respectively.

The example below illustrates how to use the *ObjectSelection* event in the *Input* callback and traverse the *SelectionArray* array to obtain object IDs of selected objects. The complete source code may be found in GlgObjectSelection example located in the *<glg_dir>/examples_java/ObjectSelection* directory.

```
public void InputCallback( GlgObject viewport, GlgObject message_obj )
{
   String origin = message_obj.GetSResource( "Origin" );
   String format = message_obj.GetSResource( "Format" );
   String action = message_obj.GetSResource( "Action" );
   String subaction = message_obj.GetSResource( "SubAction" );

   if( format.equals( "ObjectSelection" ) )
   {
      GlgObject selection_array =
         message_obj.GetResourceObject( "SelectionArray" );

      if( selection_array != null )
      {
         int size = selection_array.GetSize();
         for( int i=0; i<size; ++i )
         {
            GlgObject object =
               (GlgObject) selection_array.GetElement( i );
            String obj_name = object.GetSResource( "Name" );

            // Process object selection here. Custom properties
            // attached to the object may be used to provide additional
            // information.
            // .....
         }
      }
   }
}
```

2. Using GLG Components in Java

GLG Bean Classes Used to Display Graphics

GlgJBean is a *heavyweight Swing*-based Java bean that can be used as a top-level container for other Swing components. **GlgJLWBean** is a *lightweight Swing*-based Java bean. **GlgBean** is an *AWT* based Java bean component.

If a Swing application creates children of a GLG bean, such as a menu or dialog popup, *GlgJBean* or *GlgJLWBean* should be used to display the graphics rather than *GlgBean*, since Swing objects can be added only to a Swing-based component.

Performance Optimization

Lightweight Swing components do not have a native window, and using them to display graphics may yield slower update rates due to the damage repair that takes place to redraw intersecting lightweight components. Lightweight Swing objects are designed for simple interface objects, such as buttons and menus. However, for displaying dynamic graphics, it is more efficient to use *heavyweight* components, such as *GlgJBean* and *GlgBean*, instead of *GlgJLWBean*.

Mixing Heavyweight and Lightweight Components in a Swing Application

Since *lightweight* components do not have a native peer, they cannot be displayed in front of *heavyweight* components. Therefore, if various components intersect, all of them must be either heavyweight or lightweight.

For example, if a Swing application creates a *heavyweight* component, such as *GlgJBean*, to display graphics, then a *popup menu* created to appear in front of it must also be *heavyweight*.

To make a particular popup menu object a *heavyweight* component, the *JPopupMenu.setLightWeightPopupEnabled(false)* method can be used.

To make all menus in the application *heavyweight* components, the static method *setDefaultLightWeightPopupEnabled(false)* can be used.

Using Java Applet in a Browser

By default, browsers do not have the Java virtual machine installed. In order to run a Java applet in a browser, *Java Plug-in* should be installed and enabled in the client machine's browser.

Using Timers for Dynamic Updates

Swing does not support asynchronous updates and is *not thread-safe*. Timers must be used to update graphics synchronously on the event thread.