IBM ILOG JViews Framework V8.6

IBM ILOG JViews Framework Essential Features

© Copyright International Business Machines Corporation 1987, 2009
US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, WebSphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at *http://www.ibm.com/legal/copytrade.shtml*

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further copyright information see ***<installdir>*** **`/license/notices.txt`**.

Table of contents

Introducing IBM® ILOG® JViews Framework....7
Overview....8
A basic graphics application model....10
Getting started with JViews Framework....13
Overview of tutorial....14
Running the example....15
Stage 1 - The manager....16
Stage 2 - View interaction....19
Stage 3 - Using events....21
Stage 4 - Manipulating graphic objects....25
Framework JavaBeans(TM)....29
Installing IBM® ILOG® JViews Beans in an IDE....30
Framework classes available as JavaBeans(TM)....31
Creating a simple applet using IBM® ILOG® JViews Beans....35
Graphic objects....45
A graphic object....47
The class IlvGraphic....48
Hierarchy of predefined graphic objects....49

© Copyright IBM Corp. 1987, 2009

Geometric properties....50
User properties of graphic objects....55
Input/output operations....56
The graphic bag....58
Predefined graphic objects....59
The ShadowEllipse class....66
Creating a new graphic object class....67
Testing for a point inside an object....74
Saving and loading the object description....75
Named properties....77
Managers....79
A manager....81
A manager view....82
Layers....83
Handling input events: interactors and accelerators....84
Input/output....85
Class diagram for IlvManager....86
Multiple manager views....87
Binding views to a manager....89
Creating a manager and a view....90
Listener for the views of a manager....91
View transformation....92
Scrolled manager view....94
Managing double buffering....95
The manager view grid....96
Class diagram for IlvManagerView....97
Manager view repaint skipper....98
Managing layers....99
Layers in a manager....100
Setting up layers....101
Layers and their graphic objects....102
Listener for layer changes in a manager....104
Triple buffering layers....105
Caching layers....107
Manipulating the drawing order....108
Managing graphic objects....111
Adding objects to a manager and removing them....112

Modifying geometric properties of objects....114
Applying functions....116
Editing and selecting properties....117
Optimizing drawing tasks....118
Listener for the content of the manager....120
Selection in a manager....123
Selection objects....124
Managing selected objects....125
Creating your own selection object....127
Listener for the selections in a manager....129
Hover highlighting in a manager....131
Managing hover highlighting....132
Creating your own highlighting effect....133
Blinking of graphic objects....135
Introduction....136
Managing input events....141
Handling input events....143
Object interactors....144
Example: Extending the IlvObjectInteractor class....145
Customizing the interactor of selected graphic objects....150
View interactors....152
Class diagrams for interactors and selection classes....154
Interactor listeners....156
The selection interactor....159
Tooltips and popup menus on graphic objects....162
Saving and reading....164
File formats....165
Drawing Exchange Format (DXF)....166
Graphers....169
The grapher....170
Managing nodes and links....171
Nodes....172
Links....173
Predefined link classes....174
Managing link visibility....177
Showing and hiding grapher branches....178
Contact points....179
Default contact points....180
Using link connectors....181
Using pins....182

Other link connectors....185
Class diagram for graphers....186
Grapher interactor class....187
Creating a new class of link....188
Link shapes and crossing....193
Composite Graphics....197
Introducing composite graphics....198
Creating a composite graphic....200
Index....205

Introducing IBM® ILOG® JViews Framework

Presents the purpose, contents, role, and features of IBM® ILOG® JViews Framework.

In this section

Overview
Describes the contents and use of the IBM® ILOG® JViews Framework package.

A basic graphics application model
Describes the basic object model, comprising the core Java™ objects and their interrelationships.

© Copyright IBM Corp. 1987, 2009

Overview

IBM® ILOG® JViews Framework is a structured 2D graphics package for creating highly customizable, visually rich graphical user interfaces. It complements the simple components provided by Swing or AWT, allowing Java™ GUI programmers to develop far more intuitive displays of information. Examples of such types of displays include schematics, workflow and process flow diagrams, command and control displays, network management displays, and any application requiring a map. The IBM® ILOG® JViews Framework (or the Framework, for short) is ideal for the rapid development of these custom GUIs.

The Framework package

The IBM® ILOG® JViews Framework package consists of a set of JavaBeans™ and an Application Programming Interface (API).

JavaBeans(TM)

The JavaBean components allow you to get a fast start with the Framework. You can start your favorite Integrated Development Environment (IDE), import the Beans, connect them to any other Beans, compile, and run. You can have a working example of an IBM® ILOG® JViews applet or application in just a few minutes. These Beans are an excellent beginning when learning the features of IBM® ILOG® JViews, and can be customized for delivery to your end users as well. See *Framework JavaBeans(TM)*.

API

Most applications will, however, require functionality that goes far beyond what these pre-packaged Beans offer, and this is why the Framework is delivered with an API. This API is a fully documented class library allowing you to build the custom look-and-feel that your application needs. Once you understand how the basic parts of the library work, you can customize or extend it. The architecture of the library is completely open to extension.

JViews Framework as IBM® ILOG® JViews foundation

JViews Framework package provides the base functionality for graphics applications built with IBM® ILOG® JViews products. It handles the creation and manipulation of basic graphic objects such as lines, rectangles, and labels, as well as any of the custom objects that you might create. JViews Framework also provides the optimized data structures that allow the graphic objects to be panned, zoomed, and selected with optimal performance. Finally, it provides a set of behaviors that can be used to define the user interactions with the display and the graphic objects. All the other IBM® ILOG® JViews packages rely on JViews Framework for these core-level services.

The essential features of JViews Framework are:

- Graphic objects that are drawn on the screen.
- Managers that handle collections of graphic objects.
- Graphers that are managers for graph structures, that is, nodes and links.

- Views that are lightweight components used to display managers and graphers.
- Interactors used to manipulate objects interactively.
- Composite graphics used to create graphic objects from other graphic objects.

A basic graphics application model

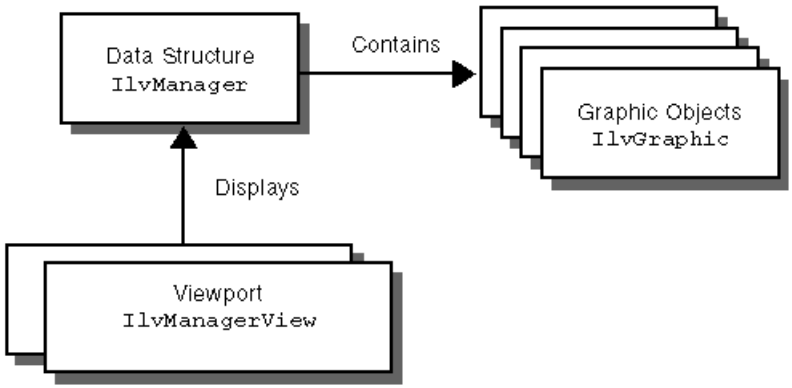
To use IBM® ILOG® JViews effectively, you need to understand how to use IBM® ILOG® JViews Framework, and to use the Framework, you need to understand the basic object model, that is, the core Java™ objects and their relationships with each other.

The Framework object model

A basic graphics application needs just a few parts:

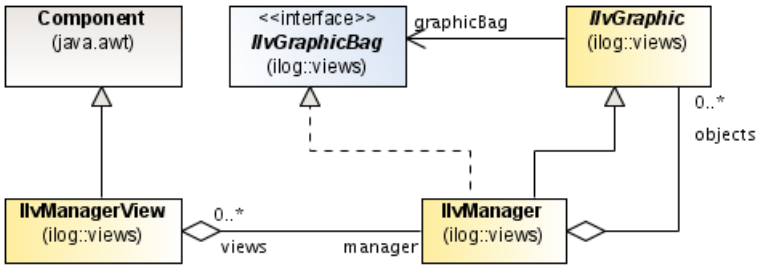
- Graphic objects (manipulated with resize, select, and draw functions)
- A data structure to put them into
- A viewport (typically a rectangular area in a window on the display) which allows zooming in and out on the graphic objects that are drawn in it

In the IBM® ILOG® JViews world, these parts are formally referred to as `IlvGraphic`, `IlvManager`, and `IlvManagerView` objects, respectively. Their organization is shown in the following figure.



Basic IBM® ILOG® JViews Classes

The following figure shows the same classes represented as a UML class diagram.



Basic Classes: UML Class Diagram

The graphic object: IlvGraphic

An `IlvGraphic` graphic object typically represents some custom entity in the end-user's application domain. For example, in a geographic display for a telecom network, there may be lines, labels, and polygons that form the background map and some other more sophisticated objects that represent the telecom devices in the network.

The graphics framework comes with a large set of predefined graphic objects, such as rectangles, polylines, polygons, splines, and labels. Other domain-specific objects (such as the network devices in the above example) can be created by subclassing one of these objects or the base class object, `IlvGraphic`, or by grouping predefined objects together.

The data structure: IlvManager

The `IlvManager` data structure is a container for graphic objects, therefore it is the most important object of the Framework. The manager organizes graphic objects into several layers; graphic objects contained in a higher level layer are displayed in front of objects located in a lower layer.

The manager also provides the means to select the graphic objects. The library comes with several predefined ways to display a selected graphic object; you can subclass `IlvManager` to create your own user-defined display methods.

The viewport: IlvManagerView

An `IlvManagerView` viewport is designed to visualize the graphic objects of a manager. The class `IlvManagerView` is a component (subclass of the *java.awt.Component*class) that you use in your AWT or Swing application to visualize the manager. A manager view lets you define which layer of the manager is visible for a view. In addition, you may use several manager views to visualize different areas of the manager. You can zoom and pan the content of the view.

Getting started with JViews Framework

Provides a tutorial explaining how to create a simple application using IBM® ILOG® JViews Framework and demonstrating its basic concepts.

In this section

Overview of tutorial
Lists the Java™ source files provided and the stages in the tutorial.

Running the example
Explains how to run example Java™ files.

Stage 1 - The manager
Shows how to create a manager and its view, and how to load a file containing graphic objects into the manager.

Stage 2 - View interaction
Shows how to add interaction to a view.

Stage 3 - Using events
Shows the use of events delivered by the view.

Stage 4 - Manipulating graphic objects
Addresses the manipulation of graphic objects, demonstrating how to create graphic objects and add them to the manager and how to change their location.

© Copyright IBM Corp. 1987, 2009

Overview of tutorial

The construction of an application explains briefly the main concepts of IBM® ILOG® JViews.

Example Java™ source files are provided representing the steps in the tutorial. The example files are as follows: `Sample1.java`, `Sample2.java`, `Sample3.java`, and `Sample4.java`.

The tutorial consists of the following stages:

1. Creating and populating the manager: in *Stage 1 - The manager*:
   - Creating the manager.
   - Loading a file containing graphic objects into this manager.
   - Creating the view to display the contents of the manager.
2. In *Stage 2 - View interaction*:
   - Adding interaction to a view.
3. In *Stage 3 - Using events*:
   - Listening to events sent by the manager view.
4. In *Stage 4 - Manipulating graphic objects*:
   - Adding and moving graphic objects in the manager.

Running the example

The samples are installed in subdirectories named `sample1`, `sample2`, `sample3`, and so on, located in the directory `/jviews-framework86/codefragments/getstart`. For details, see ***<installdir>*** **`/jviews-framework86/codefragments/getstart/index.html`**.

To run an example file such as `Sample1.java` (located at `codefragments/getstart/sample1/src/Sample1.java`):

1. Go to the `src` directory in the above path.
2. Set the `CLASSPATH` variable to include the current directory, the IBM® ILOG® JViews library: `jviews-framework-all` and the license file directory. On a Windows® machine this will be:

   ```
   set CLASSPATH=.;<installdir>/jviews-frameworknn/lib/jviews-framework-all.jar;<installdir>/jlm
   ```

   where `nn` is the version; for example, 86
3. Compile the `Sample1.java` file:

   ```
   javac Sample1.java
   ```
4. Run the application:

   ```
   java Sample1
   ```

Alternatively, you can compile and start the sample with the scripts available in

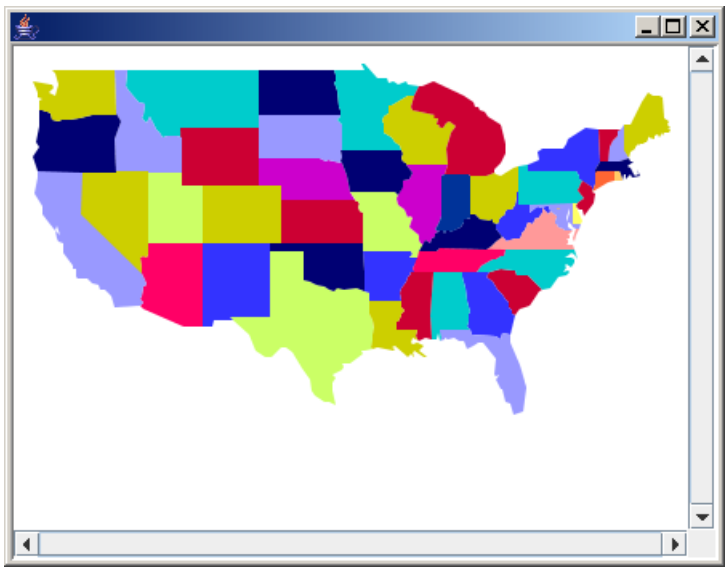
`/jviews-frameworknn/codefragments/getstart/sample1`.

Stage 1 - The manager

Overview of stage 1

The manager organizes sets of graphic objects into multiple views and layers and provides the possibility of higher-level interactions. These features are brought out as the tutorial progresses. When running the `Sample1` example, the first stage in this tutorial, you see a scrolling window displaying a map of the United States.

Explanations of the `Sample1.java` file follow.



Sample1

Importing the library and packages

The `Sample1.java` file first imports the main IBM® ILOG® JViews package and then imports the IBM® ILOG® JViews Swing package for the GUI components.

```
import ilog.views.*;
import ilog.views.swing.*;
```

To use AWT and Swing classes, the sample must import the `swing` and `awt` packages:

```
import javax.swing.*;
import java.awt.*;
```

Creating the frame class

After importing packages, you can create the class named `Sample1`. This class has two fields, `manager` (class `IlvManager`) to store the graphic objects, and `mgrview`, (class `IlvManagerView`), to display the contents of the manager.

```
public class Sample1 extends JFrame
{
  IlvManager manager;
  IlvManagerView mgrview;
  ....
}
```

Creating the manager

Use the constructor to create the manager:

```
...
  manager = new IlvManager();
   ...
}
```

Loading/reading a file

Once the manager is created, you can read the `usa.ivl` file which can be found in the `getstart` directory. IBM® ILOG® JViews Framework provides facilities to save and read graphic objects in a manager. These files are in the IVL format.

You need to catch the exception that may occur when reading the file. The method `read (java.net.URL)` of the class `IlvManager` may throw the following exceptions:

- `IOException` for basic IO errors.
- `IlvReadFileException`, if the format of the file is not correct (the file is not an `.ivl` formatted file) or if a graphic class needed to load the file cannot be found.

```
try {
  manager.read(new URL("usa.ivl");
} catch (Exception e) {}
```

Creating the view

Next create a manager view to display the contents of the manager. A manager view is an instance of the `IlvManagerView` class. To associate it with the manager, all you have to do is provide the `manager` parameter as shown below.

Note: This example uses the class `IlvJScrollManagerView`. This class encapsulates the class `IlvManagerView` and provides scroll bars.

```
mgrview = new IlvManagerView(manager);
getContentPane().setLayout(new BorderLayout(0,0));
getContentPane().add(new IlvJScrollManagerView(mgrview), BorderLayout.CENTER)
;
```

Testing the application

To test the application, you need the `jviews-framework-all.jar` file and the license file directory in your classpath. On a Windows® machine this will be:

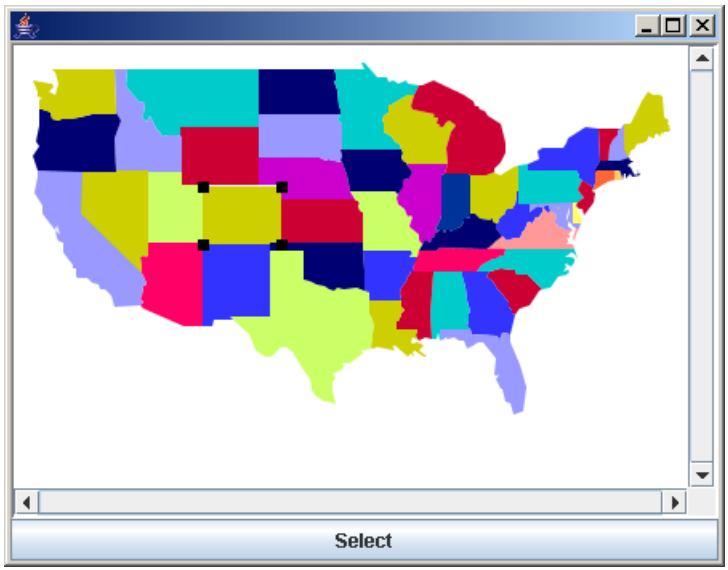
```
set CLASSPATH=.;<installdir>/jviews-framework86/lib/jviews-framework-
all.jar;<installdir>/jlm
```

The `jlm` license directory is only needed during the development phase. Once your application is ready for production, it can be deployed without the license file being needed in the `CLASSPATH`. For more information see the licensing documentation Using License Keys.

Stage 2 - View interaction

Overview of stage 2

The second part of the tutorial, the `Sample2.java` file, see **<*installdir*> / `jviews-framework86/codefragments/getstart/index.html`**, is an extension of the `Sample1` file. Compile the `Sample2.java` file and run it as you did for `Sample1`. See *Running the example*.



Sample2 Running

In this step, you add interaction to the view by placing a selection interactor on it. To do this, add a Select button and associate it with the interactor. When you click the Select button, the selection interactor is placed on the view and you can select the graphic objects in the view (in this case the states of the United States), move them around, and modify their shape.

A selection interactor is an instance of the class `IlvSelectInteractor`, a subclass of the `IlvManagerViewInteractor` class. This view interactor will process all the input events, such as mouse and keyboard events, occurring in a manager view.

Adding the selectInteractor field

To be able to use the class `IlvSelectInteractor`, first import the IBM® ILOG® JViews packages that contain the interactors, servlets and events:

```
import ilog.views.interactor.*;
import ilog.views.util.servlet.event.*;
```

```
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
```

Then add the `selectInteractor` and `button` instance variables into `Sample2`.

```
public class Sample2 extends JFrame
{
  IlvManager manager;
  IlvManagerView mgrview;
  IlvSelectInteractor selectInteractor;
  JButton button;
  ....
}
```

Creating the Select button

The following code creates a Select button and associates it with the `selectInteractor`:

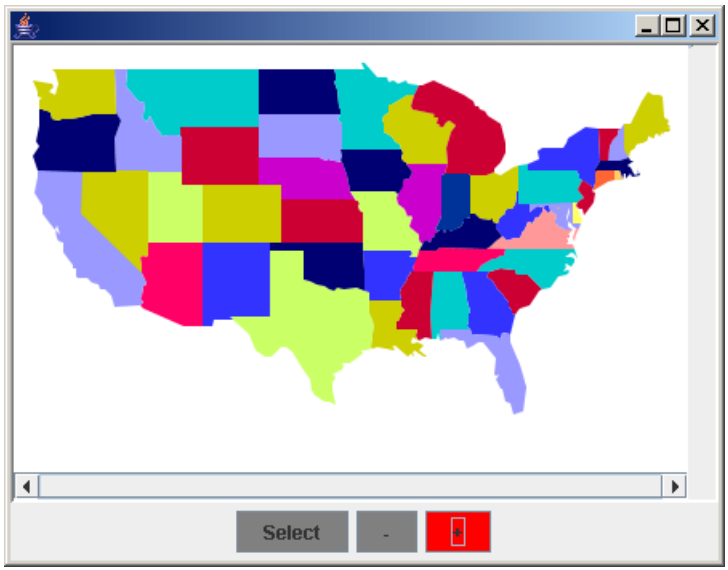
```
void createButtons()
{
  JButton button;
  button = new JButton("Select");
  button.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {
      if (selectInteractor == null)
         selectInteractor = new IlvSelectInteractor();
      if (mgrview.getInteractor() != selectInteractor)
         mgrview.setInteractor(selectInteractor);
    }
  });
  getContentPane().add(button, BorderLayout.SOUTH);
 }
```

When you click the Select button, the `actionPerformed` method first creates its interactor (if this has not already been done), then it installs the interactor on this view using the `setInteractor(ilog.views.IlvManagerViewInteractor)` method. Once the interactor is installed, you can select, move, and modify the graphics objects displayed in the view.

Stage 3 - Using events

Overview of stage 3

The third stage of the tutorial, the `Sample3.java` file, see **<*installdir*> / `jviews-framework86/codefragments/getstart/index.html`**, is an extension of the `Sample2` file. Compile the `Sample3.java` file and run it as you did for the previous example files. See *Running the example*.



Sample3 running

To make use of events, you can use the `InteractorListener` interface to listen for a change of interactors. There are three buttons in the example, each with an associated interactor. Clicking one button and then another changes the 'engaged' interactor accordingly.

Two new interactors are placed on the view: `IlvZoomViewInteractor` and the `IlvUnZoomViewInteractor`. These interactors allow you to drag a rectangle on the view to zoom in and out on this area. The third interactor is the `IlvSelectInteractor` (of `Sample2`). Their respective buttons are created inside a Swing JPanel, which automatically aligns them as seen in the above illustration.

Adding new interactor fields

To accomplish the task, change the class definition to implement `InteractorListener`, add the `zoomInteractor` and `unzoomInteractor` fields, and add the necessary interactor button fields to the `Sample3` application.

```
public class Sample3 extends JFrame
```

```
implements InteractorListener
{
  IlvManager manager;
  IlvManagerView mgrview;
  IlvSelectInteractor selectInteractor;
  IlvManagerViewInteractor zoomInteractor, unZoomInteractor;
  Button selectButton, zoomButton, unZoomButton;
  ....
}
```

Creating the interactor buttons

The `createInteractorButtons` method will create three buttons (Select, -, and +) that will be stored in the `selectButton`, `zoomButton`, and `unZoomButton` fields of the object.

Creating Interactor Buttons

```
void createInteractorButtons() {
  Panel buttons = new Panel();
  selectButton = new Button("Select");
  selectButton.setBackground(Color.gray);
  selectButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      if (selectInteractor == null)
        selectInteractor = new IlvSelectInteractor();
      if (mgrview.getInteractor() != selectInteractor)
          mgrview.setInteractor(selectInteractor);
      }
    }
  });

  buttons.add(selectButton);
  unZoomButton = new Button("-");
  unZoomButton.setBackground(Color.gray);
  unZoomButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      if (unZoomInteractor == null)
        unZoomInteractor = new IlvUnZoomViewInteractor();
      if (mgrview.getInteractor() != unZoomInteractor)
        mgrview.setInteractor(unZoomInteractor);
    }
  });

  buttons.add(unZoomButton);
  zoomButton = new Button("+");
  zoomButton.setBackground(Color.gray);
  zoomButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent evt) {
      if (zoomInteractor == null)
        zoomInteractor = new IlvZoomViewInteractor();
      if (mgrview.getInteractor() != zoomInteractor)
       mgrview.setInteractor(zoomInteractor);
    }
```

```
  });
  buttons.add(zoomButton);
  getContentPane().add(buttons, BorderLayout.SOUTH);
}
```

There are now three possible interactors, so the action performed when clicking a button removes the previously installed interactor and installs the new one by calling the `setInteractor(ilog.views.IlvManagerViewInteractor)` method of the `IlvManagerView` class.

Listening for a change of interactors

In the `Sample3.java` file, you can see that the class implements the interface `InteractorListener`. You may also have noticed the import of a new package, `package-summary`, which is the package that contains the IBM® ILOG® JViews event classes. The `InteractorListener` interface includes one method:

`interactorChanged(ilog.views.event.InteractorChangedEvent).`

In this example, the selected button becomes red when its corresponding interactor is attached to the view.

The `interactorChanged` method will be called when the interactor changes on the view (as soon as the object, the instance of `Sample3`, is registered as a listener; see *Registering the listener*). The parameter is an event that contains the old and the new interactor. You simply change the background color of the button corresponding to the newly installed interactor to red.

Changing the Color of an Interactor Button

```
public void interactorChanged(InteractorChangedEvent event)
{
     IlvManagerViewInteractor oldI = event.getOldValue();
     IlvManagerViewInteractor newI = event.getNewValue();

     if (oldI == selectInteractor)
       selectButton.setBackground(Color.gray);
     else if (oldI == zoomInteractor)
       zoomButton.setBackground(Color.gray);
     else if (oldI == unZoomInteractor)
       unZoomButton.setBackground(Color.gray);

     // there is no new interactor
     if (newI == null)
       return;
     if (newI == selectInteractor)
       selectButton.setBackground(Color.red);
     else if (newI == zoomInteractor)
       zoomButton.setBackground(Color.red);
     else if (newI == unZoomInteractor)
       unZoomButton.setBackground(Color.red);
}
```

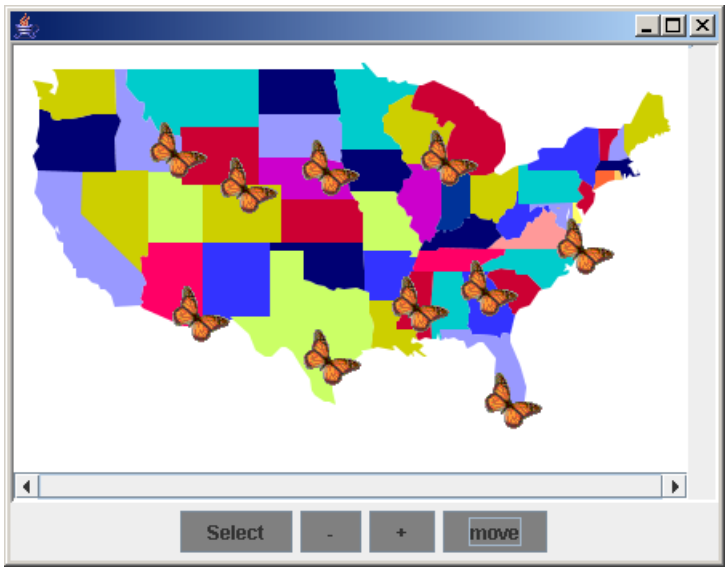
Registering the listener

It is not enough to implement the interface. You must not forget to register this new listener with the view. This is done by calling the `addInteractorListener` method in the `init` method.

```
...
    manager = new IlvManager();
    try {
      manager.read("usa.ivl");
    } catch (Exception e) {}
    mgrview = new IlvManagerView(manager);
    setLayout(new BorderLayout(0,0));
    getContentPane().add(new IlvJScrollManagerView(mgrview),
                         BorderLayout.CENTER);
    createButtons();
    mgrview.addInteractorListener(this);
 ...
```

Stage 4 - Manipulating graphic objects

The fourth step, the `Sample4.java` file, see **<*installdir*> `/jviews-framework86/codefragments/getstart/index.html`**, is an extension of the `Sample3` file. Compile the `Sample4.java` file and run it as you did for the previous example files. See *Running the example*.



Sample4 running

Adding graphic objects

To be able to manipulate graphic objects, you must first import the IBM® ILOG® JViews package that contains the graphic objects:

```
import ilog.views.graphic.*;
```

In this example, you implement the `addObjects` method which adds ten objects of the `IlvIcon` class to the manager:

```
public void addObjects()
{
  manager.setSelectable(0, false);

    for (int i = 0 ; i < 10 ; i++) {
        IlvGraphic obj = new IlvIcon("image.gif", new IlvRect(0,0,37,38));
       manager.addObject(obj, 1, false);
```

```
    }
}
```

The first line in this method calls the `setSelectable` method on the manager with `0` and `false` as its parameters:

```
manager.setSelectable(0, false);
```

The first parameter, `0`, specifies the layer in the manager to which the method applies. The second parameter, `false`, specifies whether objects in the layer passed as the first parameter can be selected (`true`) or not (`false`).

Objects in a manager can be stored in different layers, which are identified by indices. Layers are drawn on top of each other, starting at index 0. In other words, the first layer is assigned the index 0, the second layer, the index 1, and so on, with the objects stored in a higher screen layer being displayed in front of objects in lower layers.

In the `usa.ivl` file loaded in the manager, the objects that make up the map are stored in layer `0`. Calling the `setSelectable` method with `0` and `false` as parameters specifies that the map (layer `0`) cannot be selected, and hence, cannot be modified.

The following `addObject` method adds the `IlvIcon` objects to layer 1 of the manager:

```
manager.addObject(obj, 1, false);
```

Call the `addObjects()` method from the application initiation method. In this case the `Sample4` method.

Note: The `false` parameter of this method specifies that the redraw is not to be triggered. Here no redraw is needed because the application is not visible when this code is executed.

Test the interface of the application by clicking the objects with the mouse. You can see that the new objects are selectable, whereas you can no longer select or modify the map.

Moving graphic objects

`Sample4` has a new button in the `appButtons()` method which is used to move the `IlvIcon` objects in a random way.

```
Button moveButton = new Button("move");
moveButton.setBackground(Color.gray);
moveButton.addActionListener(new ActionListener() {
   public void actionPerformed(ActionEvent evt) {
       moveObjects();
     }
   });
```

```
    buttons.add(moveButton);
  }
```

The movement of the objects is implemented in the `Sample4.moveObjects()` method. This method gets an enumeration of objects contained in layer 1 (the new objects), and, for each of these objects, finds a random object in layer 0 and moves the objects of layer 1 to the center of the objects of layer 0 by calling `IlvManager`.

```
void moveObjects() {
  IlvGraphic state=null, obj=null;
  // get objects in layer 1
  IlvGraphicEnumeration objects, states;
  for (objects = manager.getObjects(1); objects.hasMoreElements();) {
    obj = objects.nextElement();
    // get an random object in layer 0
    states = manager.getObjects(0);
    int index = (int)((double)manager.getCardinal(0)*Math.random());
    state = states.nextElement();
    for (int i = 1 ; i < index; i++)
      state = states.nextElement();
    if (state != null) {
      // move the object.
      IlvRect bbox = state.boundingBox(null);
      manager.moveObject(obj, bbox.x+bbox.width/2,
                              bbox.y+bbox.height/2, true);
    }
  }
}
```

Framework JavaBeans(TM)

Describes the Beans provided as JViews Framework classes, explains how to install them in an IDE, and uses an example to explain how to create an applet with IBM® ILOG® JViews Beans within an IDE. The example shows the main functionality of the Beans.

In this section

Installing IBM® ILOG® JViews Beans in an IDE
Describes the considerations you need to keep in mind when installing IBM® ILOG® JViews Beans in an IDE.

Framework classes available as JavaBeans(TM)
Describes the groups of Beans provided as JViews Framework classes.

Creating a simple applet using IBM® ILOG® JViews Beans
Explains how to create a simple applet using the supplied Beans.

© Copyright IBM Corp. 1987, 2009

Installing IBM® ILOG® JViews Beans in an IDE

The main classes of the JViews Framework fully comply with the JavaBeans™ standard. This allows you to create an IBM® ILOG® JViews application from the visual programming environment of your favorite Integrated Development Environment (IDE).

To be able to use the JViews Framework Beans, you must first install the Beans into your IDE. The Beans are located in *<installdir>* `/jviews-framework86/lib/jviews-framework-all.jar`.

To install JViews Framework JavaBeans refer to your IDE documentation. In most cases, the IDE simply allows you to import a `.jar` file and finds the JavaBeans located in this `.jar` file automatically.

Other Beans are available for the IBM® ILOG® JViews Diagrammer, IBM® ILOG® JViews Maps, IBM® ILOG® JViews Gantt and IBM® ILOG® JViews Charts products. These Beans are not covered in this document. To learn more about them, read the section about JavaBeans in the documentation of your IBM® ILOG® JViews product.

In most IDEs, when you import JAR files, you must make sure that all classes referred to by those beans are also imported. For example, this may mean referring to JViews Framework JARs when you import IBM® ILOG® JViews Maps Beans.

Note: Some IDEs may refuse to import IBM® ILOG® JViews Beans because they are running earlier JDK versions than the one JViews requires. In this case, you may need a newer version of your IDE.

Framework classes available as JavaBeans(TM)

The JViews Framework provides the following groups of Beans:

- *IBM® ILOG® JViews main data structure Beans*
- *IBM® ILOG® JViews main GUI components*
- *Predefined interactors*
- *GUI convenience components*

These Beans are classes of the IBM® ILOG® JViews library. The details of these classes are explained throughout this manual as well as in the IBM® ILOG® JViews Framework Reference Manual. These Beans are listed below along with their icons that are displayed on the toolbar.

Note: Either the small icon or the large icon is displayed depending on the IDE you use.

IBM® ILOG® JViews main data structure Beans

The `IlvManager` Bean, the data structure that stores the graphic objects. In this Bean, you can specify the initial `.ivl` file to be loaded.

The `IlvGrapher` Bean, which organizes the graphic objects into nodes and links of a network.

IBM® ILOG® JViews main GUI components

All the IBM® ILOG® JViews GUI components needed to create an AWT or Swing applet or application are available as JavaBeans™ :

The `IlvManager` Bean, the visual Bean that displays the content of a manager Bean.

The `IlvJScrollManagerView` Bean, a Swing-based Bean that adds the scrolling functionality to `IlvManagerView` objects.

The `IlvScrollManagerView` Bean, the AWT version of the `IlvJScrollManagerView`.

The `IlvManagerViewPanel` Bean, an AWT component designed to contain an `IlvManagerView` Bean and to manage the double-buffering mechanism of the manager view. This Bean is necessary only to create AWT applets or applications using double-buffering.

The `IlvGrid` Bean, the magnetic grid that can be installed on any `IlvManagerView` Bean.

Predefined interactors

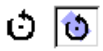
The predefined interactors provided as JavaBeans™ are given below:

The `IlvManagerViewInteractor` Bean, an interactor Bean that has no predefined interaction. You can create your own interaction by binding the different input events (mouse, keyboard) sent by this Bean.

The `IlvZoomViewInteractor` Bean, an interactor that allows the user to select a rectangle of a manager view to be zoomed in.

The `IlvSelectInteractor` Bean, an interactor that allows the user to select and edit the graphic objects of a manager.

The `IlvPanInteractor` Bean, an interactor that allows the user to pan the view of a manager.



The `IlvRotateInteractor` Bean, an interactor that allows the user to rotate objects in a manager.

The `IlvManagerMagViewInteractor` Bean, an interactor that controls the panning and zooming of a target view by manipulating a control rectangle on the view.

The `IlvDragRectangleInteractor` Bean, an interactor that allows the user to drag a rectangle on a view. You can perform any type of action when the rectangle is dragged by binding the `RectangleDragged` event.

The `IlvMakeRectangleInteractor` Bean, an interactor that allows the user to create any type of rectangular object in a manager.

The `IlvMakePolyPointsInteractor` Bean, an interactor that allows the user to create any type of graphic object defined by a set of points, such as a polyline or spline.

The `IlvEditLabelInteractor` Bean, an interactor that allows the user to create and edit a graphic object that contains a label.

The `IlvMakeLinkInteractor` Bean, an interactor that allows the user to create a link in a grapher.

The `IlvMagnifyInteractor` Bean, an interactor that allows the user to move a lens over the view of a manager to magnify the objects under it.

GUI convenience components

The `IlvJManagerViewControlBar` Bean, a Swing toolbar that allows the user to perform selection, zoom, and pan operations on an `IlvManagerView` Bean.

The `IlvManagerViewControlBar` Bean, an AWT version of the `IlvJManagerViewControlBar` Bean.

Creating a simple applet using IBM® ILOG® JViews Beans

To create a simple IBM® ILOG® JViews applet using IBM® ILOG® JViews Framework Beans, no coding is necessary. The applet you create is a simple Swing applet that displays a butterfly with a toolbar allowing you to zoom and pan the content of the view.

For information on the concepts that underlie JavaBeans™ , refer to the Web site: *http://java.sun.com/products/javabeans*. You are assumed to be familiar with the manipulation of JavaBeans inside your IDE.

Note: The Swing Beans that you will use have the letter "J" in the prefix of the Bean name. You could also create the same type of application using only AWT controls. To do so, you would simply use the `IlvScrollManagerView` Bean that is an AWT control instead of the `IlvJScrollManagerView` Bean.

The following example is carried out using a typical IDE procedure. It comprises the following stages:

1. Create the manager view
2. Set the properties of the manager view
3. Create a manager and display its content in a view
4. Load an .ivl file into the manager
5. Add a control toolbar Bean
6. Configure the toolbar
7. Test the result

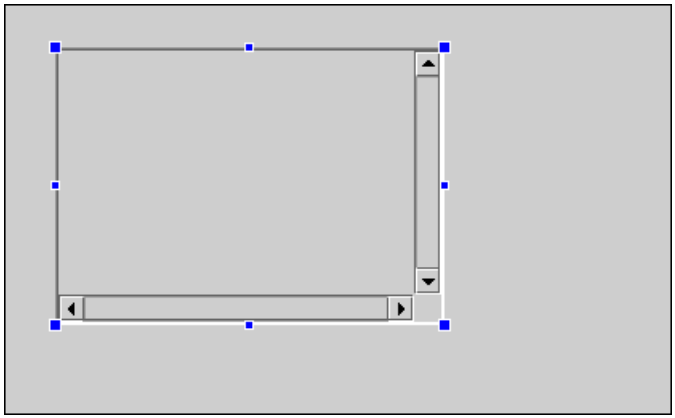
Create the manager view

To create the manager view:

1. Create a new project as a Swing applet or application.
2. Display the IBM® ILOG® JViews Beans on the toolbar by selecting that package.
3. From the toolbar, click the `IlvJScrollManagerView` Bean icon  and drag it inside the form designer of your IDE.

   Warning: There are two of these icons on the toolbar. Make sure you are using `IlvJScrollManagerView` and not `IlvScrollManagerView`.

4. Drag the handles of your `IlvJScrollManagerView` Bean until it appears as in the following figure.

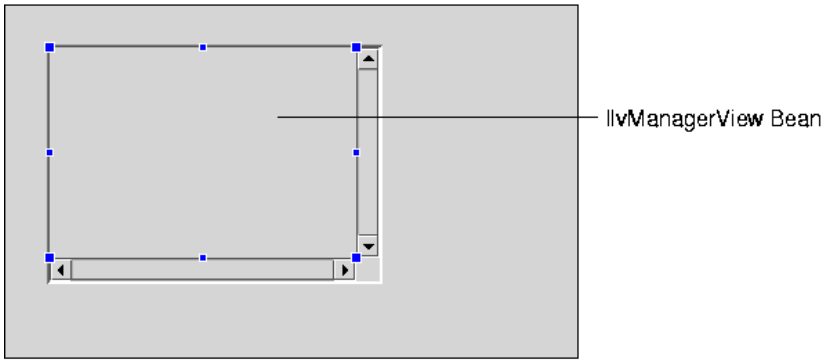


IlvJScrollManagerView Object Selected in the Form Designer

5. Click the IlvManagerView Bean icon on the toolbar and drag it inside the IlvJScrollManagerView Bean.

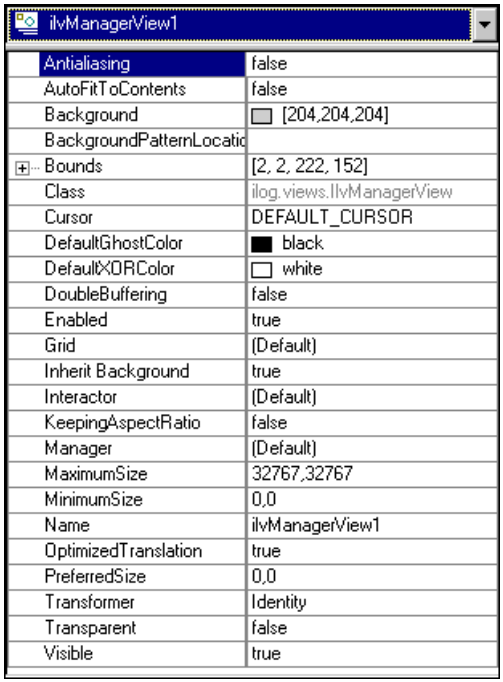
   The result is fairly similar to what you obtained previously, except that you can now select the manager view. See *IlvJScrollManagerView Object with a Selected IlvManagerView Object Inside*.

Note: If you were to compile and run the project at this point, you would see that the IlvJScrollManagerView allows you to scroll through the content of the IlvManagerView Bean.



IlvJScrollManagerView Object with a Selected IlvManagerView Object Inside

The next step is to change a manager view property of the Bean, which is done in the following property sheet. This property sheet is active because the IlvManagerView object is presently selected in the form designer. The property to change is the background property.

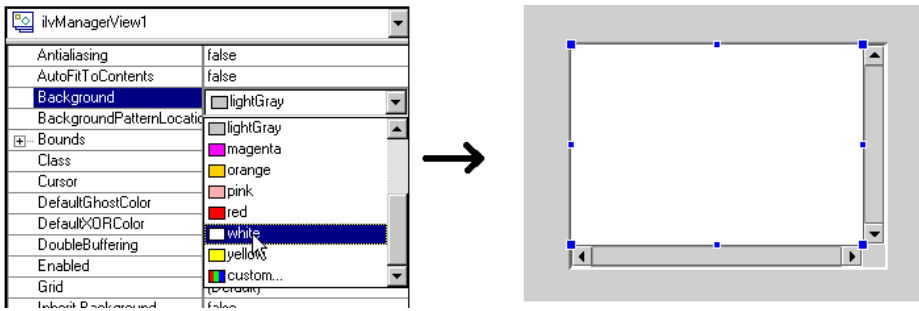


Property Sheet for `IlvManagerView` *Object*

Set the properties of the manager view

To set the properties of the manager view:

1. Click the value field of the `Background` property and change the background of the view to `white`:



Setting the Background property of a View

2. Change the `KeepingAspectRatio` property to `true`.

   This will make sure that the zoom level remains the same along the x and y axis.

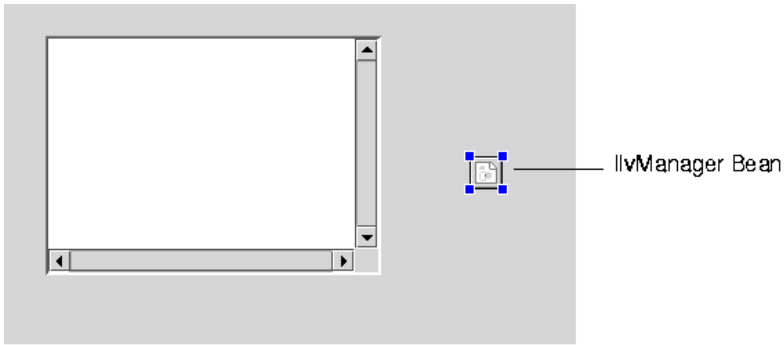
You can now create an `IlvManager` Bean. The `IlvManager` Bean provides the data structure that contains the graphic objects you want to display.

Create a manager and display its content in a view

To create the `IlvManager` Bean and display its content in a view:

1. Click the `IlvManager` Bean icon on the toolbar.
2. Drag it into the form designer.

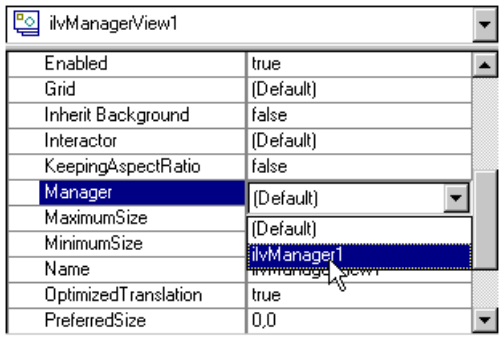
   The class `IlvManager` is not a graphical Bean, so it is not managed the same way by the different IDEs. The image below shows the manager as a small object inside the form designer.



The `IlvManager` Bean in the Form Designer

   You must now associate the view with the manager. This is done by setting the `manager` property of the `IlvManagerView` Bean to the new manager Bean.

3. Select the `IlvManagerView` object so that its property sheet is active.
4. Set the value of its `Manager` property to `ilvManager1` as shown in the following figure.



Setting the Manager property of a View

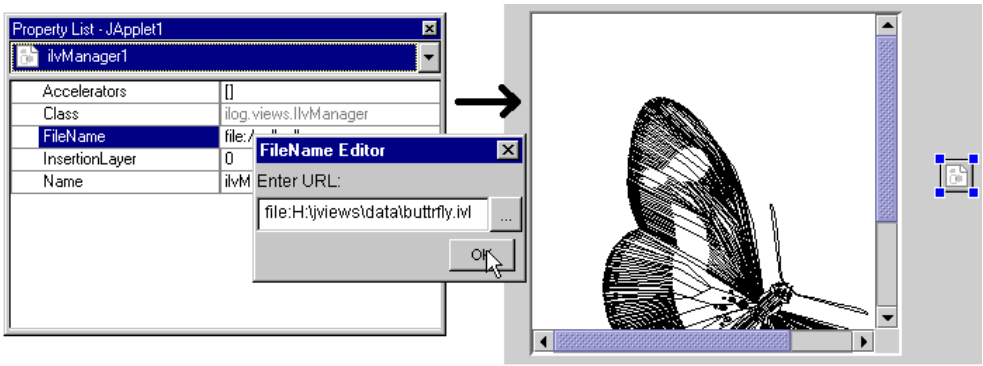
The `IlvManagerView` will now display the content of the `IlvManager` Bean. You can create several `IlvManagerView` objects and associate them with the same `IlvManager` Bean. This allows you to have several views of the same data.

Load an .ivl file into the manager

To load an `.ivl` file into the `IlvManager` Bean:

1. Select `IlvManager1` so that its property sheet is active.
2. Click in the value field of the `FileName` property and then click the ellipsis button that appears.
3. Click the ellipsis button in the FileName Editor window that appears.
4. Browse to the `buttrfly.ivl` file located in the `data` directory of JViews Framework and open it.
5. Click **OK** in the `FileName Editor` dialog box.

The file is automatically displayed in the `IlvManagerView` Bean.



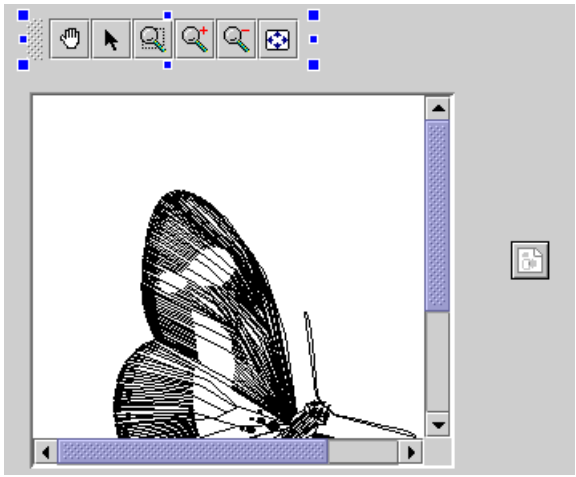
Loading a file into the Manager

The next step is to add a toolbar that allows the user to control the zoom level of the view and to pan the view.

Add a control toolbar Bean

To add a control toolbar Bean:

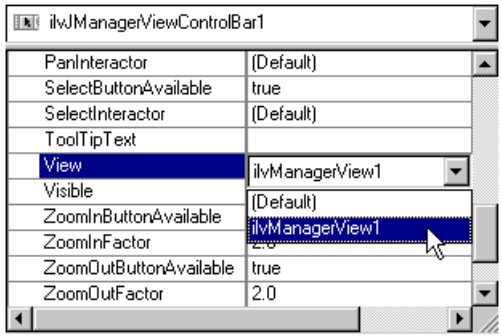
1. Click the `IlvJManagerViewControlBar` icon on the IBM® ILOG® JViews Beans toolbar.
2. Drag it into the form designer.



The Control toolbar in the form designer

You must now associate the toolbar with the view by setting the `View` property of the toolbar.

3. Verify that the `IlvJManagerViewControlBar` object is selected so that its property sheet is active.
4. Select `ilvManagerView1` in the value field of the `View` property as seen in the following figure.



Associating the toolbar with the View

You may configure the toolbar in different ways. You can:

- Hide some of the predefined button icons of the toolbar by setting the corresponding properties: `PanButtonAvailable`, `SelectButtonAvailable`, and so on
- Add your own button icons to the toolbar, as you can with any Swing toolbar
- Modify the default interactors that are used in the toolbar

For example, the toolbar has a `selectInteractor` property that allows you to change the selection interactor used when the user clicks on the Select button icon. You can modify the

properties of the selection interactor Bean to define the type of selection you need. For example, you may want to disable the editing capability.

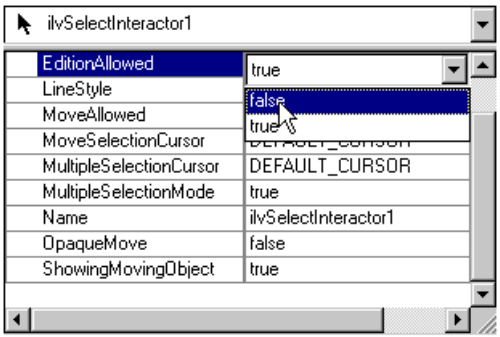
Configure the toolbar

To configure the toolbar:

1. Click an `IlvSelectInteractor` Bean on the toolbar and drag it into the form designer.

Selecting the Selection Interactor

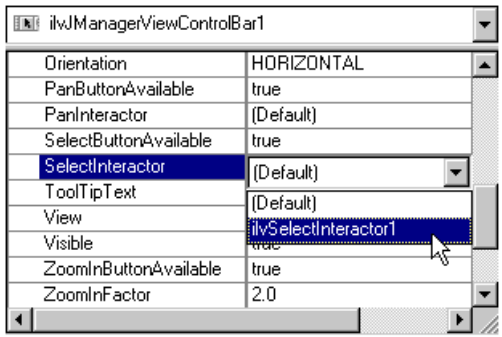
2. Set its `EditionAllowed` property to `false` as seen below.



Customizing the Selection Interactor

You are now going to replace the default selection interactor used in the toolbar by setting the `SelectInteractor` property of `ilvJManagerViewControlBar1`.

3. Select the `ilvJManagerViewControlBar1` object so that its property sheet is active.
4. Change the value of the `SelectInteractor` property to `ilvSelectInteractor1`.



Replacing the default selection interactor

5. Compile the project.

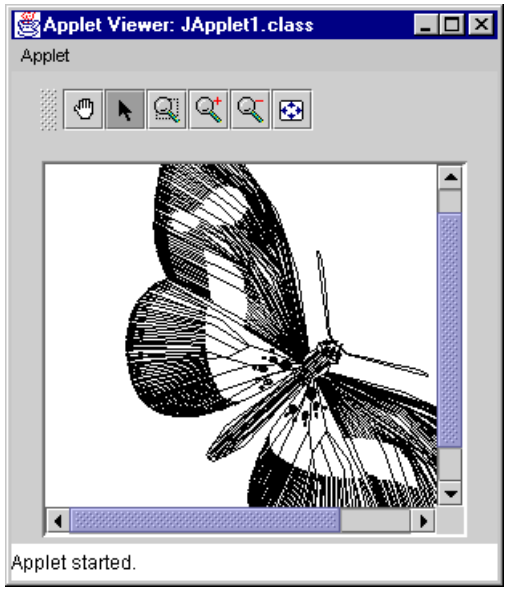
You have created a Java™ application without writing a single line of code.

Note: In this example, you have added interaction to the view by means of the control toolbar. You could also directly set an interactor Bean such as the `IlvSelectInteractor` on the manager view by using the `interactor` property of the `IlvManagerView`.

Test the result

To test the result:

1. Execute the applet. The resulting application should be as follows:



Final application

2. Use the scroll bars and the following toolbar icons to manipulate the image displayed in the manager view:

   - The Pan icon  to pan the content of a view
   - The Select arrow icon  to select objects in the view
   - The Interactive zoom icon  to drag a rectangle over an area that you want to zoom
   - The Zoom-in icon  and the zoom-out icon 
   - The Fit to view icon  to make sure that the content of the manager is fully displayed

This concludes the example. For information on how to save your project and to know what type of files are generated when saving, refer to the documentation of your IDE.

Graphic objects

Describes the Framework hierarchy of classes for creating various high-level graphic objects.

In this section

A graphic object
Describes what a graphic object is.

The class IlvGraphic
Describes the starting point for graphic objects, the class `IlvGraphic`.

Hierarchy of predefined graphic objects
Describes the way predefined graphic objects are organized in a hierarchy.

Geometric properties
Explains how to set the geometric properties that define graphic objects.

User properties of graphic objects
Explains how to set user properties to add application information to graphic objects.

Input/output operations
Describes the classes for saving graphic objects to a stream and reloading them from a stream.

The graphic bag
Describes what a graphic bag is.

Predefined graphic objects
Describes basic classes that provide you with ready-to-use drawing objects.

© Copyright IBM Corp. 1987, 2009

The ShadowEllipse class
Describes the class, ShadowEllipse, which inherits from `IlvGraphic`.. This class is used as an example of creating a new graphic object.

Creating a new graphic object class
Explains how to create a new graphic object class, ShadowEllipse, which inherits from `IlvGraphic`..

Testing for a point inside an object
Describes how to test for a point inside an object.

Saving and loading the object description
Explains the input/output methods for saving and loading an object.

Named properties
Describes the use of user properties called named properties.

A graphic object

A graphic object is an object that users can view on their screen.

When you display a graphic object, you associate its coordinates with the coordinate system of a particular graphic bag.

A graphic bag is an interface that describes the methods to be implemented by a class that contains several graphic objects. An example of a graphic bag is the class `IlvManager`, which can manage a large number of graphic objects. For more information see *Managers*.

Every graphic object has an x value, a y value, and dimensions (that is, width and height). The x and y values define the upper-left corner of the graphic object's bounding box, which is the smallest rectangle containing the entire area of the object. You define the exact shapes of graphic objects in your IBM® ILOG® JViews-based programs and then build them using various drawing methods. Other methods provide you with information about your graphic objects and let you carry out geometric tests concerning the shapes that you are using. For example, you can check whether or not a point with given coordinates lies inside a certain object form.

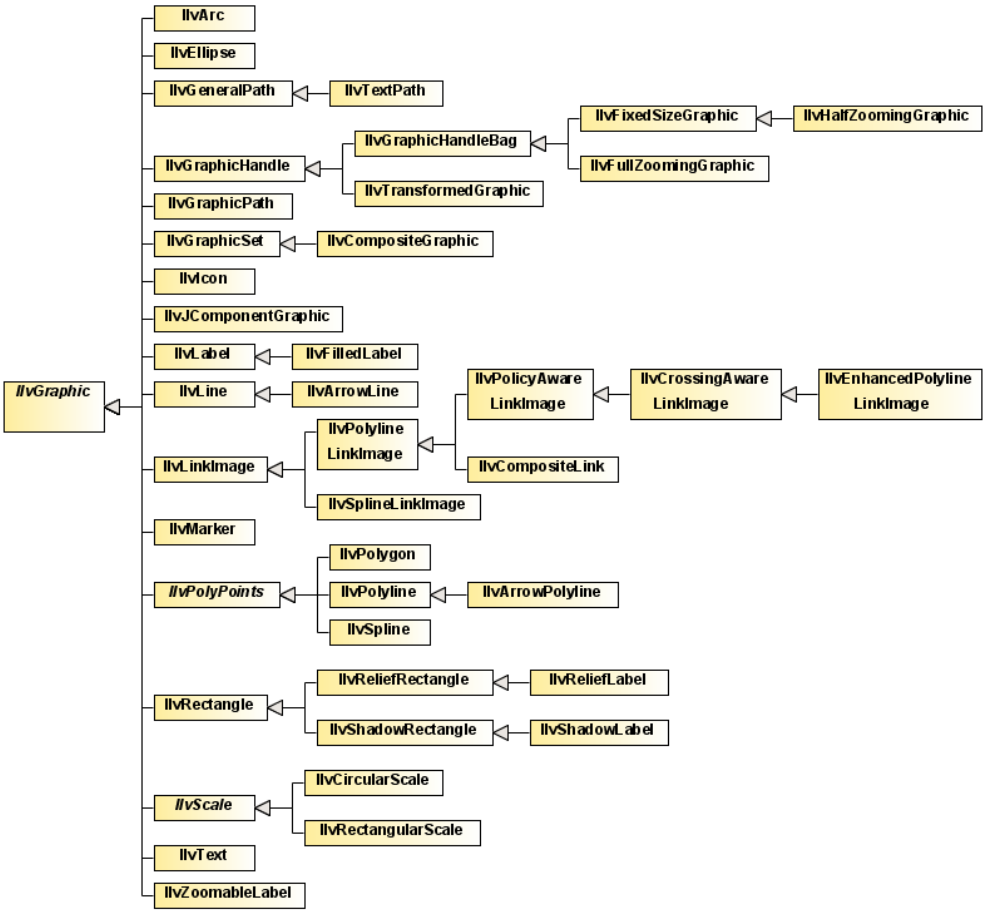
The class IlvGraphic

IBM® ILOG® JViews graphic objects inherit attributes from the `IlvGraphic` abstract base class. This class allows an IBM® ILOG® JViews graphic object to draw itself at a given destination port.If required, the coordinates of the graphic object may also be transformed by an object associated with the `IlvTransformer` class.

The class `IlvGraphic` has methods that allow you to set and change geometric dimensions but does not actually implement these methods. They are declared as nonfinal methods and are defined to perform various operations in the classes that inherit `IlvGraphic` attributes. Although the methods to manipulate geometric shapes and graphic attributes exist, their implementations are empty. Several methods are given to set and get user properties that can be associated with an object for application-specific purposes.

Hierarchy of predefined graphic objects

IBM® ILOG® JViews Framework provides a wide range of predefined graphic objects to create a sophisticated application with minimum coding. These objects/classes are illustrated in the following figure.



Partial class hierarchy of the IBM® ILOG® JViews graphic objects

Geometric properties

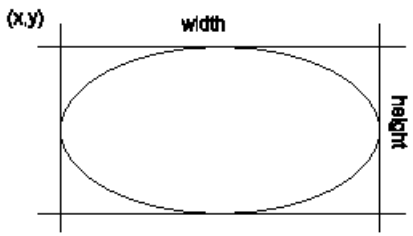
A graphic object is defined by a set of geometric properties, such as its location, size, shape, the way in which it is drawn, and so on. These properties are set by a special group of methods. Of these methods, `draw(java.awt.Graphics, ilog.views.IlvTransformer)IlvGraphic`. and `boundingBox()IlvGraphic`. are fundamental and should be defined jointly.

The boundingBox method

The bounding box defines the smallest rectangle encompassing the graphic object. It is returned by the following method:

```
public IlvRect boundingBox(IlvTransformer t)
```

The `IlvTransformer` parameter is the 2D transformation matrix used to draw the object in a particular drawing port (see transformer). This transformation may correspond to a zoom, a rotation, or a translation of the graphic object in the destination drawing port. The method must then return the rectangle that contains the graphic object when it is drawn using the specified transformation.



The Bounding Box of a Graphic Object

The following example defines the shape of a graphic object with the `drawrect` field. In order to return the bounding box of the object, the `boundingBox` method simply applies the transformer to the rectangle:

```
class MyRectangle extends IlvGraphic
{
   // The geometric rectangle that defines the object.
   final IlvRect drawrect = new IlvRect();

   //constructor
   public MyRectangle(float x, float y, float width, float height)
   {
    drawrect.reshape(x, y, width, height);
   }
    // The bounding box method.
   public IlvRect boundingBox(IlvTransformer t)
```

```
   {
      //Copies the original rectangle to avoid its modification
      IlvRect rect = new IlvRect(drawrect);
      if (t != null)
        t.apply(rect);
      return rect;
   }
}
```

The method `boundingBox` is a very important method. Since it is called very frequently, it must be written in a highly optimized way.

Note: For the MyRectangle class to compile correctly you need to overload the `draw`, `copy` and `applyTransform` methods. For an example of how this is done, see *The ShadowEllipse class*.

The draw method

The `draw` method is used to draw the graphic object. The signature of the method is as follows:

```
public void draw(Graphics dst, IlvTransformer t)
```

The `dst` parameter is the destination `Graphics` where the object is drawn. As in the `boundingBox` method, the `IlvTransformer` parameter is the 2D transformation matrix used to draw the object in the drawing port.

Note: Everything that is drawn with this method must be drawn inside the bounding rectangle of the object (the bounding rectangle of the object being the result of the call to the method `boundingBox` with the same transformation parameter). This is why these two methods should be defined jointly.

In order to draw the object, you will use the drawing methods of the AWT `Graphics` class. If you use Java™ 2 and need to perform Java2D™ drawings, you can cast the `dst` parameter in a `Graphics2D` object and then use the drawing methods of this class.

Zoomable and nonzoomable objects

A graphic object is said to be *zoomable* if its bounding box follows the zoom level. In other words, the result of calling the method `boundingBox` with a transformer is the same as when calling `boundingBox` with a `null` transformer and then applying the transformer to the resulting rectangle. That is:

```
obj.boundingBox(t) = t.apply(obj.boundingBox(null))
```

A zoomable object follows the zoom factor. When a view is magnified by 2, a zoomable object is drawn twice as big. When a view is reduced by 1/2, a zoomable object is drawn half as big. A nonzoomable object does not follow the zoom factor, that is, it may be drawn at its original size in a reduced view.

More precisely, a graphic object is zoomable if and only if for every transformer `t`, the rectangle obtained by calling `obj.boundingBox(t)` is contained in the rectangle obtained by applying the transformer to `obj.boundingBox(null)`. Equality of both rectangles is not necessary.

Important: If you define your own graphic objects, you must define `zoomable()` correctly. If `zoomable()` returns `true`, but the object does not follow the zoom factor, the object may be drawn incorrectly.

Zoomable and nonzoomable objects are managed in very different ways in IBM® ILOG® JViews: zoomable objects are managed in a more optimized way. To know whether an object is zoomable, call the `zoomable` method:

```
public boolean zoomable()
```

The returned value for the class `IlvGraphic` is `true`.

Testing whether a point is part of an object shape

The method `contains` is called by interactors to check whether a point is part of an object shape.

```
public boolean contains(IlvPoint p, IlvPoint tp, IlvTransformer t)
```

The default implementation of this method checks whether the specified point lies inside the bounding rectangle of the object. You may override this method so that it returns `false` for the transparent area of your object.

Moving and resizing a graphic object

The class `IlvGraphic` provides many methods for moving and resizing a graphic object:

- `move(float, float)`

  Moves the upper-left corner of the bounding rectangle of the object to (`x`,`y`).

- `move(ilog.views.IlvPoint)`

  Moves the upper-left corner of the bounding rectangle of the object to the point `p`.

- `moveResize(ilog.views.IlvRect)`

Sets the bounding rectangle of the object to the `IlvRect` parameter.

- `translate(float, float)`

  Translates the bounding rectangle of the object by the vector (`dx`, `dy`).

- `rotate(ilog.views.IlvPoint, double)`

  Rotates the object around the point `center` by an angle of `angle` degrees.

- `scale(double, double)`

  Resizes the bounding rectangle of the object by a factor (`scalex`, `scalex`).

- `resize(float, float)`

  Modifies the bounding rectangle of the object with the new size (`neww`, `newh`).

All of these methods call `applyTransform` to modify the bounding rectangle of the graphic object.

```
public void applyTransform(IlvTransformer t)
```

This is the only method that needs to be overridden in order to handle the transformation of an object correctly. The following code example shows how the `applyTransform` method may be used in the example class, `MyRectangle`:

```
class MyRectangle extends IlvGraphic
{
   // The rectangle that defines the object.
   final IlvRect drawrect = new IlvRect();

   ...

   public void applyTransform(IlvTransformer t)
   {
      t.apply(drawrect);
   }
}
```

The method simply applies the transformation to the rectangle.

Note: Graphic objects stored in a manager (class `IlvManager` and its subclasses) are located in a *quadtree*. This means that you cannot simply call `move` on a graphic object because the quadtree must be notified of the modification of the graphic object. Every method that modifies the bounding rectangle of the object must call `applyToObject (ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang. Object, boolean)`. This method applies a function to an object and notifies the quadtree of the modification to the bounding rectangle. The class `IlvManager` also includes several convenient methods to move and reshape a graphic object managed by this manager. These are as follows:

`moveObject(ilog.views.IlvGraphic, float, float, boolean)` public void moveObject(IlvGraphic, float, float, boolean)

`reshapeObject(ilog.views.IlvGraphic, ilog.views.IlvRect, boolean)` public void reshapeObject(IlvGraphic, IlvRect, boolean)

For more information, see *Modifying geometric properties of objects*.

User properties of graphic objects

A set of user properties can be associated with graphic objects. User properties are a set of key-value pairs, where the key is a `String` object and the value may be any kind of information value. These user property methods of the class `IlvGraphic` let you easily connect information that comes from your application to your graphic objects. You can keep track of the graphic part of your application by storing the references to objects you create and connecting this graphic part to the application by means of user properties, as in the following example:

```
Integer index = new Integer(10);
String key = "objectIndex";
myobject.setProperty(key, index);
```

The following `IlvGraphic` methods help you manage the properties of an object:

```
public boolean hasProperty(String key, Object value)
```

```
public boolean removeProperty(String key)
```

```
public Object getProperty(String key)
```

```
public boolean replaceProperty(String key, Object value)
```

```
public void setProperty(String key, Object value)
```

Input/output operations

The IBM® ILOG® JViews library provides the following two classes for saving graphic objects to, and loading graphic objects from, a stream:

- `IlvOutputStream`, used by the class `IlvManager` to save all the graphic objects that it contains
- `IlvInputStream`, allowing a file generated by `IlvOutputStream` to be read into an `IlvManager`

Graphic objects can always be written to an `IlvOutputStream` because they inherit the `write` method of the `IlvGraphic` class:

```
public void write(IlvOutputStream stream) throws IOException
```

To save the information contained in your class, you can override this method and use the methods of the class `IlvOutputStream`. When overriding this method, you must not forget to call the `write` method of the superclass to save the information related to the superclass. You will obtain something that resembles the following example.

```
public void write(IlvOutputStream stream) throws IOException
{
  // write fields of super class
  super.write(stream);
  // write fields of my class
  stream.write("color", getColor());
  stream.write("thickness", getThickness());
  ....
}
```

To read your graphic object from an `IlvInputStream`, you must create a constructor with an `IlvInputStream`. This constructor is mandatory even if you have not overridden the `write` method. The corresponding constructor in the class `IlvGraphic` is:

```
public IlvGraphic(IlvInputStream stream) throws IlvReadFileException
```

Assuming that `MyClass` is the name of your class, your new constructor will look like this:

```
public MyClass(IlvInputStream stream) throws IlvReadFileException
```

In the body of this constructor, you first call the corresponding constructor in the superclass, then you read the information you have saved in the `write` method. In the above example, the corresponding constructor is:

```
public MyClass(IlvInputStream stream) throws IlvReadFileException
{
  super(stream);
  setColor(stream.readColor("color"));
```

```
  setThickness(stream.readInt("thickness"));
  ...
}
```

Important: The recommended way to serialize any `IlvManager` object is through IVL serialization and not Java™ serialization. Serialization cannot work for managers that contain graphic objects such as `IlvIcon` or some other classes, since these classes internally manage Java SE objects that are not serializable.

The graphic bag

Graphic objects are placed in a graphic bag.

A graphic bag (interface `IlvGraphicBag`) is an object that contains several graphic objects, which can be added or removed. The interface `IlvGraphicBag` is implemented by the class `IlvManager`. This class allows you to manage a large set of graphic objects. The following method returns the graphic bag, if there is one, where the object is located:

```
public IlvGraphicBag getGraphicBag()
```

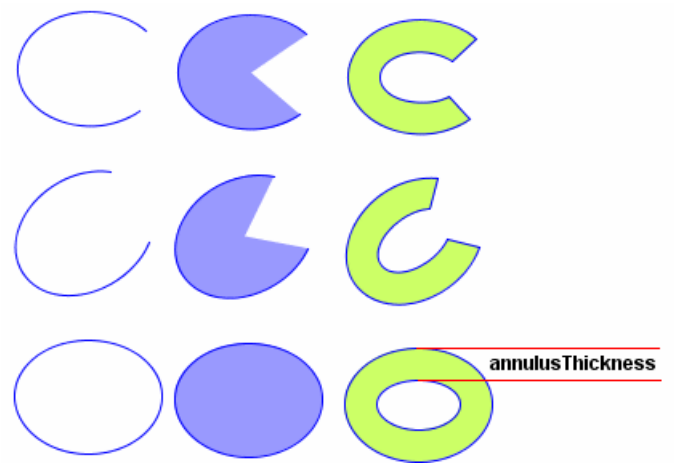
Read *Managers* for more information on this topic.

Predefined graphic objects

There are various predefined graphic objects.

Arcs

There is one arc object, IlvArc.



An `IlvArc` object appears as an outlined, a filled, or a filled and outlined arc of an ellipse. Since JViews 8.0, the `IlvArc` object has an annulus thickness. When the annulus thickness is 0.0, the arc has the same behavior as before. When the annulus thickness is grater than 0.0, the arc object becomes an annulus. The arc object has also an `IlvTransformer` that allows you to apply transformations to the arc object when its `transformerMode` is set to `true`.

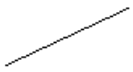
Ellipses

There is one ellipse object, IlvEllipse.

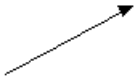
An `IlvEllipse` object appears as an outlined, a filled, or filled and outlined ellipse.

Lines

The line objects are IlvLine and IlvArrowLine.



An `IlvLine` object appears as a straight line between two given points.



An `IlvArrowLine` object appears as a straight line between two given points, with a small arrowhead drawn at the end of the trajectory.

Rectangles

The rectangle objects are IlvRectangle, IlvReliefRectangle, and IlvShadowRectangle.

An `IlvRectangle` object appears as a closed rectangle. It can be outlined, filled, or filled and outlined. You can also set rounded corners on the `IlvRectangle` object.

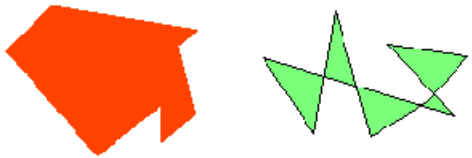
IlvReliefRectangle

An `IlvReliefRectangle` object appears as a filled rectangle in relief.

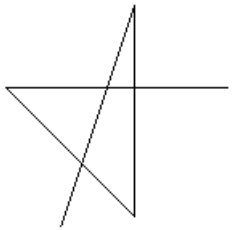
An `IlvShadowRectangle` object appears as a rectangle with a shadow underneath.

Polygons and polylines

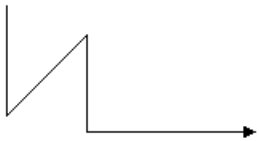
The class `IlvPolyPoints` is an abstract class from which every class having shapes made up of several point coordinates is derived.



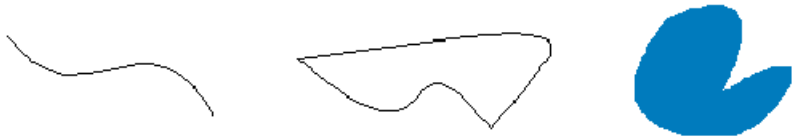
An `IlvPolygon` object appears as a filled, outlined, or filled and outlined polygon.



An `IlvPolyline` object appears as connected segments.



An `IlvArrowPolyline` object appears as a polyline and adds one or more arrows to the various lines.



An `IlvSpline` object appears as a Bézier spline. It can be either opened or closed, and may also be filled.

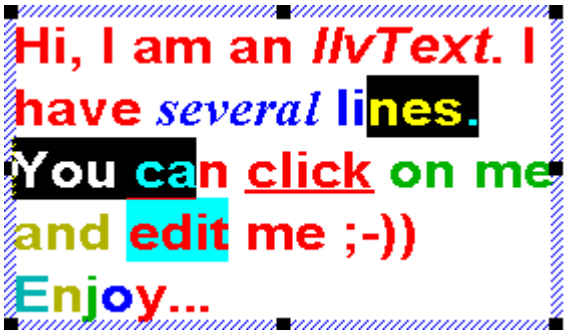
Labels and text

The label objects are IlvLabel, IlvReliefLabel, and IlvShadowLabel. The text object is IlvText.

An `IlvLabel` object appears as a single line of text. It cannot be zoomed in or reshaped. `IlvLabel` supports WYSIWYG text editing.

An `IlvReliefLabel` object appears as a relief rectangle holding a single line of text.

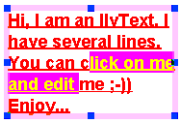
An `IlvShadowLabel` object appears as an `IlvShadowRectangle` with a label.



An `IlvText` object appears as a single line of text or several lines of text that can be zoomed and rotated. In multiline mode, the text can be either wrapped or truncated and can be aligned on the leading, center, or trailing position. `IlvText` supports WYSIWYG text editing.

In-place text editing

IBM® ILOG® JViews supports WYSIWYG editing for `IlvLabel` and `IlvText` objects.



A WYSIWYG editable object

This editing behavior is implemented by the `IlvTextSelection` and `IlvTextEditor` classes. While editing selected text, the user can perform the following actions with the mouse and keyboard:

- Click in the text to indicate an insertion point.
- Select a zone of text by dragging the mouse over it.
- Use the arrow keys to navigate in the selected text.
- Combine the Shift and arrow keys to extend the selection zone.
- Copy and paste using the Ctrl+C and Ctrl+V keys.

During an edit session you can perform the following move and reshape actions.

- Drag the handles to resize a selected `IlvText`object.
- Drag the borders to move a selected `IlvText` object.

Note: You cannot do the following:

- Edit an `IlvText` object in `WRAP_TRUNCATE` mode. This is because you cannot see all the text on the screen. Change the object to `WRAP_WORD` mode before editing.
- Edit an `IlvLabel` or an `IlvText`object that is not editable. Change the attributes of one object to editable using `IlvManager.setEditable(...)` before editing.
- Resize an `IlvLabel` object. `IlvLabel`objects are never resizable.

For more information, see ***<installdir>*** **`/jviews-framework86/codefragments/interactors/texteditor/index.html`**.

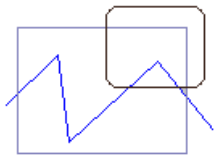
Markers

The marker object is IlvMarker.

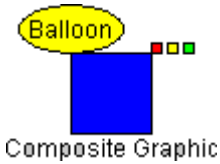
An `IlvMarker` object is a nonzoomable object that displays a graphic symbol.

Groups

The group objects are IlvGraphicSet and IlvCompositeGraphic.

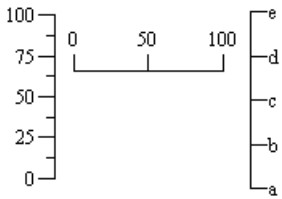


An `IlvGraphicSet` object is an object that groups graphic objects together.

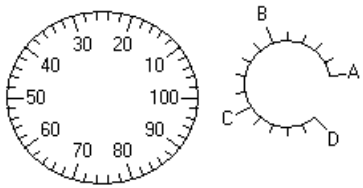


The class `IlvCompositeGraphic`.enables you to associate graphic objects in a single object that features both dynamic and layout functionality. See *Composite Graphics* for more information.

Scales



An `IlvRectangularScale` object displays a vertical or horizontal scale.



An `IlvCircularScale` object displays a circular scale defined by a portion of an ellipse, a starting angle, and an angle range.

Icons

An `IlvIcon` object appears as an image.

Paths

IlvGeneralPath objects can display any Java 2D™ Shape objects. This means that they can represent curves, rectangles, ellipses, general paths, and so on., and any combination of them. You can define Java 2D properties for these objects, such as Paint or Stroke. The last two objects with the fade-out effect use “gradient paint” objects.

IlvGraphicPath

An IlvGraphicPath object is a set of polypoints that can be drawn as polylines or as polygons. Depending on the position of its points, a polypoint may either appear as an ordinary polygon or as a hole in another polygon.

Component graphics

An IlvComponentGraphic object is a wrapper class that lets you embed a Swing JComponent in a manager.

The ShadowEllipse class

The ShadowEllipse object is an ellipse object with a drop shadow, as seen below:

A ShadowEllipse

You can design such an object from scratch by implementing a subclass of the `IlvGraphic` class. `IlvGraphic` is an abstract class. Therefore, some of its methods must be redefined in derived classes. This is the case for the following:

```
public abstract void draw(Graphics dst, IlvTransformer t)
```

```
public abstract IlvRect boundingBox(IlvTransformer t)
```

```
public abstract void applyTransform(IlvTransformer t)
```

```
public abstract IlvGraphic copy()
```

Other methods, such as `move`, `resize`, `rotate`, and `contains`, have a default implementation in the `IlvGraphic` implementation. These methods, as well as any other method that modifies the bounding box, are implemented by means of a call to the `applyTransform` function. If the new class has a parent that defines some of these methods, you simply inherit the functions from this parent class.

The `ShadowEllipse` class defines the `draw`, `contains`, and `boundingBox` methods. In addition, it defines a `write` method that is necessary to write the object to a stream and a constructor that takes an `IlvInputStream` as a parameter to read the object from a stream. For details, see *The write method* and *The read constructor*. These methods have no default implementation. You must provide a version of them for each subclass of the `IlvGraphic` class. If you do not intend to write additional information to the stream, you do not need to implement the `write` method, but you always need to define a constructor with an `IlvInputStream` parameter. Otherwise, you will not be able to read the object from a stream.

Creating a new graphic object class

The procedure for creating a new graphic object class comprises stages for defining methods to deal with geometric properties and drawing and stages for making the object persistent.

The stages are as follows:

1. *Stage 1 - Creating the class*
2. *Stage 2 - Defining the Constructors*
3. *Stage 3 - Overriding the draw Method*
4. *Stage 4 - Overriding the boundingBox method*
5. *Stage 5 - Overriding the applyTransform method*
6. *Stage 6 - Overriding the copy method*
7. *Stage 7 - Defining accessors*

This example creates the class `ShadowEllipse`. The complete source code of the `ShadowEllipse` example is available at ***<installdir>*** **`/jviews-framework86/codefragments/shadow-ellipse/src/ShadowEllipse.java`**.

Stage 1 - Creating the class

To create the class:

1. Create a file named `ShadowEllipse.java` that defines the new class and the necessary overloaded methods. Not every method needs to be overloaded.
2. Add the following statements at the beginning of your file:

   ```
   import ilog.views.*;
   import ilog.views.io.*;
   import ilog.views.graphic.*;
   import java.awt.*;
   import java.io.*;
   ```

   These statements allow you to use the basic, the input/output, and the graphic packages of the IBM® ILOG® JViews library.
3. Define a class that inherits from the `IlvGraphic` class.

   This class has two colors: one for the ellipse and one for the shadow. It also defines the thickness of the shadow.
4. Define the bounding rectangle of the object.

   For this, you add a member variable named `drawrect` of type `IlvRect`.

   ```
   import ilog.views.*;
   import ilog.views.io.*;
   import ilog.views.graphic.*;
   import java.awt.*;
   ```

```
import java.io.*;

/**
 * A shadow ellipse object. A graphic object defined by two
 * ellipses: The main ellipse and a second ellipse of the same
 * size underneath the first one that represents a shadow.
 */
public class ShadowEllipse
extends IlvGraphic
{
  /**
   * The definition rectangle of the ellipse.
   * This rectangle is the bounding rectangle of the
   * graphic object.
   */
  protected final IlvRect drawrect = new IlvRect();
  /**
   * The color of the ellipse.
   */
  private Color  color = Color.blue;
  /**
   * The color of the shadow.
   */
  private Color  shadowColor = Color.black;
  /**
   * The thickness of the shadow.
   */
  private int thickness = 5;
```

Stage 2 - Defining the Constructors

- Define a constructor to create a new shadow ellipse. These constructors simply set the value of the definition rectangle or create a new `ShadowEllipse` from an existing `ShadowEllipse` instance:

```
  /**
   * Creates a new shadow ellipse.
   * @param rect the bounding rectangle of the shadow ellipse.
   */
  public ShadowEllipse(IlvRect rect)
  {
    super();
    // Stores the bounding rectangle of the object.
    drawrect.reshape(rect.x, rect.y, rect.width, rect.height);
  }

/**
   * Creates an ellipse by copying another one.
   * @param source the object to copy.
   */
  public ShadowEllipse(ShadowEllipse source)
  {
    // First call the superclass constructor
```

```
    // that will copy the information of the superclass.
    super(source);

    // Copies the bounding rectangle.
    drawrect.reshape(source.drawrect.x, source.drawrect.y,
       source.drawrect.width, source.drawrect.height);
    // Copies the color and the color of the shadow.
    setColor(source.getColor());
    setShadowColor(source.getShadowColor());
    // Copies the thickness
    setThickness(source.getThickness());
  }
```

Stage 3 - Overriding the draw Method

- Draw the object by calling some of the primitive methods contained in the AWT `Graphics` class.

```
/**
   * Draws the object.
   * Override the draw method to define the way the object will appear.
   * @param dst The AWT object that will perform the
   * drawing operations.
   * @param t This parameter is the transformer used to draw the object.
   * This parameter may be a translation, a zoom or a rotation.
   * When the graphic object is drawn in a view (IlvManagerView),
   * this transformer is the transformer of the view.
   */
  public void draw(Graphics dst, IlvTransformer t)
  {
    // First copy the rectangle that defines the bounding
    // rectangle of the object so that it is not modified.

    IlvRect r = new IlvRect(drawrect);

    // To compute the bounding rectangle of the object in
    // the view coordinate system, apply the transformer 't'
    // to the definition rectangle.
    // The transformer may define a zoom, a translation or a rotation.

    // applyFlooris used so the resulting rectangle
    // is correctly projected for drawing in the view.
    // The object's coordinate system is defined by 'float' values
    // Need 'int' values to be able to draw. applyFloor will
    // apply the transformation to 'r' and then call Math.floor to
    // translate 'float' values to 'int' values.
    if (t != null)
      t.applyFloor(r);
    else
      r.floor();

    // The variable 'r' now contains the bounding rectangle
```

```
    // of the object in the view's coordinate system ready to
    // draw in this rectangle. In this rectangle, two ellipses
    // are drawn: first the shadow ellipse on the bottom
    // right corner of the definition rectangle, then the main
    // ellipse on the top-left corner. Each ellipse will be of size
    // (r.width-thickness, r.height-thickness).

    int thick = thickness;

    // Computes a correct value for thickness.
    // Since the size of the ellipses should
    // be (r.width-thickness, r.height-thickness), need to
    // check that the thickness is not too big.

    if ((r.width <= thick) || (r.height <= thick))
      thick = (int)Math.min(r.width, r.height);

    // Sets the size of the ellipses.
    r.width  -= thick;
    r.height -= thick;

    // 'r' now contains the bounding area of the main ellipse.

    // Computes a rectangle to draw the shadow.
    // Copy the variable 'r', needed for the
    // second ellipse.
    IlvRect shadowRect = new IlvRect(r);
    shadowRect.translate(thick, thick);

    // Draws the shadow ellipse
    dst.setColor(getShadowColor());
    dst.fillArc((int)shadowRect.x,
                (int)shadowRect.y,
                (int)shadowRect.width,
                (int)shadowRect.height,
                0, 360);

    // Draws the main ellipse.
    dst.setColor(getColor());
    dst.fillArc((int)r.x,
                (int)r.y,
                (int)r.width,
                (int)r.height,
                0, 360);
  }
```

The method `draw` fills the two ellipses. The bounding rectangle, `drawrect`, actually covers both ellipses.

Note: The AWT methods, such as `fillArc`, require all coordinates to be integers. In IBM® ILOG® JViews, however, the bounding box of a graphic object is defined by `float`

values. To convert coordinates from `float` to `int`, use the `applyFloor` and `floor` methods of the `IlvTransformer` class. You must use the same technique to ensure that the other objects comply with the library.

Stage 4 - Overriding the boundingBox method

- Define the method `boundingBox` to transform the bounding box. It creates a copy of the rectangle `drawrect` even if the transformer is `null`. This is so the returned rectangle can be modified by IBM® ILOG® JViews.

```
/**
  * Computes the bounding rectangle of the graphic
  * object when drawn with the specified transformer.
  */
 public IlvRect boundingBox(IlvTransformer t)
 {
   // First copy the definition rectangle
   // so that it is not modified.
   IlvRect rect = new IlvRect(drawrect);
   // Apply the transformer on the rectangle to
   // translate to the correct coordinate system.
   if (t != null) t.apply(rect);
   return rect;
 }
```

Stage 5 - Overriding the applyTransform method

- Override the `applyTransform` method to apply a transformation to the shape of the ShadowEllipse rectangle.

```
public void applyTransform(IlvTransformer t)
 {
   // This method is called by method such as IlvGraphic.move
   // IlvGraphic.rotate or IlvGraphic.scale to modify the
   // shape of the object. For example, when this method
   // is called from IlvGraphic.move, the parameter 't' is the
   // corresponding translation.
   // Simply need to apply the transformer to
   // the definition rectangle of the object.
   t.apply(drawrect);
 }
```

Stage 6 - Overriding the copy method

- Override the copy method to call the `ShadowEllipse` copy constructor to make a new instance.

```
/**
  * Copies the object.
  */
 public IlvGraphic copy()
 {
   // Simply call the copy constructor that is defined above.
   return new ShadowEllipse(this);
 }
```

Stage 7 - Defining accessors

- Add public accessors to the graphic object. These accessors deal with thickness and color. They appear in **bold** type in the following code example.

```
/**
 * Changes the thickness of the shadow ellipse
 * @param thickness the new thickness
 */
public void setThickness(int thickness)
{
  this.thickness = thickness;
}

/**
 * Returns the thickness of the shadow ellipse
 * @return the thickness of the object.
 */
public int getThickness()
{
  return thickness;
}

/**
 * Changes the color of the ellipse.
 * @param color the new color.
 */
public void setColor(Color color)
{
  this.color = color;
}

/**
 * Returns the color of the shadow ellipse
 * @return the color of the object.
 */
public Color getColor()
{
  return color;
}

/**
 * Changes the color of the shadow
```

```
 * @param color the new color
 */
public void setShadowColor(Color color)
{
  this.shadowColor = color;
}

/**
 * Returns the color of the shadow
 * @return the color of the shadow
 */
public Color getShadowColor()
{
  return shadowColor;
}
```

Testing for a point inside an object

When drawing IBM® ILOG® JViews graphic objects you often need to validate the presence of a point inside the object. The `ShadowEllipse` example implements the `contains` method. It returns `true` if the specified point is located within the main ellipse. All the coordinates are specified relative to the coordinate system of the view.

```
/**
  * Tests whether a point lies within the shape of the object.
  * This method will be called when you click on the object.
  * @param p The point where user clicks in the object's coordinate system.
  * @param tp Same point as 'p' but transformed by transformer 't'
  * @param t The transformer used to draw the object.
  */
 public boolean contains(IlvPoint p, IlvPoint tp,
                         IlvTransformer t)
 {
   // Allow the user to click on the main ellipse
   // but not on the shadow ellipse.
   // This method will return true when the clicked point is
   // on the main ellipse.
   // First compute the bounding rectangle of the main
   // ellipse in the view coordinate system, just like in the
   // method draw.
   IlvRect r = new IlvRect(drawrect);

   if (t != null)
     t.apply(r);

   int thick = thickness;

   if ((r.width <= thick) || (r.height <= thick))
     thick = (int)Math.min(r.width, r.height);

   r.width  -= thick;
   r.height -= thick;

   // Then call PointInFilledArc that will return true
   // if the point is in the ellipse. 'r' and 'tp' are both
   // in the view coordinate system.
   return IlvArcUtil.PointInFilledArc(tp, r, (float)0, (float)360);
 }
```

Saving and loading the object description

To save and read an object in an IBM® ILOG® JViews formatted file, you need to implement the `write` method and a constructor that takes an `IlvInputStream` parameter.

Important: The recommended way to serialize any `IlvManager` object is through IVL serialization and not Java™ serialization. Serialization cannot work for managers that contain graphic objects such as `IlvIcon` or some other classes, since these classes manage internally Java SE objects that are not serializable.

The write method

The method `write` writes the colors of the object, the dimensions of the rectangle, and the thickness of the shadow to the provided output stream:

```
/**
  * Writes the object to an output stream.
  */
 public  void write(IlvOutputStream stream)
   throws IOException
 {
   // Calls the super class method that will write
   // the fields specific to the super class.
   super.write(stream);
   // Writes the colors.
   stream.write("color", getColor());
   stream.write("shadowColor", getShadowColor());
   // Writes the thickness.
   stream.write("thickness", getThickness());
   // Writes the definition rectangle.
   stream.write("rectangle", drawrect);
 }
```

In the `write` method, the `write` method of the superclass is called to save the information specific to the superclass. Then the `write` methods of the class `IlvOutputStream` are used to save the information specific to the class.

The read constructor

To read a graphic object from a file, you must provide a specific constructor with an `IlvInputStream` parameter. This constructor must be public to allow the file reader to call it. Also, the constructor must read the same information, and in the same order, as that written by the `write` method.

```
/**
  * Reads the object from an IlvInputStream
```

```
  * @param stream the input stream.
  * @exception IlvReadFileException an error occurs when reading.
  */
public ShadowEllipse(IlvInputStream stream) throws
                                              IlvReadFileException
{
  // Calls the super class constructor that reads
  // the information for the super class in the file.
  super(stream);
  // Reads the color.
  setColor(stream.readColor("color"));
  // Reads the shadow color.
  setShadowColor(stream.readColor("shadowColor"));
  // Reads the thickness
  setThickness(stream.readInt("thickness"));
  // reads the definition rectangle.
  IlvRect rect = stream.readRect("rectangle");
  drawrect.reshape(rect.x, rect.y, rect.width, rect.height);
}
```

The above constructor calls the read constructor of the superclass which reads the information specific to the superclass from the stream object. The subclass can then read its own information. The constructor uses the `read` methods defined in the class `IlvInputStream`

Named properties

Another kind of user property, called named property, can also be set on a graphic object. A named property is an instance of the class `IlvNamedProperty`. This class is an abstract class; it must be subclassed for your own needs. The difference between a named property and a user property is mainly that this type of property is named and can be saved with the graphic object in an `.ivl` file when the graphic object is saved. Note that a named property can also be stored in the manager or in a *layer* object, which is described in *Layers*.

To store a named property in a graphic object, use:

```
void setNamedProperty(IlvNamedProperty)
```

To get a named property, use:

```
void getNamedProperty(String name)
```

The following example shows a named property.

```
import ilog.views.*;
import ilog.views.io.*;

public class MyNamedProperty extends IlvNamedProperty
{
  String value;

  public MyNamedProperty(IlvInputStream stream) throws IlvReadFileException
  {
    super(stream);
    this.value = stream.readString("value");
  }

  public MyNamedProperty(String name, String value)
  {
    super(name);
    this.value = value;
  }

  public MyNamedProperty(MyNamedProperty source)
  {
    super(source);
    this.value = source.value;
  }

  public IlvNamedProperty copy()
  {
    return new MyNamedProperty(this);
  }

  public boolean isPersistent()
```

```
  {
    return true;
  }

  public void write(IlvOutputStream stream) throws IOException
  {
    super.write(stream);
    stream.write("value", value);
  }

}
```

This named property defines a member variable value of type `String` to store the value of the property.

Several methods have been created to allow the storage of the property in an `.ivl` file:

- The method `isPersistent` of the named property returns `true`.
- The method `write` is used to store the object in an `.ivl` file.

  This method is overridden to store the string in the member variable `value`.Note that the `super.write` call is mandatory for a correct storage of the property.
- The class defines a public constructor with an `IlvInputStream` parameter.

  This constructor is used to reload the property from the `.ivl` file.

> **Note**: The complete name of the class (including the name of the package) will be stored in the `.ivl` file. Therefore, if you change the name of the class, the property can no longer be loaded. Also, the class must be a public class to be saved in an `.ivl` file. Otherwise, it is impossible for the `.ivl` file reader to instantiate the class.

Managers

Describes how to coordinate a large quantity of graphic objects through the use of a manager, that is, through the `IlvManager` class and its associated classes.

In this section

A manager
Describes what a manager is with a diagram.

A manager view
Explains the purpose of a manager view and how it operates.

Layers
Explains the purpose of layers and how they operate.

Handling input events: interactors and accelerators
Describes how all input events are handled by means of an interactor or an accelerator.

Input/output
Describes the set of methods for writing graphic object descriptions to a file and reading them to a file.

Class diagram for IlvManager
Describes the relationships between the main manager classes with a class diagram.

Multiple manager views
Explains the purpose of multiple manager views and how to create them.

© Copyright IBM Corp. 1987, 2009

Binding views to a manager
Describes how to use manager views.

Managing layers
Explains how to use layers.

Managing graphic objects
Describes how to assign graphic objects to a manager.

Selection in a manager
Describes how to select objects through the manager and display them as selected.

Hover highlighting in a manager
Describes how to use hover highlighting in a top-level manager.

Blinking of graphic objects
Describes the three types of blinking mode supported by IBM® ILOG® JViews.

Managing input events
Describes how to handle input events using an interactor on the view or on a graphic object.

Saving and reading
Describes the facilities for saving and loading the contents of a manager.

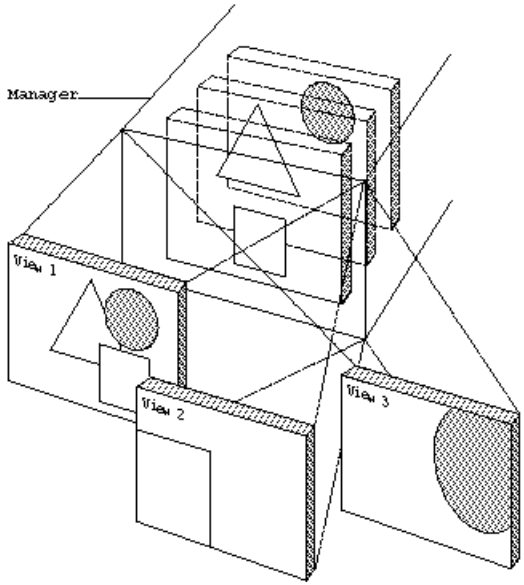
File formats
Describes the file formats supported by JViews Framework.

Drawing Exchange Format (DXF)
Describes how to use and customize DXF.

A manager

A manager is the data structure that contains the graphic objects.

A manager organizes graphic objects in multiple storage places and coordinates the interactions between the display of graphic objects in multiple views, as illustrated in the following figure.



Manager concept

A manager view

A manager view is the AWT component where the graphic objects of a manager are displayed.

To display graphic objects contained in a manager, you create at least one view, and often multiple views. The manager lets you connect as many views as you require to display graphic objects. The creation of a view is shown in *Creating a manager and a view*.

A geometric transformation can be associated with each view so that you can display any portion of the global space where your graphic objects are located with appropriate scales (zooming) and rotations for each view. See *View transformation*.

Layers

To organize graphic objects in a manager you place them in multiple storage areas called layers.

Each graphic object stored in a layer is unique to that layer and can be stored only in that layer.

Instances of the `IlvManager` class handle a set of graphic objects derived from the IBM® ILOG® JViews class called `IlvGraphic`.. The different graphic objects stored throughout the manager all share the same coordinate system. For this reason, a manager is a tool designed to handle objects placed on different priority levels. “Priority level” here means that objects stored in a higher screen layer are displayed in front of objects in lower layers.

Handling input events: interactors and accelerators

Interactors

An `IlvManager` instance responds to user actions according to the state of the manager when a certain input event occurs, and also according to the position and shape of the object that receives the event.

`IlvManager` actions can be either global (that is, applied to a whole view through instances of classes derived from `IlvManagerViewInteractor`) or local (applied to an object or a set of objects in a view through instances of classes derived from `IlvObjectInteractor`).

The manager associates an interactor object, that is, an instance of the `IlvManagerViewInteractor` class, with each view. This interactor object processes events that are intended for that particular view. If the manager has not associated an interactor object with a view, then each event is handled by the interactor object associated with the graphic object that received the event. In this case, the interactor object belongs to the `IlvObjectInteractor` class, which manages the events for a particular object.

Accelerators

If no object is indicated when the event is received, or, if it has no associated `IlvObjectInteractor`, the manager tries to apply an accelerator, which is a direct call of a user-defined action, such as the pressing of a certain key sequence. In fact, in certain situations, the best solution is to establish a generic action for all the objects associated with a single event sequence so that, for example, pressing Ctrl+Z causes the view to zoom. To do this, IBM® ILOG® JViews allows you to associate direct actions with events. These actions, which are bound neither to the view nor to the object that was clicked, are called accelerators.

Input/output

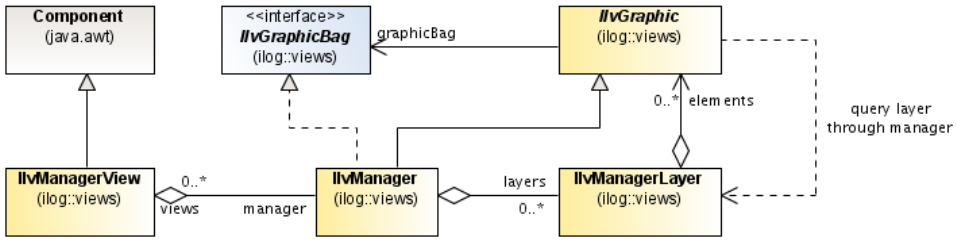
The `IlvManager` class has a set of methods to read and write graphic object descriptions to a file. Manager properties, such as the layer or name of an object, can also be read and written.

Important: The recommended way to serialize any `IlvManager` object is through IVL serialization and not Java™ serialization. Serialization cannot work for managers that contain graphic objects such as `IlvIcon` or some other classes, since these classes manage internally Java SE objects that are not serializable.

Class diagram for IlvManager

The following UML class diagram summarizes the relationships between `IlvManager`, `IlvGraphic`, `IlvManagerLayer`, and `IlvManagerView`.

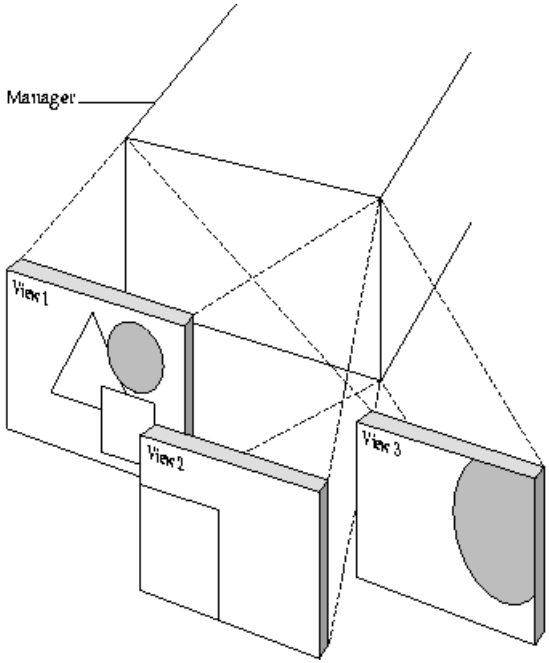
The manager contains a number of manager views that in turn contain graphic objects as elements. The layer of a graphic object can be queried and manipulated by the API on the `IlvManager`. The contents of the manager are displayed by a manager view. Multiple manager views can be attached to the same manager.



The classes related to IlvManager

Multiple manager views

Attaching multiple views to a manager allows your program to display graphic objects simultaneously in various configurations.



Multiple views bound to a manager

To bind a view to a manager, you simply need to create a manager view, an instance of the class `IlvManagerView`. The constructors of `IlvManagerView` take an `IlvManager` parameter.

Binding views to a manager

Describes how to use manager views.

In this section

Creating a manager and a view
Describes how to create a manager and one view.

Listener for the views of a manager
Presents the listeners to events constituting changes to a manager view.

View transformation
Describes the role of the transformer in displaying an area of the manager in a view.

Scrolled manager view
Describes the way to have a manager view with scroll bars.

Managing double buffering
Describes how to implement double buffering.

The manager view grid
Describes how to implement a grid with snap-to functionality.

Class diagram for IlvManagerView
Describes the relationships between the main manager view classes with a class diagram.

Manager view repaint skipper
Describes how to skip some repaint requests for performance reasons.

Creating a manager and a view

The following code creates a manager and a view:

```
Frame frame= new Frame("JViews");
IlvManager mgr = new IlvManager();
IlvManagerView view = new IlvManagerView(mgr);
frame.add("Center", view);
frame.setSize(200,200);
frame.setVisible(true);
```

The class `IlvManagerView`. is a subclass of the AWT class `java.awt.Container`. A manager view is visible when added to a parent container, that is an AWT Frame or a Swing JFrame. It becomes invisible when removed from a visible parent container.

To obtain a list of all the views attached to a manager, use the following `IlvManager` method:

```
Enumeration getViews()
```

You may also retrieve and change the manager displayed by a particular view using the following methods of the class `IlvManagerView`:

```
IlvManager getManager()
```

```
void setManager(IlvManager manager)
```

Listener for the views of a manager

ManagerViewsChangedEvent

When a view is attached or detached from a manager, a `ManagerViewsChangedEvent` event is fired by the manager. A class must implement the `ManagerViewsChangedListener` interface to be notified that an `IlvManagerView` has been attached or detached from the manager. This interface contains only the `viewChanged` method, which is called for each modification:

```
void viewChanged(ManagerViewsChangedEvent event)
```

To be notified, a class implementing this interface must register itself using the following method of the class `IlvManager`:

```
void addManagerViewsListener(ManagerViewsChangedListener l)
```

ManagerChangedEvent

When the manager displayed by a view changes, as a result to a call to `setManager` on the view, the view fires a `ManagerChangedEvent`. A class must implement the `ManagerChangedListener` interface in order to be notified that the manager of the view has changed and must register itself on the view using the `addManagerChangedListener(ilog.views.event.ManagerChangedListener)` method of the `IlvManagerView`. You can also specify that the listener no longer be notified of such events using the `removeManagerChangedListener(ilog.views.event.ManagerChangedListener)` method.

When the manager of a view changes, the view calls the `managerChanged` method of the listeners.

```
void managerChanged(ManagerChangedEvent event)
```

This method is called with an instance of the class `ManagerChangedEvent` as a parameter containing information on the new and the old manager.

View transformation

Each manager view (class `IlvManagerView`) has its own transformer to define the area of the manager that the view is displaying and also to define the zoom level and rotation applied to objects.

You may retrieve the current transformer of a view using the following method:

```
IlvTransformer getTransformer()
```

To modify the transformer associated with a view, use the following methods:

```
void setTransformer(IlvTransformer t)
```

```
void addTransformer(IlvTransformer t)
```

```
void translate(float deltax, float delaty, boolean redraw)
```

```
void zoom(IlvPoint, double, double, boolean)
```

```
void fitTransformerToContent()
```

```
void ensureVisible(IlvPoint p)
```

```
void ensureVisible(IlvRect rect)
```

To avoid distorting the image when it is zoomed in or out, you can specify that the vertical and horizontal aspect ratio remain the same by using the following methods:

```
boolean isKeepingAspectRatio()
```

```
void setKeepingAspectRatio(boolean set)
```

When the `KeepingAspectRatio` property is on, the view ensures that the horizontal and vertical scaling are always the same, whatever transformer you set in the view.

Example: Zooming a view

The following code zooms a view in by a scale factor of 2:

```
managerView.zoom(point, 2.0, 2.0, true);
```

The `point` given as an argument keeps its position after the zoom. The last parameter forces the redrawing of the view.

Transformer listeners

When the transformer of a view changes, the view fires a `TransformerChangedEvent` event. A class must implement the `TransformerListener` interface to be notified that the transformer of the view has changed, and must register itself using the `addTransformerListener(ilog.views.event.TransformerListener)` method of `IlvManagerView`. You can also specify that the listener no longer be notified of such events using the `removeTransformerListener(ilog.views.event.TransformerListener)` method.

When the transformer of a view changes, the view calls the `transformerChanged` method of all listeners.

```
void transformerChanged(TransformerChangedEvent event)
```

This method is called with an instance of the class `TransformerChangedEvent` as a parameter. The `event` parameter can be used to retrieve the old and the new value of the transformer.

Scrolled manager view

The library provides a convenience class that handles a manager view with two scroll bars in an AWT application: `IlvScrollManagerView`. This class automatically adjusts the scroll bars according to the area defined by the graphic objects contained in the manager. An equivalent object exists to be integrated into a Swing application: `IlvJScrollManagerView`

Managing double buffering

Double buffering is a technique that is used to prevent the screen from flickering in an unpleasant manner when many objects are being manipulated. Since the manager view is implemented as a lightweight component, that is, as a direct subclass of `java.awt.Container`, it cannot handle double buffering by itself. To use double buffering in an AWT environment, the manager view must be the child of a heavyweight component, specially designed to handle double-buffering for instances of `IlvManagerView`. These components can be of the `IlvManagerViewPanel` or of the `IlvScrollManagerView` class.

The methods of the `IlvManagerViewPanel` and the `IlvScrollManagerView` class that handle double-buffering are:

```
boolean isDoubleBuffering()
```

```
void setDoubleBuffering(boolean set)
```

In a Swing application, the manager view is embedded in a `JComponent`. `JComponent` objects have their own double-buffering mechanism:

```
   jcomponent.setDoubleBuffered(true);
```

When you add an `IlvManagerView` into an `IlvJManagerViewPanel` or an `IlvJScrollManagerView`, local double buffering in the `IlvManagerView` instance is disabled and Swing double buffering is used instead. In specific situations, when Swing double buffering is disabled, enable `IlvManagerView` local double buffering by calling `setDoubleBuffering(boolean)` after the view has been added to the Swing component.

Example: Using double buffering

This example creates a standard `IlvManagerView`, associates it with an `IlvManagerViewPanel`, and sets the double-buffering mode:

```
IlvManager mgr = new IlvManager();
IlvManagerView v = new IlvManagerView(mgr);
IlvManagerViewPanel panel = new IlvManagerViewPanel(v);
panel.setDoubleBuffering(true);
```

The manager view grid

Most editors provide a snapping grid that forces the objects to be located at specified locations. The coordinates where the end user can move the objects are called *grid points*. The class `IlvGrid` provides this functionality.

An instance of the class `IlvGrid` can be installed on each manager view. The view provides methods to set or retrieve the grid:

```
public void setGrid(IlvGrid grid)
```

```
public IlvGrid getGrid()
```

The following code installs a grid on a view with a vertical and horizontal grid point spacing of 10. The last two parameters are set to `true` to specify that the grid is visible and active:

```
mgrview.setGrid(new IlvGrid(Color.black, new IlvPoint(), 10f, 10f, true, true)
);
```

When a grid is installed on a view, the standard IBM® ILOG® JViews editing interactors, such as those for creating, moving, or editing an object, snap objects to the grid automatically.

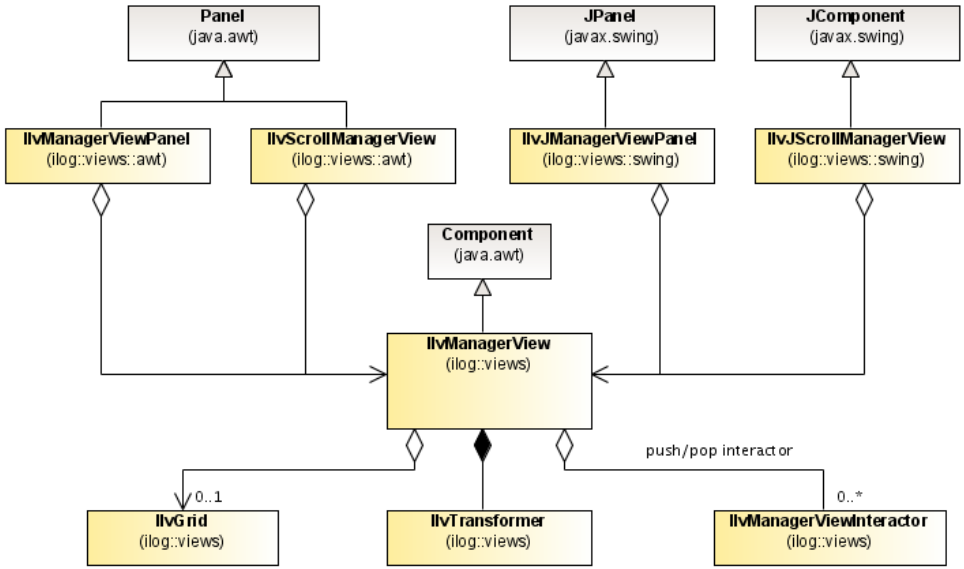
These operations are not performed by the manager, but by the interactor itself. If you want to implement this mechanism in a new interactor you create, use the following method of the `IlvManagerView` class in the code of your new interactor:

```
public final void snapToGrid(IlvPoint point)
```

This method moves the `IlvPoint` argument to the closest point on the grid if a grid is installed and active. Otherwise, it does nothing.

Class diagram for IlvManagerView

The following UML class diagram summarizes the relationships between `IlvManagerView`, `IlvTransformer`, `IlvGrid`, and `IlvManagerViewInteractor`. The manager view can be contained in an `IlvManagerViewPanel` or in an `IlvScrollManagerView` object; for Swing applications use `IlvJManagerViewPanel` and `IlvJScrollManagerView`. The manager view contains a local `IlvTransformer` that allows the user to zoom into the view. Optionally, the manager view can contain an `IlvGrid`. The `IlvManagerViewInteractor` class handles all interactions that are specific to the view.



The Classes Related to IlvManagerView

Manager view repaint skipper

To improve repaint performance, the manager view allows you to skip some of the repaint requests.

When you manipulate a graphic object in a manager view, the repaint requests are sent to all views that are attached to the same manager. The view in which you are manipulating the graphic object should be refreshed as often as possible, so that you receive feedback on your manipulation in real time. This view is called the main view.

During the manipulation, it is unlikely that you would pay attention to the other views attached to the same manager. Therefore, it is not essential to keep refreshing these auxiliary view as often as the main view. It is the main view that holds the focus of the manipulation.

In a typical configuration you have the main view showing a region of a large map and an auxiliary overview showing where the region occurs in the map. When you zoom or manipulate objects in the main view, you do not need the overview to be refreshed as often as the main view.

To save CPU processing time and to gain fluidity in the main view, the refresh rate of the auxiliary views can be reduced.

When the refresh delay is set, the manager view will skip some repeated repaint requests if the delay has not elapsed since the last time the view was refreshed. You can adjust the setting of this parameter between 300ms and 800ms according to your needs. For example, to set the delay to 300ms, use the manager view method:

`setRepaintSkipThreshold(long)`(300);

The default value is 0, which disables this feature.

Important: You should use this feature with caution. In certain circumstances it might skip some desired repaint requests and pollute the view until the delay elapses.

To avoid skipping desired repaints, you can temporarily turn the view repaint mode to `DIRECT_REDRAW`. Repaints are skipped only under `THREADED_REDRAW` mode.

Managing layers

Explains how to use layers.

In this section

Layers in a manager
Explains how to organize objects into various layers in a manager.

Setting up layers
Explains how to create, access, and modify layers.

Layers and their graphic objects
Explains how to place objects in specific layers and make them visible and selectable.

Listener for layer changes in a manager
Describes how to implement a listener for a layer.

Triple buffering layers
Describes how to implement triple buffering for applications with a static background.

Caching layers
Describes how to cache any layers without the constraints of triple buffering and how to combine this feature with triple buffering.

Manipulating the drawing order
Describes how to change the drawing order to get certain objects in front of others.

Layers in a manager

Layers are storage places for graphic objects in a manager. Each layer, with its graphic objects, is unique to a single manager and can only be controlled by this manager.

When you store graphic objects in layers, you indicate their placement throughout multiple layers. When you display graphic objects stored in multiple layers, you present layer contents in a series of one or several views, with each view controlled by and specific to the same manager.

Various methods let you manipulate layers or the objects that they own. When redrawing takes place, a layer with the number N is placed in front of layers with numbers from N-1 to zero.

Inherent to the notion of layers is the concept of visual hierarchy among graphic objects stored in layers and displayed in views. In general, graphic objects of a more static nature, such as objects that might serve as shading or background for your IBM® ILOG® JViews programs, should be put in a lower layer of the manager. Those graphic objects of a dynamic nature, such as objects with which users interact, should typically be put in a higher layer so that they are not hidden.

Setting up layers

Layers are handled internally by the class `IlvManagerLayer`. Layers can be accessed by their index or by their instance (a pointer to an `IlvManagerLayer`). By default, a manager is created with one layer. However, you can specify how many layers you want to create for a manager in the second parameter of the `IlvManager` constructor.

Once the manager has been created, you can modify the number of layers using the following methods:

```
void addLayer(int index)
```

```
void removeLayer(int index, boolean redraw)
```

and retrieve the number of layers with:

```
int getLayersCount()
```

Layers and their graphic objects

When an object is added to a manager, you can specify the index of the layer where it should be inserted.

The following method adds the specified graphic object to the specified layer:

```
void addObject(IlvGraphic obj, int layer, boolean redraw)
```

To retrieve the index of the layer that contains a certain graphic object, use the following method:

```
int getLayer(IlvGraphic obj)
```

To change the layer, use the following method:

```
void setLayer(IlvGraphic obj, int newLayer, boolean redraw)
```

There are two essential properties that you can specify for the objects within a layer: visibility and scalability.

Visibility

With the following methods, you can indicate whether the objects within a certain layer should be visible to the user:

```
void setVisible(int layer, boolean value, boolean redraw)
```

```
boolean isVisible(int layer)
```

You can also decide whether a layer is visible or not within a particular view. Refer to the following methods:

```
void setVisible(IlvManagerView view, int layer, boolean set, boolean redraw)
```

```
boolean isVisible(IlvManagerView view, int layer)
```

Finally, you can have a visible layer in a view temporarily hide itself depending on certain conditions, generally depending on the zoom factor. This can be achieved through an `IlvLayerVisibilityFilter` that is called each time the `IlvManager` needs to redraw a layer. You should implement this interface and return whether or not the layer is visible with the `isVisible` method. To be active, this filter must be registered on the corresponding `IlvManagerLayer` using the `addVisibilityFilter(ilog.views.IlvLayerVisibilityFilter)` method.

Selectability

You can specify whether objects within a layer can be selected or not using the following methods. Objects that cannot be selected cannot be modified:

```
boolean isSelectable(int layer)
```

```
void setSelectable(int layer, boolean v)
```

For more methods dealing with layers, see the class `IlvManager` in the reference documentation.

Listener for layer changes in a manager

A class must implement the `ManagerLayerListener` interface to be notified that layers have been inserted, removed, or moved in a manager. This interface contains four methods:

- `layerInserted(ilog.views.event.ManagerLayerInsertedEvent)`

  which is called when a layer is added to a manager.

- `layerMoved(ilog.views.event.ManagerLayerMovedEvent)`

  which is called when a layer is moved in a manager.

- `layerRemoved(ilog.views.event.ManagerLayerRemovedEvent)`

  which is called when a layer is removed from a manager.

- `layerChanged(ilog.views.event.ManagerLayerEvent)`

  which is called for other changes in a layer.

To be notified of layer modifications, a class implementing this interface must register itself using the following method of the class `IlvManager`:

```
void addManagerLayerListener(ManagerLayerListener l)
```

Convenience class for listener

The class `IlvManagerLayerAdapter` is a convenience class. It implements the `ManagerLayerListener` interface with empty methods. This is useful if you want to implement a manager layer listener that needs only to listen to some, but not all, of the events.

Triple buffering layers

Certain applications can use layers to display a static background on top of which “live” graphic objects will be drawn and manipulated by the user.

In this type of application, the graphic objects taking part in the background are static and are not modified by the user or the application. Thus, it is possible to draw just once for all of the layers constituting the graphic background. This increases the drawing speed of the application.

The process is called triple buffering. This term is used because the layers will be drawn in an additional off screen image. Thereafter, when the view needs to be redrawn, this image is used instead of redrawing the graphic objects.

Note: Unlike double buffering, triple buffering is not engaged to remove flickering but to increase the drawing speed. Double and triple buffering can be used together.

For an instance of `IlvManagerView`, it is possible to indicate that a certain number of layers will be part of the triple buffering.

This is done using the following method of `IlvManagerView`:

```
void setTripleBufferedLayerCount(int n)
```

When this method is called, layers with indices between 0 and n-1 (the nth background layers) will be triple buffered.

Once the method is called and the view has been painted once, further modifications to graphic objects will not be rendered on the screen, since only the triple buffer image will be displayed.

Note that the triple buffer will be updated only when:

- The transformer of the view changes (you are zooming or panning the view).
- You add or remove layers.
- You change the number of triple-buffered layers.
- The triple-buffered layer visibility changes.
- You add graphic objects to or remove them from the triple-buffered layers.
- You call `applyToObject(ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)` to make changes to the triple-buffered graphic objects on the triple-buffered layers.

Note: Some interactors, such as the reshape interactor, call this function. Therefore, when you reshape a graphic object on a triple-buffered layer with the mouse, the triple buffer is invalidated.

- You change the visibility of the graphic objects on the triple-buffered layers.
- You change the triple-buffered layer on which the graphic objects are positioned.

If for any reason you need to update the triple buffer, you can use these methods:

```
void invalidateTripleBuffer(boolean repaint)
void invalidateTripleBuffer(IlvRect rect, boolean repaint)
```

To summarize, an application will use triple buffering when the contents of background layers are static, and when the application does not require the user to zoom and pan frequently.

Caching layers

Triple buffering is used to cache a set of layers for one view. The constraint is that these layers must be contiguous from layer 0 to layer n. If you want cache layers that are not contiguous or a layer whose index is not 0, since JViews 8.1, `IlvManagerView` allows you to cache any layer for the view concerned.

To enable or disable a layer cache for the view, you can call the following method:

```
void setLayerCached(int layer, boolean enabled)
```

To know if a given layer is cached or not, call the following method:

```
boolean isLayerCached(int layer)
```

When a layer for a view is cached, it will first draw into a buffered image of the same size as the view. When the view needs to be repainted, the buffered image is displayed on the screen. The buffered image must be transparent so that layers behind it are not hidden.

You can enable the cache on any layer. Usually, caches are enabled on layers having many static graphic objects, that is, objects whose drawing does not change frequently, such as layers containing map (cartographic) information. However, this does not mean that the content of the cached layers cannot be changed. It just means that the speed benefit of the cache is higher when the content of the layer changes rarely. When you make a change to a graphic object such as inserting, removing, or applying an operation (see `applyToObject (ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)`), the cache is automatically invalidated.

If you hesitate between enabling the layer cache and using triple buffering, the following facts can help you make the choice:

- If the layers you want to cache or to buffer are contiguous and their indexes are from 0 to n, you should use triple buffering. In this case, triple buffering gives better performance than layer caches because the latter have to handle transparency.
- If the layers you want to cache or to buffer are not contiguous, you have to use layer caches.

You can also use both features for the same view. You can triple buffer contiguous layers from 0 to n, and in addition you can cache any layer above layer n. In this way, you will get the best performance.

Note: With some very rare configurations (Java ™ SE and OS), the transparent buffered image might not give good performance. In this case, you can perform a test to see if layer caches can improve performance.

Manipulating the drawing order

When several objects overlap partially, some objects appear in front of other objects. This effect is called *drawing order* or *Z-order*. Objects of layer N are placed in front of objects of layers N-1 to zero. Moving objects from one layer to another is one way of influencing the drawing order.

If there are several objects within the same layer, then these objects again have a drawing order. Each layer has a spatial data structure called quadtree which allows you to determine very quickly which objects are at which position. By default, the quadtree is enabled and determines the drawing order automatically in order to achieve optimal performance. In this case, the drawing order cannot be influenced.

If you want to specify the drawing order of objects within the same layer, you must first enable the Z-ordering option of the layer, by using the following method:

```
setZOrdering(boolean enable)
```

When Z-ordering is enabled for the layer, you can specify the drawing order of the objects within the layer:

```
setIndex(IlvGraphic object, int index)
```

Note that the index is always a continuous range from 0 to N. This means that if you set the index of an object, the index of other objects will shift by +1 or -1 to adjust the index range. The current index can be retrieved by

```
getIndex(IlvGraphic object)
```

When Z-ordering is enabled, objects with a higher index appear in front of objects of the same layer with a lower index.

Note: The drawing order between different layers takes precedence over the drawing order within each layer: An object that is in a higher-numbered layer is drawn in front of another object in a lower-numbered layer even if the Z-order index of the first object is smaller than Z-order index of the second object. Therefore, the Z-order index determines only the drawing order of objects within the same layer.

Scenarios for experts

There are basically three scenarios:

- The quadtree is enabled and Z-ordering is disabled. This results in the highest performance. In particular the hit-test (determining which objects are at a given position) is optimally fast. However, it is not possible to influence the drawing order within each layer.

- The quadtree is enabled and Z-ordering is enabled. This is slightly slower, depending on how many objects overlap in average. The hit-test uses the quadtree and is still fast. It is possible to specify the drawing order completely.
- The quadtree is disabled. No matter whether Z-ordering is enabled or disabled, it results in the same speed, which is a large magnitude slower than when the quadtree is enabled. When the quadtree is disabled, it is also possible to specify the drawing order.

A test with 10000 objects showed that enabling Z-ordering slows down the hit-test in average by a factor of 1.2-5, but disabling the quadtree slows down the hit-test by a factor of 20-70. These factors depend on the number of objects, and the factors are negligible if you have only a very few objects. Furthermore, if Z-ordering is enabled, the slowdown is mainly influenced by the overlapping depth (the number of objects that are overlapping cover exactly the same location): the higher the overlapping depth, the larger the slowdown of enabled Z-ordering.

Managing graphic objects

Describes how to assign graphic objects to a manager.

In this section

Adding objects to a manager and removing them
Describes how to add a graphic object to a manager and remove it, and how to find out how many objects are managed by the manager.

Modifying geometric properties of objects
Describes how to modify the geometric properties of objects using a manager method.

Applying functions
Describes how to apply a user-defined function to objects.

Editing and selecting properties
Describes how to specify the editing properties of an object.

Optimizing drawing tasks
Describes how drawing tasks can be minimized.

Listener for the content of the manager
Describes how to listen for changes to the content of a manager.

Adding objects to a manager and removing them

The purpose of the manager is to manage a large set of graphic objects. Each graphic object can be managed by only one manager at a time, which means that you cannot add the same graphic object to two different managers.

The following methods allow you to add a graphic object to a manager.

```
void addObject(IlvGraphic obj, int layer, boolean redraw)
```

```
void addObject(IlvGraphic obj, boolean redraw)
```

The following is an example that creates a rectangle object and adds it to a manager:

```
IlvManager mgr = new IlvManager();
IlvGraphic obj = new IlvRectangle(new IlvRect(10,10,100,100));
mgr.addObject(obj, false);
```

The second `addObject` method does not specify the layer where the object must be inserted. The reason for this is that there is a default insertion layer which allows you to add objects without specifying the layer at every call. The initial value for the default insertion layer is 0 but it can be modified using the following methods:

```
int getInsertionLayer()
```

```
void setInsertionLayer(int layer)
```

Once an object has been added to a manager, you can remove it with:

```
void removeObject(IlvGraphic obj, boolean redraw)
```

You can also remove the objects from the manager or from a specific layer using one of the following methods:

```
void deleteAll(boolean redraw)
```

```
void deleteAll(int layer, boolean redraw)
```

The following method can be used to know whether a graphic object is managed by the current manager:

```
boolean isManaged(IlvGraphic obj)
```

You can access all the objects of the manager or of a specified layer using one of the following methods:

```
IlvGraphicEnumeration getObjects()
```

```
IlvGraphicEnumeration getObjects(int layer)
```

These methods return an `IlvGraphicEnumeration` object facilitating the enumeration of the contents of the manager (or layer). You may use it in the following manner:

```
IlvGraphicEnumeration objects = manager.getObjects();
IlvGraphic obj;
while(objects.hasMoreElements()) {
    obj = objects.nextElement();
    //perform some action
}
```

Note: When stepping through the contents of the manager (or layer) by means of an enumeration, you must not modify the contents of the manager by adding or removing objects or layers. Doing so may lead to unpredictable results.

Other useful methods will give you the number of objects in the manager or in a layer:

```
int getCardinal()
```

```
int getCardinal(int layer)
```

Modifying geometric properties of objects

For every operation that leads to a modification of the bounding box of a graphic object, you must use the `applyToObject(ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)` method of the `IlvManager` class. As this method notifies the manager of the modification of the bounding box, you never directly call the `moveObject` and `reshapeObject` methods of `IlvGraphic`:

```
void moveObject(IlvGraphic obj, float x, float y, boolean redraw)
```

```
void reshapeObject(IlvGraphic obj, IlvRect newrect, boolean redraw)
```

For these basic operations, the manager has methods that call `applyToObject` for you:

Example: Moving an object

The following code gets a reference to an object named `test` from the manager. If the object exists, it is moved to the point (10, 20) and redrawn (fourth parameter set to `true`).

```
IlvGraphic object = manager.getObject("test");
if (object != null)
  manager.moveObject(object, 10, 20, true);
```

The `moveObject` method is equivalent to the following code:

```
manager.applyToObject(object,
  new IlvApplyObject()
  {
   public void apply(IlvGraphic obj, Object arg){
     IlvPoint p = (IlvPoint) arg;
     obj.move(p.x, p.y);
   }
  },
  new IlvPoint(10,20), true);
```

This code calls the `applyToObject` method with `object` as a parameter and an anonymous class that implements the `IlvApplyObject` interface. The `arg` parameter is an `IlvPoint` object that gives the new location of the object.

The method `applyToObject` is defined in the `IlvGraphicBag` interface, so you may call `applyToObject` directly from a graphic object using:

```
obj.getGraphicBag().applyToObject(obj, ...);
```

Modifying multiple graphic objects

To apply an operation to many graphic objects repeatedly, call:

```
void applyToObjects(IlvGraphicVector vector,
                    IlvApplyObject f,
                    Object arg,
                    boolean redraw)
```

This applies the operation specified by the `IlvApplyObject f` to each graphic object contained in the input vector.

To apply complex operations that affect the bounding box of many graphic objects to many objects once only, call:

```
void applyToObjects(IlvGraphicVector vector,
                    IlvApplyObjects f,
                    Object arg,
                    boolean redraw)
```

This applies the operation specified by the `IlvApplyObjects f` once only. This is useful for complex operations that affect the bounding box of many objects.

Note: The `applyToObjects` method is overloaded. In the first example it takes an `IlvApplyObject` object as a parameter. In the second example it takes an `IlvApplyObjects` (plural) object as a parameter.

Applying functions

To apply a user-defined function to objects that are located either partly or wholly within a specific region, use the following `IlvManager` methods:

- `mapInside(ilog.views.IlvApplyObject, java.lang.Object, ilog.views.IlvRect, ilog.views.IlvTransformer)`

  to apply a function to all graphic objects inside a specified rectangle.

- `mapIntersects(ilog.views.IlvApplyObject, java.lang.Object, ilog.views.IlvRect, ilog.views.IlvTransformer)`

  to apply a function to all graphic objects that intersect a specified rectangle.

Editing and selecting properties

The `IlvManager` class contains the following methods, which allow you to control the editing and selecting properties of an object added to a manager:

- To specify whether an object can be moved:

  ```
  setMovable(ilog.views.IlvGraphic, boolean)
  ```

  ```
  isMovable(ilog.views.IlvGraphic)
  ```

- To specify whether an object can be edited:

  ```
  setEditable(ilog.views.IlvGraphic, boolean)
  ```

  ```
  isEditable(ilog.views.IlvGraphic)
  ```

- To specify whether an object can be selected:

  ```
  setSelectable(ilog.views.IlvGraphic, boolean)
  ```

  ```
  boolean isSelectable(IlvGraphic obj)
  ```

These properties can be specified for graphic objects that are handled by an `IlvSelectInteractor` object. An `IlvSelectInteractor` object allows objects to be interactively selected in the manager, to be moved around, and to have their graphic properties edited.

Optimizing drawing tasks

A special manager feature minimizes the cost of drawing tasks to be done after geometric operations have been performed. This is useful in situations where you want to see the results of your work. This feature uses a region of invalidated parts of the display called the *update region*. The update region stores the appropriate regions before any modifications are carried out on objects, as well as those regions that are relevant after these modifications have been carried out for each view.

To successfully apply an applicable function, you must mark the regions where the objects are located as invalid, apply the function, and then invalidate the regions where the objects involved are now located (applying the function may change the location of the objects). This mechanism is greatly simplified by a set of methods of the `IlvManager` class. Regions to be updated are only refreshed when the method `reDrawViews` is called. This means that refreshing the views of a manager is done by marking regions to be redrawn in a cycle of `initReDraws` and `reDrawViews`.

These cycles can be nested so that only the last call to the method `reDrawViews` actually updates the display. The `IlvManager` methods that help you optimize drawing tasks are:

- `initReDraws()`

  Marks the beginning of the drawing optimization operation by emptying the region to update for each view being managed. Once this step is taken, direct or indirect calls to a draw instruction are deferred. For every `initReDraws`, there should be one call to `reDrawViews`, otherwise, a warning is issued. Calls to `initReDraws` can be embedded so that the actual refresh only takes place when the last call to `reDrawViews` is reached.

- `invalidateRegion(ilog.views.IlvGraphic)`

  Defines a new region as invalid, that is, this region will be redrawn later. Each call to `invalidateRegion` adds the region to the update region in every view.

- `reDrawViews()`

  Sends the drawing commands for the whole update region. All the objects involved in previous calls to `invalidateRegion` are then updated.

- `abortReDraws()`

  Aborts the mechanism of deferred redraws (for example, if you need to refresh the whole screen). This function resets the update region to empty. If needed, you should start again with an `initReDraws` call.

- `isInvalidating()`

  Returns `true` when the manager is in an `initReDraws`/`reDrawViews` state.

This mechanism is used in the `applyToObject` method.

In fact the call:

```
manager.applyToObject(obj, func, userArg, true);
```

is equivalent to:

```
manager.initReDraws();
manager.invalidateRegion(obj);
manager.applyToObject(obj, func, userArg, false);
manager.invalidateRegion(obj);
manager.reDrawViews();
```

The `invalidateRegion` method works with the bounding box of the object given as a parameter. When an operation applied to the object modifies its bounding box, `invalidateRegion` must be called twice: once before and once after the operation. For example, for a move operation, you must invalidate the initial region where the object was before being moved and invalidate the final region so that the object can be redrawn. In other situations, such as changing the background, only the call after the operation is necessary.

Listener for the content of the manager

When the content of the manager changes, the manager will fire a `ManagerContentChangedEvent` event. Any class can listen for the modification of the content of the manager by implementing the `ManagerContentChangedListener` interface.

This interface contains only the `contentsChanged` method.

```
void contentsChanged(ManagerContentChangedEvent evt)
```

This method is called when an object is added to or removed from the manager, or when the visibility, the bounding box, or the layer of a graphic object changes. A class that implements this interface will register itself by calling the `addManagerContentChangedListener(ilog.views.event.ManagerContentChangedListener)` method of the manager.

A `ManagerContentChangedEvent` can be of several types depending on the type of modification in the manager. For each type, there is a corresponding subclass of the class `ManagerContentChangedEvent`. The type of the event can be retrieved with the `getType` method of the class. The list of these subclasses is indicated below along with the type of change in the manager that is responsible for it:

- `ObjectInsertedEvent` (type `OBJECT_ADDED`)

  A graphic object has been inserted. You can retrieve the graphic object that was inserted with the `getGraphicObject` method.

- `ObjectRemovedEvent` (type `OBJECT_REMOVED`)

  A graphic object has been removed. You can retrieve the graphic object that was removed with the `getGraphicObject` method.

- `ObjectBBoxChangedEvent` (type `OBJECT_BBOX_CHANGED`)

  The bounding box of a graphic object has changed. You can retrieve the graphic object concerned using the `getGraphicObject` method and the old and new bounding box with the `getOldBoundingBox` and `getNewBoundingBox` methods.

- `ObjectLayerChangedEvent` (type `OBJECT_LAYER_CHANGED`)

  A graphic object has changed layers. You can retrieve the graphic object concerned using `getGraphicObject` and the old and new layer using the `getOldLayer` and `getNewLayer` methods.

- `ObjectVisibilityChangedEvent` (type `OBJECT_VISIBILITY_CHANGED`)

  The visibility of a graphic object has changed. You can retrieve the graphic object concerned using `getGraphicObject`. The method `isObjectVisible` will tell you the new state of the object.

A listener will cast the event depending on the type:

```
public void contentsChanged(ManagerContentChangedEvent event)
{
```

```
  if (event.getType() == ManagerContentChangedEvent.OBJECT_ADDED) {
    ObjectInsertedEvent e = (ObjectInsertedEvent)event;
    IlvGraphic object = e.getGraphicObject();
    ....
  }
}
```

As `ManagerContentChangedEvent` events can be sent very often (especially when numerous objects are being added as in the case of reading a file), the manager provides a way to notify the listeners that it is currently doing a series of modifications. In this case, the event will contain a flag telling the listener that the manager is currently performing several modifications. This flag can be tested using the `isAdjusting` method of the `ManagerContentChangedEvent` class. The manager will notify the listeners of the end of a series by sending a final `ManagerContentsChangedEvent` of type `ADJUSTMENT_END`.

Thus, a listener can decide to react to global modifications, but not to all individual modifications using the following code:

```
public class MyListener implements ManagerContentsChangedListener
{
  public void contentsChanged(ManagerContentChangedEvent event)
  {
     if (!event.isAdjusting()) {
        // do something
     }
  }
}
```

When making numerous modifications in a manager, you may want to be able to notify the listeners in the same way. To do so, you can use the `setContentsAdjusting` method of the manager in the following way:

```
manager.setContentsAdjusting(true);
try {
 //add a lot of objects
} finally {
  manager.setContentsAdjusting(false);
}
```

All operations done between the two calls to `setContentsAdjusting` will fire a `ManagerContentChangedEvent` event with the `isAdjusting` flag set to `true`. A call to the `setContentsAdjusting` method with the parameter set to `false` can send the file `ADJUSTMENT_END` event.

This mechanism can also help the internal listeners of IBM® ILOG® JViews to work in a more efficient way, so you are recommended to use it.

Events related to imminent content changes

Since JViews 8.1, `IlvManager` fires events when a graphic object is about to change or about to be deleted. These events are fired before the graphic objects are changed or deleted. To listen for these events, you need to implement the following interface.

```
public interface ManagerContentMonitor
  extends ManagerContentChangedListener
{
  public void contentAboutToChange(ManagerContentAboutToChangeEvent event);
}
```

As this interface extends `ManagerContentChangedListener`, you can install a `ManagerContentMonitor` by calling `addManagerContentChangedListener(ilog.views.event.ManagerContentChangedListener)`. `IlvManager` will call the method `contentAboutToChange()` that you have implemented when a graphic object is about to change or about to be deleted.

Selection in a manager

Describes how to select objects through the manager and display them as selected.

In this section

Selection objects
Explains how selection objects are used for displaying selected objects in a manager.

Managing selected objects
Describes how to select and deselect objects in a manager and perform related operations.

Creating your own selection object
Explains how a selection object is assigned and how to override the default behavior.

Listener for the selections in a manager
Describes how to be notified of changes to selections in a manager.

Selection objects

The manager allows you to select objects. To display selected objects within a manager, IBM® ILOG® JViews creates selection objects which are drawn on top of the selected objects. An example of a selection object is a set of handles drawn around the selected object.

Selection objects are stored in the manager. Unlike regular graphic objects, they are internally managed and cannot be manipulated.

When a graphic object is selected, a selection object is created and is drawn on top of the graphic object. Selection objects are subclass instances of the class `IlvSelection`. As such, they are also graphic objects. The class `IlvSelection` is an abstract class that has been subclassed to create several classes of selection objects specialized in the selection of specific graphic objects. For example, the class `IlvSplineSelection` is a selection object for the selection of an `IlvSpline` object.

The default selection object for graphic objects is an instance of the class `IlvDrawSelection`. This class draws eight handles around the object, one on each of the four sides and one on each corner.

Managing selected objects

To select or deselect a graphic object in a manager, use the `setSelected` method:

```
void setSelected(IlvGraphic obj, boolean select, boolean redraw)
```

Once an object has been selected, you can retrieve its selection object using:

```
IlvSelection getSelection(IlvGraphic obj)
```

This method returns `null` if the object does not have an associated selection object; in other words, if the graphic object is not selected. You can also use the following method to determine whether the object is selected or not:

```
boolean isSelected(IlvGraphic obj)
```

To obtain the selected object from the selection object, use the `getObject()` method of `IlvSelection`.

You can obtain an enumeration of all the selected objects in the manager with:

```
IlvGraphicEnumeration getSelectedObjects()
```

You can use this method as follows.

```
IlvGraphicEnumeration selectedobjs = manager.getSelectedObjects();
IlvGraphic obj;

while(selectedobjs.hasMoreElements()) {
    obj = selectedobjs.nextElement();
    //perform some action
}
```

Note: To avoid unpredictable results, you must not select or deselect graphic objects when stepping through the enumeration as in the example above.

Other methods of `IlvManager` allow you to select and deselect all objects in the manager or in a particular layer:

```
void selectAll(IlvManagerView view, boolean redraw)
```

```
void selectAll(boolean redraw)
```

```
void deSelectAll(boolean redraw)
```

```
void deSelectAll(int layer, boolean redraw)
```

Selection interactor

The library provides the `IlvSelectInteractor` class which allows you to select and deselect objects in an interactive way (using the mouse). It also allows you to edit graphic objects. For more information, see *The selection interactor*.

Creating your own selection object

The selection object depends on the graphic object. In fact, the manager creates the selection object using the following method of the graphic object:

```
IlvSelection makeSelection()
```

You can override this method to return your own instance of the selection object. Another possibility is to set an `IlvSelectionFactory` on the manager and let this factory decide which subclass of `IlvSelection` should be instantiated depending on the graphic object. The following is an example which creates a new selection object (a white border) around the selected object.

```
class mySelection extends IlvSelection
{
  static final int thickness = 3;
  mySelection(IlvGraphic obj)
  {
    super(obj);
  }

  public void draw(Graphics g, IlvTransformer t)
  {
    g.setColor(Color.white);
    IlvRect rect = boundingBox(t);
    for (int i = 0; i < thickness; i++) {
      if ((int)Math.floor(rect.width) >
           2*i && (int)Math.floor(rect.height) > 2*i)
         g.drawRect((int)Math.floor(rect.x)+i,
                    (int)Math.floor(rect.y)+i,
                    (int)Math.floor(rect.width)-2*i-1,
                    (int)Math.floor(rect.height)-2*i-1);
    }
  }

  public IlvRect boundingBox(IlvTransformer t)
  {
    // get the bounding rectangle of the selected object
    IlvRect bbox = getObject().boundingBox(t);
    bbox.x-= thickness;
    bbox.y-= thickness;
    bbox.width+= 2*thickness;
    bbox.height+= 2*thickness;
    return bbox;
  }

  public boolean contains(IlvPoint p, IlvPoint tp, IlvTransformer t)
  {
    return false;
```

```
    }
}
```

You can see that the selection object is defined in the same way as a graphic object. The constructor of a selection object always takes the selected object as a parameter. Note that the `boundingBox` method of the selection object uses the `boundingBox` method of the selected object so that the selection object (in this case, the white border) is always around the selected object, whatever the transformer is.

Listener for the selections in a manager

A class must implement the `ManagerSelectionListener` interface to be notified that selections in a manager have been modified. This interface contains only the `selectionChanged` method, which is called each time an object is selected or deselected.

```
void selectionChanged(ManagerSelectionChangedEvent event)
```

To be notified of selections and deselections, a class must register itself using the following method of the class `IlvManager`:

```
void addManagerSelectionListener(ManagerSelectionListener l)
```

Note that the `selectionChanged` method is called just after the object is selected or deselected, so you can easily determine whether it is a selection or a deselection. You do this in the following way:

```
class MyListener implements ManagerSelectionListener
{
  public void selectionChanged(ManagerSelectionChangedEvent event)
  {
    // retrieve the graphic object
    IlvGraphic obj  = event.getGraphic();
    IlvManager manager = event.getManager();
    if (manager.isSelected(obj)) {
      // object was selected
    } else {
      // object was deselected
    }
  }
}
```

When numerous objects are being selected, for example, as a result of a call to the `selectAll` method of the manager, many selection events will be sent to the selection listeners. This can be inefficient for some listeners that need to perform an action when the selection is stable. For example, a property inspector showing the properties of selected objects does not need to be updated for each individual selection when a number of objects are selected at the same time. To solve this kind of problem, the `ManagerSelectionChangedEvent` class has the following methods:

- `isAdjusting` to tell you if the event is part of a series of selection events.
- `isAdjustmentEnd` to indicate that the event is the last one of a series.

In the case of a “property inspector,” the listener would be as follows:

```
class MyListener implements ManagerSelectionListener
{
  public void selectionChanged(ManagerSelectionChangedEvent event)
  {
```

```
    if (!event.isAdjusting() || event.isAdjustmentEnd())
    {
       // update the properties only if this is a single
       // selection or the end of a series.
    }
  }
}
```

You may want to use the same “adjustment” notification when selecting numerous objects in a manager. The `IlvManager` class allows you to do this using the `setSelectionAdjusting` method:

```
boolean isAdjusting = manager.isSelectionAdjusting();
manager.setSelectionAdjusting(true);
try {
  // select or deselect a lot of objects.
} finally {
  manager.setSelectionAdjusting(isAdjusting);
}
```

Hover highlighting in a manager

Describes how to use hover highlighting in a top-level manager.

In this section

Managing hover highlighting
Describes how to set up and enable hover highlighting.

Creating your own highlighting effect
Describes how to create a Java 2D™ image to use as a highlighting effect and how to customize the effect.

Managing hover highlighting

The manager allows you to highlight objects when the pointer hovers on top of them.

To display highlighted objects within a manager, the manager creates specific images that are drawn on top of the selected objects as a highlighting effect. By default, hover highlighting is not enabled. If you want to activate it, you need first to decide what highlighting effect you want to use.

The images used as highlighting effects are transient Java 2D™ artifacts, they are internally managed and cannot be manipulated.

Each effect is applied through a Java 2D image operation on the regular object representation, and then displayed on top of all the objects with a set opacity.

The manager provides 5 predefined effects:

- Invert colors: This changes the intensity of each color component (red, green, blue). For example, a red object will be highlighted with a yellow color.
- Blur: The highlighted object becomes blurred.
- Brighten: Every color used when drawing the object becomes brighter. This may have no effect on objects that are already very bright.
- Gray scale: The colors of the object are converted into tones of grays.
- Sharpen: the borders of the object are accentuated.
- None: to remove the hover highlight effect.

A sixth effect (Custom) is detailed in *Creating your own highlighting effect*.

To select your hover highlight effect, call `setHoverHighlightingMode`.

Note:
1. The hover highlighting effect will only be used when set on a top-level manager (See Nested managers and nested graphers for more information on nested managers).
2. The hover highlighting mode and the operation - see *Creating your own highlighting effect* are not persistent. This means that the information is not stored in .ivl files.

Creating your own highlighting effect

If you do not want or cannot use (because of your color scheme, for example) a predefined effect, you can also build your own instance of `IlvHoverHighlightingImageOperation`.

This class allows you to define which Java 2D™ image operation you want to use to highlight your objects. It also contains methods to provide a filter, to indicate which objects will be highlighted, and an opacity (or alpha) if you want to see the regular object representation through the highlighting effect.

For example, here is an operation that will have a more blurred effect, be half transparent and only for nongraphic bag objects.

```
float[] blur5Kernel = new float[5 * 5];
    for (int i = 0; i < blur5Kernel.length; i++) {
      blur5Kernel[i] = 1f / blur5Kernel.length;
    }
  IlvHoverHighlightingImageOperation myOperation=new
          IlvHoverHighlightingImageOperation(
            new ConvolveOp(new Kernel(5, 5, blur5Kernel)), 1));
  myOperation.setAlpha(0.5f);
  myOperation.setHighlightFilter(new
IlvHoverHighlightingImageOperation.NonGraphicBagFilter());
```

You can set your operation instance on the manager by means of `setHoverHighlightingImageOperation`. The mode will then be known to the manager as "Custom".

Blinking of graphic objects

Describes the three types of blinking mode supported by IBM® ILOG® JViews.

In this section

Introduction
Briefly introduces the blinking mode.

Introduction

Blinking is the periodical change of the drawing of a graphic object. A blinking object drags attention and can be used to indicate a specific alarm state of an object. IBM® ILOG® JViews supports three kinds of blinking:

- visibility blinking: the object becomes periodically visible and invisible;
- color and paint blinking: the color or paint of an object changes periodically;
- blinking actions: an arbitrary property change is performed periodically on the object.

The blinking mode of a view determines whether a view displays the blinking effects. Usually it is not needed to display any blinking effect on the overview, and therefore the blinking can be switched off for this view by using the following code:

```
managerView.setBlinkingMode(IlvManagerView.BLINKING_DISABLED);
```

Visibility blinking

All graphic objects support visibility blinking. In this case, they are periodically shown and hidden. You simply have to set the blinking timing, as in the following example:

```
graphic.setBlinkingOnPeriod(1000);
graphic.setBlinkingOffPeriod(2000);
```

The object is now shown every 3 seconds: it is visible for 1 second and hidden for 2 seconds. All objects with the same blinking timing will blink synchronously. Since the blinking mode requires a periodical redraw, it is recommended to use the same timing for many objects when possible, otherwise the performance of the system will degrade by too many non-synchronized draw operations.

The blinking of the object starts when both the "on-period" and the "off-period" are not 0. The visibility blinking is a drawing mechanism and does not change the visible flag of the graphic object, that is, the method `isVisible()` called on the graphic object remains unchanged whether the blinking mode currently hides or shows the object.

Blinking colors and paints

The class `IlvBlinkingColor` represents a blinking color. It can be used as color of those properties of `IlvGraphic` objects that expect a `java.awt.Color` as parameter and are documented to support blinking colors. A blinking color changes the visible color periodically.

```
java.awt.Color color = new IlvBlinkingColor(Color.green, Color.blue, 1000,
1000);
IlvLine line = new IlvLine();
line.setForeground(color);
```

This line object switches periodically from green to blue every second. It is also possible to create colors or paints that switch between multiple states:

```
Color color = new IlvBlinkingMultiColor(1000, Color.blue, Color.red, Color.
green, Color.yellow);
Paint redGreen = new GradientPaint(0, 0, Color.red, 0, 100, Color.green);
Paint blueYellow = new GradientPaint(0, 0, Color.blue, 100, 0, Color.yellow);
Paint blackWhite = new GradientPaint(0, 0, Color.black, 0, 100, Color.white);
Paint paint1 = new IlvBlinkingPaint(redGreen, blueYellow, 1000, 2000);
Paint paint2 = new IlvBlinkingMultiPaint(1000, redGreen, blueYellow, blackWhite)
;
```

The first color switches from blue to red to green to yellow back to blue. `paint1` switches between 2 gradient paints, it stays 1 second red-green, then 2 seconds blue-yellow. `paint2` switches every second between 3 gradient paints.

The classes `IlvBlinkingColor`, `IlvBlinkingMultiColor`, `IlvBlinkingPaint`, `IlvBlinkingMultiPaint` can only be used in combination with `Java API` objects. They work if the set method of a property expects `java.awt.Color` or `java.awt.Paint` and is documented to support blinking colors or paints. They have no blinking effect when used as colors for other objects such as `JComponent` or `JPanel`.

Note: Since blinking requires a periodical redraw, it is recommended to use the same timing for many blinking colors and paints when possible, otherwise the performance of the system will degrade by too many non-synchronized draw operations.

Adding blinking facilities into your own IlvGraphic subclass

If you implement your own subclass of `Java API`, this subclass might have various colors or paints that are used to draw specific parts of the graphic objects. To enable these colors and paints to support blinking, you need to register them as blinking resources by calling the method

```
void registerBlinkingResource(Object oldResource, Object newResource);
```

You must register the colors and paints whenever they change, that is, in setter methods, copy constructors, stream-constructors and so on. Here is an example class that supports blinking color and paint properly:

```
import ilog.views.internal.impl.IlvUtility2D;

/**
 * A new class.
 */
public class MyClass extends IlvGraphic
{
    // the default color is not a blinking color
    private static Color _defaultColor = Color.black;

    private Color _color = _defaultColor;
    private Paint _paint = _defaultColor;

    /**
     * The default constructor.
```

```
     */
    public MyClass()
    {
        super();
        // the default color black does not blink, hence
        // no blinking resource must be registered.
    }

    /**
     * The copy constructor.
     */
    public MyClass(MyClass source)
    {
        super(source);
        Color oldColor = _color;
        Paint oldPaint = _paint;
        _color = source._color;
        _paint = source._paint;
        registerBlinkingResource(oldColor, _color);
        registerBlinkingResource(oldPaint, _paint);

        // or alternatively
        // setColor(source.getColor());
        // setPaint(source.getPaint());
        // then omit the additional calls of registerBlinkingResource
    }

    /**
     * The input stream constructor.
     */
    public MyClass(IlvInputStream stream) throws IlvReadFileException
    {
        super(stream);
        Color oldColor = _color;
        Paint oldPaint = _paint;
        _color = stream.readColor("color");
        _paint = stream.readPaint("paint");

        registerBlinkingResource(oldColor, _color);
        registerBlinkingResource(oldPaint, _paint);

        // or alternatively
        // setColor(stream.readColor("color"));
        // setPaint(IlvUtility2D.readPaint(stream, "paint", "p"));
        // then omit the additional calls of registerBlinkingResource
    }

    /**
     * Writes the object to an IVL file.
     */
    public void write(IlvOutputStream stream)
        throws IOException
    {
        super.write(stream);
```

```
        stream.write("color", _color);
        stream.writePaint(_paint, "paint", _defaultColor);
    }

    /**
     * Sets the color.
     * As Bean Property, you can use the property editor
     * ilog.views.util.beans.editor.IlvBlinkingColorPropertyEditor
     * which supports blinking.
     */
    public void setColor(Color c)
    {
        if (c == null)
            c = _defaultColor;

        Color oldColor = _color;
        _color = c;
        registerBlinkingResource(oldColor, c);
    }

    /**
     * Returns the color.
     */
    public Color getColor()
    {
        return _color;
    }

    /**
     * Sets the color.
     * As Bean Property, you can use the property editor
     * ilog.views.util.beans.editor.IlvBlinkingPaintPropertyEditor
     * which supports blinking.
     */
    public void setPaint(Paint p)
    {
        if (p == null)
            p = _defaultColor;

        Paint oldPaint = _paint;
        _paint = p;
        registerBlinkingResource(oldPaint, c);
    }

    /**
     * Returns the paint.
     */
    public Paint getPaint()
    {
        return _paint;
    }
}
```

Blinking actions

Visibility blinking and color blinking are optimized cases of blinking. In general, you can define an arbitrary action that is periodically performed on the object.

```
IlvMarker marker = new IlvMarker();
IlvBlinkingAction action = new IlvBlinkingAction(1000,1000) {
    protected void changeState(IlvGraphic obj, boolean isOn) {
        // no applyToObject necessary because the caller does it already for
us
        IlvMarker marker = (IlvMarker)obj;
        if (isOn) {
            marker.setType(IlvMarker.IlvMarkerCircle);
        } else {
            marker.setType(IlvMarker.IlvMarkerSquare);
        }
    }
};
marker.setBlinkingAction(action);
```

In this example, the marker type is periodically changed every second from circle to square. By using the class `IlvBlinkingMultiAction`, it is even possible to perform an arbitrary number of steps periodically on the graphic object.

Note: Blinking actions that change the bounding box of the graphic object work correctly but they are very inefficient. Since blinking requires a periodical redraw, it is recommended to use the same timings for many blinking actions when possible, otherwise the performance of the system will degrade by too many non-synchronized draw operations.

Managing input events

Describes how to handle input events using an interactor on the view or on a graphic object.

In this section

Handling input events
Describes various ways of handling input events.

Object interactors
Describes how to use object interactors for associating specific behavior with an object or set of objects.

Example: Extending the IlvObjectInteractor class
Describes how to extend the `IlvObjectInteractor` class with an example for dragging an object to a new position.

Customizing the interactor of selected graphic objects
Describes how to create a new graphic object with customized interactions.

View interactors
Describes how to use view interactors to handle view behavior.

Class diagrams for interactors and selection classes
Describes the relationships between classes used for interactors and selection.

Interactor listeners
Describes how to be notified when the active interactor of a view changes with an example of implementing a drag rectangle.

The selection interactor
Describes the predefined view interactor functionality and how to customize it.

Tooltips and popup menus on graphic objects
Describes how to implement tooltips and popup menus on graphic objects in Swing applications.

Handling input events

There are two ways of handling the input events in a manager view:

- An interactor can be set to the view using the class `IlvManagerViewInteractor`, which will handle all the events that occur on a view.
- An object interactor can be used on a graphic object. This interactor, an instance of the class `IlvObjectInteractor`, handles the events occurring on a particular object if no view interactor has been installed on the view.

Tooltips and popup menus are handled by the `IlvToolTipManager` and `IlvPopupMenuManager` central managers. They are neither manager view interactors nor object interactors. `IlvToolTipManager` and `IlvPopupMenuManager` listen for the specific events that trigger tooltip or a popup menu display in the registered view.

Object interactors

When you want to associate a specific behavior with an object, you can use an object interactor (class `IlvObjectInteractor` and its subclasses). Whenever an event is received by a manager view that has no associated view interactor, the manager attempts to send it to an object by a call to an attached object interactor. If there is an object at the event location, and if this object is connected to an object interactor, the manager sends the event to that interactor. If the interactor does not manage this event, or if the situation is not applicable, the manager tries to handle the event by means of accelerators.

You can create an `IlvObjectInteractor` instance and bind it to an object or a set of objects using the `IlvGraphic` method `setObjectInteractor`. As soon as this binding occurs, the object receives user events and deals with them, therefore it is the interactor and not the object itself that manages these events.

Querying, setting, or removing an object interactor can be done by means of calls to the following methods on the `IlvGraphic` instance:

```
IlvObjectInteractor getObjectInteractor()
```

```
void setObjectInteractor(IlvObjectInteractor interactor)
```

An instance of `IlvObjectInteractor` can be shared by several graphic objects. This allows you to reduce the amount of memory needed to handle the same interaction on a large number of graphic objects. To share the same object interactor instance, do not use `new` to create your interactor; use the `Get(java.lang.String)` method of the class `IlvObjectInteractor`.

Example: Extending the IlvObjectInteractor class

The `MoveObjectInteractor` class defined in this example allows you to move an object using the mouse, moving it to where you release the mouse button. You can see the complete code of the example in `MoveObjectInteractor.java` located at `codefragments/interactors/moveobjinter/src/MoveObjectInteractor.java` in the installed product. For details, see ***<installdir>* /jviews-framework86/codefragments/interactors/moveobjinter/index.html**.

```
public class MoveObjectInteractor extends IlvObjectInteractor
{
  private IlvRect mrect;
  private float dx, dy;
  private boolean dragging = false;

  /** Creates an moveObjectInteractor. */
  public MoveObjectInteractor()
  {
    super();
  }
  ...
}
```

The `MoveObjectInteractor` extends the `IlvObjectInteractor` class and defines the following attributes:

- The attribute `mrect` specifies the future bounding rectangle of the graphic object. This data is updated each time the object is dragged.
- The `dx` and `dy` attributes represent the translation from the original clicked point and the current top-left corner of the graphic object.
- The Boolean value `dragging` is set to `true` when the user starts dragging the object.

Events are processed by the `processEvent` method as follows.

```
protected boolean processEvent(IlvGraphic obj, AWTEvent event,
                               IlvObjectInteractorContext context)
{
   switch (event.getID())
   {
     case MouseEvent.MOUSE_PRESSED:
       return processButtonDown(obj, (MouseEvent)event, context);
     case MouseEvent.MOUSE_DRAGGED:
       return processButtonDragged(obj, (MouseEvent)event, context);
     case MouseEvent.MOUSE_RELEASED:
       return processButtonUp(obj, (MouseEvent)event, context);
     default:
       return false;
```

```
    }
}
```

The `processEvent` method dispatches events to three different methods depending on their type. The `processEvent` method takes a graphic object, an event, and a context as its parameters. The context `IlvObjectInteractorContext` is an interface that must be implemented by classes allowing the use of an object interactor. The class `IlvManager` takes care of that and passes a context object to the object interactor. From a context, you can get a transformer, change the mouse cursor, and so on.

In the `processButtonDown` method, the distance between the clicked point and the current top-left corner of the graphic object is stored in the attributes `dx` and `dy`. Note that the positions are stored in the view coordinate system. The top-left corner of the graphic object is extracted from the bounding rectangle of the object, which is computed using the `boundingBox` method. The `invalidateGhost` method is then called. This method requests the interactor context to redraw the region corresponding to the current bounding rectangle (stored in `mrect`).

```
public boolean
processButtonDown(IlvGraphic obj,
                  MouseEvent event,
                  IlvObjectInteractorContext context)
{
  if ((event.getModifiers() & InputEvent.BUTTON2_MASK) != 0 ||
      (event.getModifiers() & InputEvent.BUTTON3_MASK) != 0)
    return true ;
  if (dragging)
    return true ;
  dragging = true;
  IlvPoint p = new IlvPoint(event.getX(), event.getY());
  mrect = obj.boundingBox(context.getTransformer());
  dx = p.x - mrect.x;
  dy = p.y - mrect.y;
invalidateGhost(obj, context);
  return true;
}
```

The `invalidateGhost` method is implemented as follows..

```
private void
invalidateGhost(IlvGraphic obj,
                IlvObjectInteractorContext context)
{
  if (obj == null || context == null)
    return;
  if (mrect == null || mrect.width == 0 || mrect.height == 0)
    return;
  IlvRect invalidRegion = new IlvRect(mrect);
  context.repaint(invalidRegion);
}
```

The principle for drawing and erasing the ghost is the following: the interactor invalidates regions of the context (the bounds of the ghost before and after any position change), while

the drawing of the ghost is only performed when requested by the drawing system. Indeed, the method `handleExpose`, defined on the base class `IlvObjectInteractor`, is called only when the view is redrawn. This method, whose implementation in the base class does nothing, is overridden as follows.

```
public void handleExpose(IlvGraphic obj,
                         Graphics g,
                                  IlvObjectInteractorContext context)
{
  drawGhost(obj, g, context);
}
```

The actual drawing of the ghost is done by the following `drawGhost` method..

```
protected void drawGhost(IlvGraphic obj,
                         Graphics g,
                         IlvObjectInteractorContext context)
{
  if (mrect != null) {
    g.setColor(context.getDefaultGhostColor());
    g.setXORMode(context.getDefaultXORColor());
    IlvTransformer t = context.getTransformer();
    IlvRect r = obj.boundingBox(t);
    IlvTransformer t1 = new IlvTransformer(new IlvPoint(mrect.x - r.x,
                                                        mrect.y -r.y));
    t.compose(t1);
    obj.draw(g, t);
  }
}
```

The `Graphics` object that is passed as an argument is set to XOR mode using the default XOR color and ghost color defined in the context. Next, a transformer is computed that will draw the object in the desired location given by `mrect`. Note that the object is not translated, only drawn at another location. The method does nothing if `mrect` is `null`. This prevents the ghost from being drawn if the `drawGhost` method happens to be called after the end of the interaction.

Mouse dragged events are handled as follows.

```
protected boolean
processButtonDragged(IlvGraphic obj, MouseEvent event,
                     IlvObjectInteractorContext context)
{
  if (!dragging || mrect == null)
    return false;
  IlvPoint p = new IlvPoint(event.getX(), event.getY());
  invalidateGhost(obj, context);
  mrect.move(p.x - dx, p.y - dy);
  IlvTransformer t = context.getTransformer();
  if (t != null)
    t.inverse(mrect);
  context.ensureVisible(p);
```

```
  t = context.getTransformer();
  if (t != null)
    t.apply(mrect);
  invalidateGhost(obj, context);
  return true;
}
```

First the current ghost is invalidated by a call to `invalidateGhost`. The new required location is changed to the position of the mouse, translated with the original translation.

```
mrect.move(p.x - dx, p.y - dy);
```

Then `ensureVisible` is called on the context. If the current dragged point is outside the visible area of the view, this will scroll the view of the manager so that the dragged point becomes visible. This operation may change the transformer of the view, as the value of `mrect` is stored in the coordinate system of the view. Before `ensureVisible` is called, the value of `mrect` is transformed to the manager coordinate system as follows.

```
IlvTransformer t = context.getTransformer();
if (t != null) t.inverse(mrect);
```

After the call to `ensureVisible`, the value of `mrect` is transformed back to the view coordinate system as follows.

```
t = context.getTransformer();
if (t != null) t.apply(mrect);
```

The actual moving of the graphic object is done when the mouse button is released. The mouse released event is handled like this:

```
protected boolean
processButtonUp(IlvGraphic obj,
                MouseEvent event,
                IlvObjectInteractorContext context)
{
  if (!dragging || mrect == null)
    return true;
  dragging = false;
  invalidateGhost(obj, context);
  doMove(obj, context);
  mrect = null;
  return true;
}
```

First the ghost is invalidated by calling the `invalidateGhost` method. Then the `doMove` method is called. This method updates the position of the graphic object according to the final coordinates of `mrect`. After moving the object, `mrect` is set to `null` to prevent further drawing of the ghost.

The implementation of the method `doMove` is as follows

```
void doMove(IlvGraphic graphic,
            IlvObjectInteractorContext context)
{
  if (mrect == null)
    return;
  IlvTransformer t = context.getTransformer();
  if (t != null)
    t.inverse(mrect);
  graphic.getGraphicBag().moveObject(graphic, mrect.x,
                                     mrect.y, true);
}
```

The value of `mrect` is translated to the coordinate system of the manager as follows.

```
IlvTransformer t = context.getTransformer();
if (t != null)
  t.inverse(mrect);
```

You should never try to change the position or the shape of a managed graphic object directly (or, more precisely, to modify its bounding box), for example by calling methods of the graphic object directly. Such changes must be done through a function, in this case `moveObject`, which is applicable to managers and takes all the necessary precautions. For further information, see *Modifying geometric properties of objects*.

Customizing the interactor of selected graphic objects

The selection interactor (`IlvSelectInteractor`) is also a view interactor. It allows you to select, move and reshape objects. The way an object is reshaped depends on the type of the object. For example, you can only change the size of an `IlvRectangle` while you can add points to an `IlvPolygon`. The dependency of possible interaction on the type of the object comes through the object interactor associated with the selection of the object.

If an object is selected, the method `makeSelection` is called to create a suitable selection object (subclass of `IlvSelection`. This selection object is drawn on top of the selected object.

Since `IlvSelection` is a subclass of `IlvGraphic`, it can have object interactors. When the select interactor moves the mouse over a selection object, it queries for the object interactor of the selection object.

If no object interactor is explicitly set on the selection object, it calls `getDefaultInteractor` to retrieve the class name of the default interactor and then sets the object interactor of the selection object to an instance of the default interactor.

Then, the select interactor forwards all events to the object interactor of the selection. This object interactor receives the mouse events as long as the mouse is over the selection object.

The object interactor of the selection object can react on the received events by reshaping the original selected object. In most cases, the object interactor of the selection object does not modify the selection object itself but rather the original selected object.

To create a new graphic object class with a customized selection and customized reshape interaction on the selected object:

1. Create your new derived class of `IlvGraphic` as described in *Creating a new graphic object class*.
2. Create a new derived class of `IlvSelection` as described in *Creating your own selection object*.
3. Create a new object interactor that works on the selected object instead of the selection.
4. In the derived graphic class, override `makeSelection` to return an instance of the new selection class.
5. In the new selection class, override `getDefaultInteractor` to return the class name of the new object interactor. Alternatively, you could call `setObjectInteractor` on the selection object when it is allocated. The former is more convenient if you want to have the same interactor for all instances of the new subclass of `IlvSelection`, while the latter can be used if you want to assign specific object interactors to specific instances of the selection.

The following code example creates a new graphic subclass `MyMarker` that has a special selection object `MyMarkerSelection`. If the marker is selected, the object interactor `MyMarkerEdition` becomes active. Each click in the marker selection changes the type (and therefore also the shape) of the marker.

```
public class MyMarker extends IlvMarker
{
   ...
```

```
  public IlvSelection makeSelection()
  {
    return new MyMarkerSelection(this);
  }
}

public class MyMarkerSelection extends IlvUnresizeableDrawSelection
{
  ...

  public String getDefaultInteractor()
  {
    return MyMarkerEdition.class.getName();
  }
}

public class MyMarkerEdition extends IlvReshapeSelection
{
  public MyMarkerEdition()
  {
    super();
  }

  protected boolean handleButtonDown(IlvDrawSelection sel, MouseEvent event,
                                     IlvObjectInteractorContext context)
  {
    // each click with the left mouse button into the selection object
    // changes the type of the selected object

    if ((event.getModifiers() & InputEvent.BUTTON2_MASK) != 0 ||
        (event.getModifiers() & InputEvent.BUTTON3_MASK) != 0)
      return true;

    MyMarkerSelection msel = (MyMarkerSelection)sel;
    MyMarker marker = (MyMarker)msel.getObject();

    // even though the object interactor is on the selection, it
    // modifies the selected object, not the selection
    final int tp = (marker.getType() >= 512 ? 1 : marker.getType() * 2);
    IlvGraphicBag bag = marker.getGraphicBag();
    if (bag == null)
      marker.setType(tp);
    else
      bag.applyToObject(marker, new IlvApplyObject() {
          public void apply(IlvGraphic g, Object arg) {
            ((MyMarker)g).setType(tp);
          }
        }, null, true);
    return true;
  }
}
```

View interactors

The `IlvManagerViewInteractor` class handles view behavior. The role of this class is to handle complex sequences of user input events that are to be processed by a particular view object.

View interactor methods

You can add or remove a view interactor with the following methods:

```
IlvManagerViewInteractor getInteractor()
```

```
void setInteractor(IlvManagerViewInteractor inter)
```

```
void pushInteractor(IlvManagerViewInteractor inter)
```

```
IlvManagerViewInteractor popInteractor()
```

Predefined view interactors

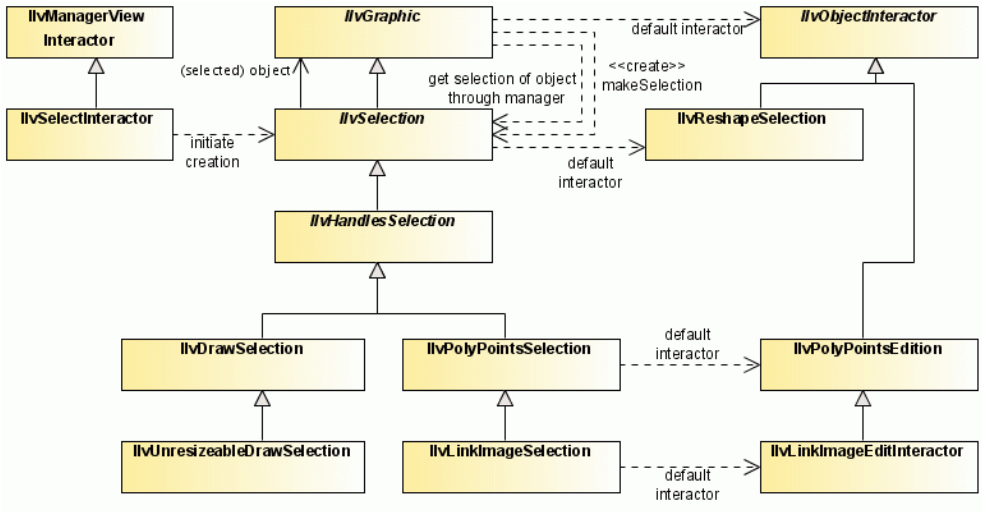
IBM® ILOG® JViews provides predefined view interactors. Following is a list of these interactors:

- `IlvDragRectangleInteractor` - Draws a rectangle that can be used for several purposes. See *Example: Implementing the DragRectangleInteractor class*.
- `IlvMakeRectangleInteractor` - Allows creation of `IlvRectangle` objects.
- `IlvMakeArcInteractor` - Allows creation of `IlvArc` objects.
- `IlvMakeEllipseInteractor` - Allows creation of `IlvEllipse` objects.
- `IlvMakeReliefRectangleInteractor` - Allows creation of objects of the `IlvReliefRectangle` class.
- `IlvMakeRoundRectangleInteractor` - Allows creation of objects of the `IlvRoundRectangle` class with round corners.
- `IlvUnZoomViewInteractor` - Allows the unzooming command. You have to draw a rectangular region into which the area you are watching is unzoomed.
- `IlvZoomViewInteractor` - Allows the zooming command. You draw a rectangular region where you want to zoom.
- `IlvMakePolyPointsInteractor` - Allows creation of polypoints objects.
- `IlvMakeLineInteractor` - Allows creation of objects of the `IlvLine` class.

- `IlvMakeArrowLineInteractor` - Allows creation of objects of the `IlvArrowLine` class.
- `IlvMakeLinkInteractor` - Allows creation of objects of the `IlvLinkImage` class.
- `IlvMakePolyLinkInteractor` - Allows creation of objects of the `IlvPolylineLinkImage` class.
- `IlvMakePolygonInteractor` - Allows creation of objects of the `IlvPolygon` class.
- `IlvMakePolylineInteractor` - Allows creation of objects of the `IlvPolyline` class.
- `IlvMakeArrowPolylineInteractor` - Allows creation of objects of the `IlvArrowPolyline` class.
- `IlvMakeSplineInteractor` - Allows creation of objects of the `IlvSpline` class.
- `IlvEditLabelInteractor` - Allows creation and editing of objects that implement the `IlvLabelInterface` such as `IlvLabel` or `IlvZoomableLabel`.
- `IlvMoveRectangleInteractor` - Drags a rectangle and performs an action when releasing the mouse button.
- `IlvSelectInteractor` - Allows selection and editing of graphic objects.
- `IlvRotateInteractor` - Allows rotation of a selected graphic object.
- `IlvPanInteractor` - Allows translation of a view without using scroll bars.
- `IlvMagnifyInteractor` - Allows magnification of part of the view under the mouse pointer.

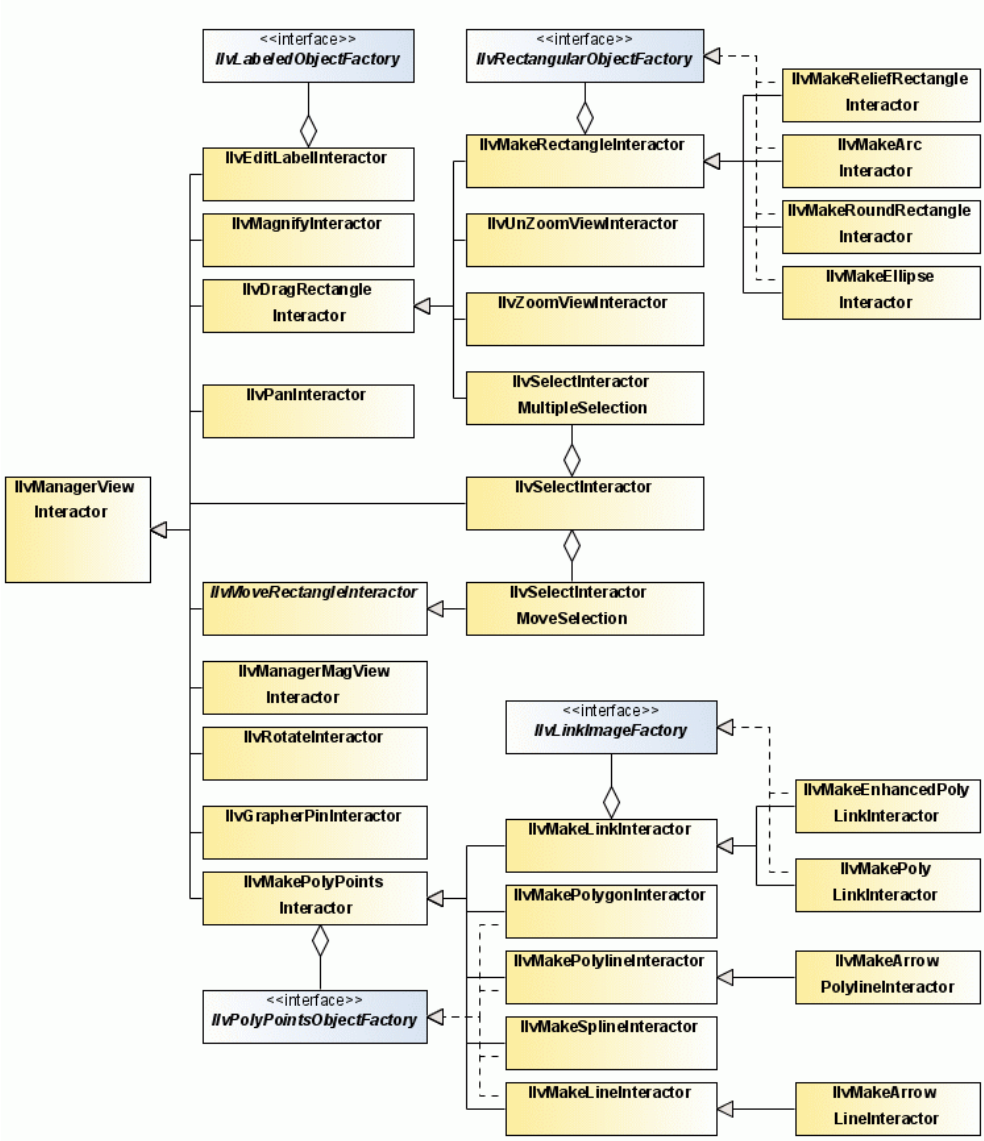
Class diagrams for interactors and selection classes

The most important selection objects and corresponding object interactors summarizes the relationships between `IlvObjectInteractor` and `IlvSelection`. Each graphic object can have an object interactor that handles the interactions. When an object is selected by `IlvSelectInteractor`, an `IlvSelection` object is created for the selected object through the method `makeSelection`. With the help of the selection object, the selected object can be manipulated, for example, it can be reshaped. Thus, the selection object is associated with a default object interactor. Different subclasses of `IlvSelection` have different default object interactors.



The most important selection objects and corresponding object interactors

The view interactor classes shows the different subclasses of `IlvManagerViewInteractor`. Most interactors are used to create certain kinds of object, such as `IlvMake`. Other interactors allow the view to be zoomed in or out. The class `IlvSelectInteractor` allows graphic objects to be selected. It delegates some functionality to `IlvSelectInteractorMoveSelection` and `IlvSelectInteractorMultipleSelection`.



The view interactor classes

Interactor listeners

When the active interactor of a view changes, the view fires an `InteractorChangedEvent` event. A class must implement the `InteractorListener` interface in order to be notified that a view interactor has been modified and must register itself using the `addInteractorListener(ilog.views.event.InteractorListener)` method of `IlvManagerView`. You can also specify that the listener no longer be notified of such events by using the `removeInteractorListener(ilog.views.event.InteractorListener)` method.

When the interactor of a view changes, the view calls the `interactorChanged` method of the listeners.

```
void interactorChanged(InteractorChangedEvent event)
```

This method is called with an instance of the class `InteractorChangedEvent` as a parameter containing information on the new and the old interactor.

Example: Implementing the DragRectangleInteractor class

This example shows how the methods of the predefined view interactor `IlvDragRectangleInteractor` are implemented. You can use this example as a starting point for creating your own interactor functionality. The class `IlvDragRectangleInteractor` is used to specify a rectangular region in a view. When this rectangle is selected, the `fireRectangleDraggedEvent(ilog.views.IlvRect, java.awt.event.MouseEvent)` method is called. The rectangle can then be used for various purposes in derived interactors. For example, you can create a subclass of this interactor to zoom in on the selected area. You can see the complete code example file `DragRectangleInteractor.java` located at `codefragments/interactors/dragrectinter/src/DragRectangleInteractor.java` in the installed product. For details, see **<*installdir*> /jviews-framework86/codefragments/interactors/dragrectinter/srchtml/DragRectangleInteractor.java.html**.

The `DragRectangleInteractor` class defines the following attributes: `start`, `rectangle`, and `dragging`.

```
public class DragRectangleInteractor extends
                                          IlvManagerViewInteractor {
  /**  The anchor point of the rectangle. */
  private final IlvPoint start = new IlvPoint();
  /**  The rectangle when dragging. */
  private final IlvRect rectangle = new IlvRect();
  /**  True if dragging. */
  private boolean dragging = false;
...
}
```

The attribute `start` is the point at the start of the drag action; `rectangle` is the rectangle that is drawn when dragging; `dragging` is a Boolean variable whose value is `true` when the user drags.

The `enableEvents` method called in the constructor takes the `MOUSE_EVENT_MASK` and `MOUSE_MOTION_EVENT_MASK` as parameters. Events must be enabled to be taken into account by the interactor:

```
public DragRectangleInteractor()
{
  enableEvents(AWTEvent.MOUSE_EVENT_MASK |
                                  AWTEvent.MOUSE_MOTION_EVENT_MASK);
}
```

The `processMouseEvent` method handles the `MOUSE_PRESSED` and `MOUSE_RELEASED` events:

```
protected void processMouseEvent(MouseEvent event)
{
  switch (event.getID()) {
    case MouseEvent.MOUSE_PRESSED:
    {
      if (dragging) break;
      if ((event.getModifiers() & InputEvent.BUTTON2_MASK) == 0 &&
          (event.getModifiers() & InputEvent.BUTTON3_MASK) == 0)
      {
        dragging = true;
        IlvTransformer t = getTransformer();
        start.move(event.getX(), event.getY());
        t.inverse(start);
        rectangle.width = 0;
        rectangle.height = 0;
      }
      break;
    }
    case MouseEvent.MOUSE_RELEASED:
      if (dragging) {
        dragging = false;
        drawGhost();
        rectangle.width = 0;
        rectangle.height = 0;
          fireRectangleDraggedEvent(new IlvRect(rectangle), event);
      }
   }
}
```

When the mouse button is pressed, the mouse pointer coordinates are stored in the `start` variable and are converted for storage in the coordinate system of the manager. When the mouse is released, the `drawGhost` method of `IlvManagerViewInteractor` is called to erase the ghost image. The width and height of the rectangle are set to `0` to prevent further drawings of the ghost, and the `fireRectangleDraggedEvent` method is called to notify the end of the drag operation. The following code demonstrates the dragged rectangle.

Note: The `drawGhost()` method can be used to perform a temporary drawing that gives the user feedback on the action of his present operation.

The `processMouseMotionEvents` handles the `MOUSE_DRAGGED` events:

```
protected void processMouseMotionEvent(MouseEvent event)
{
  if (event.getID() == MouseEvent.MOUSE_DRAGGED && dragging) {
    drawGhost();
    IlvTransformer t = getTransformer();
    IlvPoint p = new IlvPoint(event.getX(), event.getY());
    ensureVisible(p);
    rectangle.reshape(start.x, start.y, 0,0);
    t.inverse(p);
    rectangle.add(p.x, p.y);
    drawGhost();
  }
}
```

First the rectangle is erased by a call to `drawGhost`. The call to `ensureVisible` ensures that the dragged point remains visible on the screen. The new rectangle is then computed in the coordinate system of the manager and `drawGhost` is called to draw the new rectangle.

The `drawGhost` method simply draws the dragged rectangle. Since the rectangle is in the manager coordinate system, the method needs to apply the view transformer before drawing.

```
protected void drawGhost(Graphics g)
{
  IlvRect rect = new IlvRect(rectangle);
  IlvTransformer t = getTransformer();
  if (t != null)
    t.apply(rect);
  if (rect.width > 0 && rect.height >0) {
    g.drawRect((int)Math.floor(rect.x), (int)Math.floor(rect.y),
               (int)Math.floor(rect.width),
               (int)Math.floor(rect.height));
  }
}
```

The selection interactor

The IBM® ILOG® JViews library provides a predefined view interactor, `IlvSelectInteractor`, for selecting and editing graphic objects in a manager. This class allows you to:

- Select an object by clicking on it.
- Select or deselect several objects using Shift-Click.
- Select several objects by dragging a rectangle around them.
- Move one or several objects by selecting them and dragging the mouse.
- Edit objects by manipulating their selection object.

The interactor can be customized to your needs, as follows:

- You can enable or disable multiselection using:

```
public void setMultipleSelectionMode(boolean v)

isMultipleSelectionMode()
```

- You can select the mode for selecting objects by dragging a rectangle around them: opaque or ghost:

```
public void setOpaqueDragSelection(boolean o)

public boolean isOpaqueDragSelection()
```

- You can select the mode for moving graphic objects: opaque or ghost:

```
public void setOpaqueMove(boolean o)

public boolean isOpaqueMove()
```

- You can select the mode for resizing graphic objects: opaque or ghost:

```
public void setOpaqueResize(boolean o)

public boolean isOpaqueDragSelection()
```

- You can select the mode for editing polypoint objects: opaque or ghost:

```
public void setOpaquePolyPointsEdition(boolean o)
```

```
public boolean isOpaqueDragSelection()
```

- You can specify the modifier that allows multiple selection:

```
public void setMultipleSelectionModifier(int m)
```

```
public boolean getMultipleSelectionModifier()
```

- You can specify the modifier that allows selection by dragging a rectangle starting from a point on top of a graphic object:

```
public void setSelectionModifier(int m)
```

```
public boolean getSelectionModifier()
```

- You can allow selection of several objects using a dragged rectangle:

```
public void setDragAllowed(boolean v)
```

```
public boolean isDragAllowed()
```

- You can change the ability to move objects by:

```
public void setMoveAllowed(boolean v)
```

```
public boolean isMoveAllowed()
```

- You can change the ability to edit objects by:

```
public void setEditionAllowed(boolean v)
```

```
public boolean isEditionAllowed()
```

Note: The ability to move, edit, or select an object can be controlled object by object using the properties of the object.

Many other customizations can be done by subclassing the interactor and overriding the appropriate method.

The editing of a graphic object is controlled by an object interactor. When a graphic object is selected, the manager can dispatch events occurring on the selection object to the object interactor attached to the selection object. This object interactor is created by the selection object with a call to the `getDefaultInteractor()` method of the class `IlvSelection`. When creating your own selection object, you can also create an object interactor to edit the selected object. The default object interactor is the class `IlvReshapeSelection`, which allows the user to reshape graphic objects by pulling the handles of the objects.

Tooltips and popup menus on graphic objects

IBM® ILOG® JViews provides facilities to specify tooltips and popup menus for graphic objects in Swing applications. Tooltips and popup menus are handled by the `IlvToolTipManager` and the `IlvPopupMenuManager` central managers. These managers control events that trigger tooltip or popup menu display; they are not implemented as object interactors or manager view interactors. Tooltips and popup menus can be used in combination with any interactor.

Tooltips

In order to specify a tooltip for a graphic object, set the tooltip text as follows:

```
graphic.setToolTipText("Some Tooltip");
```

Tooltips only work when the view is registered with the tooltip manager. In order to enable tooltips in a manager view, register the view as follows:

```
IlvToolTipManager.registerView(managerView);
```

After a graphic object is registered, whenever a user holds the mouse over the object, the tooltip appears. When the mouse is moved away from the graphic, the tooltip disappears.

The `IlvToolTipManager` relies on the Swing tooltip manager. Parameters such as the initial delay or the dismiss delay can be set on the Swing tooltip manager. For example:

```
IlvToolTipManager.getToolTipManager().setInitialDelay(3000);
```

For more information, see `IlvToolTipManager`.

Popup menus

In order to associate a specific popup menu with a graphic object, create a `JPopupMenu` object and link it to a graphic as follows:

```
graphic.setPopupMenu(popupMenu);
```

Popup menus only work when the view is registered with the popup menu manager. In order to enable popup menus in a manager view, register the view as follows:

```
IlvPopupMenuManager.registerView(managerView);
```

After the popup menu is registered, whenever a user right-clicks the grapic, its popup menu appears.

Popup menus use a lot of memory. To avoid wasting memory, share popup menus between multiple graphic objects. To do this, instead of registering a popup menu with an individual

graphic using `graphic.setPopupMenu(...)`, register the popup menu directly with the popup menu manager by calling:

```
IlvPopupMenuManager.registerMenu("name", popupMenu);
```

You then assign this popup menu to graphics in the following way:

```
graphic1.setPopupMenuName("name");
graphic2.setPopupMenuName("name");
```

Popup menus registered with a single graphic object are active for that object only. Popup menus registered with the popup menu manager can be:

- Saved in `.ivl` files.
- Used for cut and paste operations.

When a popup menu is shared, you need to know which graphic object triggered the event. The action listeners associated with popup menu items can retrieve the context of the popup menu using an `IlvPopupMenuContext` object. This is done in the following way:

```
public void actionPerformed(ActionEvent e) {
    JMenuItem m = (JMenuItem)e.getSource();
    IlvPopupMenuContext context=IlvPopupMenuManager.getPopupMenuContext(m);
    if (context == null) return;
    IlvGraphic graphic = context.getGraphic();
    IlvManagerView view = context.getManagerView();
    //Do the action on this graphic for this view.
  }
```

For more information, see `IlvPopupMenuContext` and `IlvPopupMenuManager` in the *Java API Reference Manual*.

`IlvSimplePopupMenu` is a subclass of `JPopupMenu` that allows you to configure popup menus easily. For more information, see `IlvSimplePopupMenu`.

An example that illustrates the different ways of using popup menus is available as part of the demonstration software. For details, see ***<installdir>*** **`//jviews-framework86/codefragments/popupmenu/index.html`**.

Saving and reading

The manager provides facilities to save its contents to a file. The resulting file is an ASCII or binary file in the `.ivl` format that contains information about the layers and the graphic objects.

The saving methods are as follows:

```
void write(OutputStream stream, boolean binary) throws IOException
```

```
void write(String filename) throws IOException
```

```
void write(String filename, boolean binary) throws IOException
```

You can save data in either an ASCII or a binary file, the binary format being more compact and faster to read than the ASCII format.

The loading methods are as follows:

```
void read(InputStream stream) throws IOException, IlvReadFileException
```

```
void read(String filename) throws IOException, IlvReadFileException
```

```
void read(URL url) throws IOException, IlvReadFileException
```

The `read` methods may throw an exception in the following situations:

- The file is not an `.ivl` file.
- The `.ivl` format is not correct.
- A graphic class cannot be found.

The `read` methods detect automatically whether the `.ivl` file is an ASCII or a binary file.

You can save/read the information about your own graphic objects by providing the appropriate methods when creating your own graphic object class. For more information, see *Input/output operations* and also *Saving and loading the object description*.

Important: The recommended way to serialize any `IlvManager` object is through IVL serialization and not Java™ serialization. Serialization cannot work for managers that contain graphic objects such as `IlvIcon` or some other classes, since these classes manage internally Java SE objects that are not serializable.

File formats

JViews Framework provides support for various file formats as well as the `.ivl` format. Some file formats are specific to certain application domains and are explained in the corresponding user's documentation of specific IBM® ILOG® JViews products; for example:

- GIS formats in maps are explained in *Programming with JViews Maps*.
- BPMN file formats are explained in *Integrating BPMN Facilities* of IBM® ILOG® JViews Diagrammer.

The following general purpose file formats are also supported:

- SVG (reading and writing): see Deploying IBM® ILOG® JViews applications as SVG thin clients in the Advanced Features of JViews Framework.
- DXF (reading only): documented in *Drawing Exchange Format (DXF)*.

Drawing Exchange Format (DXF)

The Drawing Exchange Format (DXF) is the exchange format of AutoCAD. This format supports vector graphics (such as polygons, arcs, lines, and points) and layers. The different editions of the specifications of the DXF format corresponding to the various AutoCAD releases can be accessed from the following URL: *http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=5129239*.

Using the DXF reader

Reading a DXF File into IlvManager shows how to read the content of a DXF file into an `IlvManager` object.

Reading a DXF File into IlvManager

```
IlvManager manager = new IlvManager(0); //with no layer
IlvDXFReader reader = new IlvDXFReader();
try {
  reader.read("myDXFFile.dxf", manager);
} catch (IOException e) {
  e.printStackTrace();
}
```

You can also read a DXF file directly with the `read` method of the manager. First, you must create the manager and the stream factory as shown in *Preparing to Read a DXF File with the Manager*.

Preparing to Read a DXF File with the Manager

```
IlvManager manager = new IlvManager(0); // with no layer
IlvDXFStreamFactory factory = new IlvDXFStreamFactory();
manager.setStreamFactory(factory);
```

When the stream factory is set, calling `read(java.lang.String)` loads a DXF file instead of an IBM® ILOG® JViews IVL file. See *Loading a DXF File*.

Loading a DXF File

```
try {
  manager.read("myDXFFile.dxf");
} catch (IOException ex) {
  ex.printStackTrace();
}
```

The content of the DXF file can be read into an `IlvManager` object or into any implementation of the `IlvGraphicBag` interface, such as `IlvGraphicSet`. The layer information of the DXF file is ignored when the file is read into anything other than an `IlvManager` object.

The `IlvDXFReader` reads the DXF file and adds the graphic objects defined in the DXF file to the manager. If an error occurs during this process, an exception of the type `IOException` can occur and must be caught.

An example of the use of the reader is available as part of the demonstration software. For details, see ***<installdir>*** **/jviews-framework86/samples/dxfreader/index.html**.

Customizing the DXF reader

Configuration options can be set on `IlvDXFReaderConfigurator`. A default configurator is created when the default constructor of `IlvDXFReader` is used. You can also pass your own instance of a configurator to the constructor `IlvDXFReader(IlvDXFReaderConfigurator)`. You can retrieve the current instance of the configurator by using `IlvDXFReader.getConfigurator()`.

The reader delegates the conversion of DXF entities into IBM® ILOG® JViews graphic objects to a factory, `IlvDXFGraphicFactory`.

A default implementation is provided:

```
ilog.views.dxf.IlvDefaultDXFGraphicFactory
```

You can provide your own implementation or specialize the default implementation.

To set a new factory, use the method:

```
IlvDXFReaderConfigurator.setGraphicFactory(IlvDXFGraphicFactory)
```

The following DXF entities are read:

- `3DFACE`
- ARC
- CIRCLE
- DIMENSION
- LINE
- POLYLINE
- LWPOLYLINE
- TEXT
- MTEXT
- POINT
- SOLID
- TRACE

Limitations

Only the 2D information of the DXF file is read; 3D information is ignored.

The reader recognizes the most popular attributes, but does not process all the attributes of the entities. Some attributes are not rendered in the same way as Autodesk® AutoCAD.

Graphers

Describes how to use graphers, which are higher-level classes, to create graphic programs that include and present large numbers of dynamic graphic objects.

In this section

The grapher
Describes the grapher, a high-level IBM® ILOG® JViews functionality.

Managing nodes and links
Introduces nodes and links and explains how to manage the nodes and links in a grapher.

Contact points
Describes the default contact points on a node for attaching links and how to define link connectors and pins to specify different contact points.

Class diagram for graphers
Describes the relationships between grapher-related classes with a class diagram.

Grapher interactor class
Describes how to use the grapher interactor class.

Creating a new class of link
Explains how to create a new class of link with an example.

Link shapes and crossing
Explains the support provided for link shape policies and link crossings.

© Copyright IBM Corp. 1987, 2009

The grapher

Based on the manager, the grapher is a natural extension of the manager concepts. A grapher is composed of nodes and links.

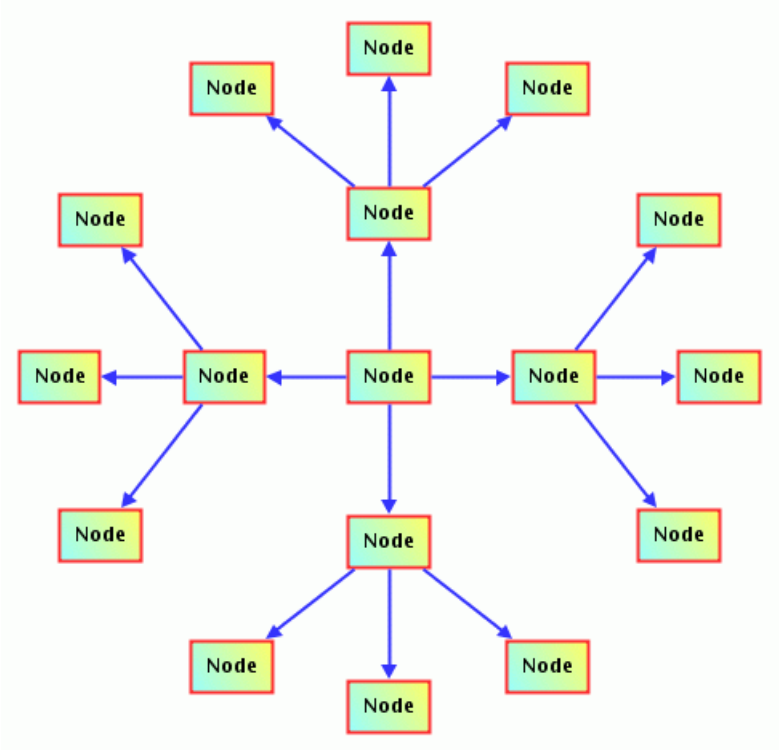
A grapher is an instance of the `IlvGrapher` class, a subclass of the `IlvManager` class. The grapher library offers enhanced performance to create programs, including a large quantity of dynamic interconnected information, such as network management and file management programs.

You have an extensive set of objects in the grapher library for creating the following:

- **Nodes:** The visual reference points in a hierarchy of information. A node is a graphic object, an instance of a subclass of the `IlvGraphic` class.
- **Links**: The visual representation of connections between nodes. Links are also graphic objects, instances of the `IlvLinkImage` class or its subclasses.

Graphic objects in a manager can be visible or invisible. For details, see *Visibility*.

The following example shows a grapher that connects `IlvZoomableLabel` graphic objects (nodes). The lines in blue are node connections (links).



Grapher connecting graphic objects of IlvReliefLabel

Managing nodes and links

Introduces nodes and links and explains how to manage the nodes and links in a grapher.

In this section

Nodes
Describes what nodes are.

Links
Describes what links are.

Predefined link classes
Describes the predefined link classes.

Managing link visibility
Describes how to manage the visibility of links.

Showing and hiding grapher branches
Describes how to manage the visibility of grapher branches.

Nodes

Nodes are simply graphic objects, presented in a grapher. Any graphic object can be used as a node in a grapher.

To add a node to a grapher, use one of the following methods:

```
void addNode(IlvGraphic obj, boolean redraw)
```

```
void addNode(IlvGraphic obj, int layer, boolean redraw)
```

These methods add the object to the specified layer of a grapher and add additional information to the object so that it becomes a node. Graphic objects that have already been added to the grapher using the `addObject` method can be made into nodes of the grapher using the method:

```
void makeNode(IlvGraphic obj)
```

Links

Links are graphic objects used to interconnect nodes in a grapher.

All links are instances of the class `IlvLinkImage` (or subclasses). The constructor of the class `IlvLinkImage` has two graphic objects as parameters, so, when creating a link, you always have to give the origin and destination of the link. Here is the constructor of `IlvLinkImage`:

```
IlvLinkImage(IlvGraphic from, IlvGraphic to, boolean oriented)
```

The `oriented` parameter specifies whether or not an arrowhead is to be drawn at one end of the link. Once a link is created, you can add it to the grapher using one of the following methods:

```
void addLink(IlvLinkImage obj, boolean redraw)
```

```
void addLink(IlvLinkImage obj, int layer, boolean redraw)
```

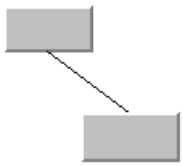
The following code creates a grapher with two nodes and a link.

```
IlvGrapher grapher = new IlvGrapher();
IlvGraphic node1 = new IlvLabel(new IlvPoint(0,0), "node 1");
grapher.addNode(node1, false);
IlvGraphic node2 = new IlvLabel(new IlvPoint(100, 0), "node 2");
grapher.addNode(node2, false);
IlvLinkImage link = new IlvLinkImage(node1, node2, true);
grapher.addLink(link, false);
```

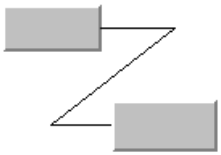
Predefined link classes

The library provides a set of predefined links:

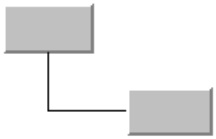
- `IlvLinkImage` - a direct link between two nodes.



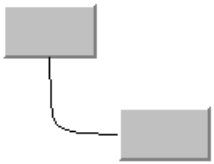
- `IlvPolylineLinkImage` - a link defined by a polyline.



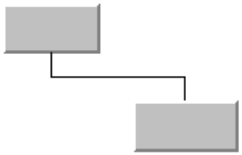
- `IlvOneLinkImage` - a link defined by two lines forming a right angle.



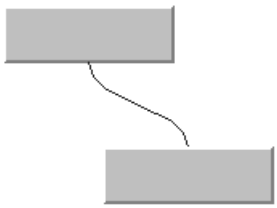
- `IlvOneSplineLinkImage` - a link that shows a spline.



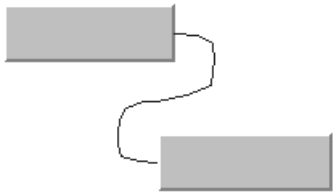
- `IlvDoubleLinkImage` - a link defined by three lines forming two right angles.



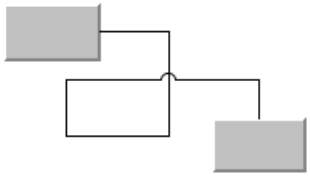
- `IlvDoubleSplineLinkImage` - a link defined by two spline angles.



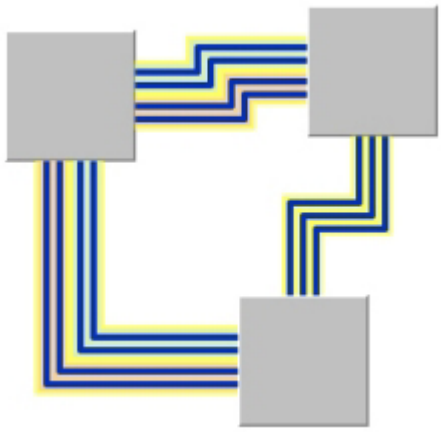
- `IlvSplineLinkImage` - a free-form spline capable of creating mono- or multicurve links.



- `IlvEnhancedPolylineLinkImage` - a link defined by a polyline that supports link shape policies to keep the link in orthogonal shape or to display link crossings or to organize the multi links and self links in bundles. See *Shape policies* for details.



- `IlvLinkBundle` - a link that displays a set of individual links between two nodes in a fixed order and with a specified distance between them.



Managing link visibility

The visibility of links can be controlled independently of the visibility of nodes. This allows you to set a link visible even though its origin and destination nodes are invisible.

It is possible to couple the visibility of a link to the visibility of its end nodes. In this case, the visibility of the link can no longer be controlled independently. A link becomes automatically invisible if its origin node or its destination node becomes invisible. To enable this behavior, you must install the IlvLinkVisibilityHandler as manager listener on the grapher.

```
IlvGrapher grapher = new IlvGrapher();
grapher.addManagerContentChangedListener(new IlvLinkVisibilityHandler());
```

When the link visibility handler is installed, it is possible to select which links are managed by the handler and which links are not managed.

The visibility of a managed link is derived from the visibility of its end nodes. The visibility of an unmanaged link is independent of the visibility of its end nodes and can be controlled by `setVisible`. By default, all links are managed. *Setting a Link as Unmanaged to Control Its Visibility* shows how to set a link as unmanaged to control its visibility.

Setting a Link as Unmanaged to Control Its Visibility

```
// install a link visibility handler. All links are managed.
grapher.addManagerContentChangedListener(new IlvLinkVisibilityHandler());
// mark one link as unmanaged
IlvLinkImage link = ...
IlvLinkVisibilityHandler.setManaged(link, false);
// the visibility of the unmanaged link can be controlled independently
grapher.setVisible(link, false, redraw);
```

Note: In nested graphers, it is sufficient to install the link visibility handler on the top-level grapher only as tree content-changed listener to manage the visibility of all links in all subgraphers. See Content-change events in nested managers.

Showing and hiding grapher branches

You can switch on or off the visibility of the nodes and links that compose a branch of a grapher. Use the method:

```
IlvGrapher.setVisibleBranch(IlvGraphic node, boolean visible,
                            boolean origin)
```

If the argument `origin` is `true`, the method will show or hide the branch that has the node as its origin; that is, all nodes and links reachable by a traversal from the origins to the destinations. The visibility of the node from which the traversal starts is never changed.

If the argument `origin` is `false`, the method will show or hide the branch that has the node as its destination; that is, all nodes and links reachable by a traversal from the destinations to the origins. The visibility of the node from which the traversal starts is never changed.

In addition, the following method allows you to specify how many levels away from the start node that nodes and links should start to be shown or hidden:

```
IlvGrapher.setVisibleBranch(IlvGraphic node, int level,
                            boolean visible, boolean origin)
```

If the level is 0, this method is equivalent to the first method without the level argument.

If the level is 1, the visibility is kept unchanged for the links incident to the starting node and for the nodes adjacent to these links.

If the level is 2, the visibility is kept unchanged for the links incident to the starting node, the nodes adjacent to these links, the links incident to these later nodes, the nodes adjacent to these later links, and so on.

The change of visibility is performed using the method `setVisible(boolean)`.

For an example, see ***<installdir>*** **`/jviews-framework86/codefragments/show-hide-branch/index.html`**.

Contact points

Describes the default contact points on a node for attaching links and how to define link connectors and pins to specify different contact points.

In this section

Default contact points
Describes the default contact points on a node for attaching links.

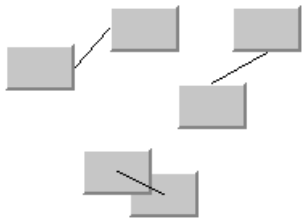
Using link connectors
Describes how to define link connectors.

Using pins
Explains the role of pins and how to use them, with an example.

Other link connectors
Describes other link connectors that do not use pins.

Default contact points

When a link is created between two nodes, it is attached to the default contact point of each node. Each node has five default contact points, one at the center of each side of its bounding rectangle and one at the center of the bounding rectangle. The contact point actually used depends on the location and size of the origin and destination node. The following are examples of connections:



Examples of node connections

Using link connectors

The grapher provides a way to specify the contact points you need on a graphic object. This is done by using *link connectors*, which are subclasses of the class `IlvLinkConnector`,

The class `IlvLinkConnector`is dedicated to the computation of the connection points of links. Subclasses of this abstract class can be used to obtain contact points other than the default ones. An `IlvLinkConnector` is associated with a node. The implementation of the method `getConnectionPoint(ilog.views.IlvLinkImage, boolean, ilog.views.IlvTransformer)` decides where the connection point of a link (provided as an argument) should be located.

An instance of `IlvLinkConnector` can be specified for each node of a grapher. To do this, simply create it using the constructor `IlvLinkConnector(ilog.views.IlvGraphic)` or use the method `attach(ilog.views.IlvGraphic, boolean)`. Notice that the same instance of link connector cannot be shared by several nodes.

A link connector specified for a node controls the connection points of all the links incident to this node. If you need the connection points of the incident links not to all be computed in the same way (that is, by the same link connector), you can specify a link connector individually for each extremity of each link. To do this, simply create it using the constructor

```
IlvLinkConnector(ilog.views.IlvLinkImage, boolean)
```

or use the method

```
attach(IlvLinkImage, boolean, boolean) .
```

Notice that the same link connector can be shared by several links incident to the same node.

To get the instance of link connector actually used to compute the contact point of a given link, use the static method `Get(ilog.views.IlvLinkImage, boolean)`.

Using pins

Each link is attached to what is called a *pin*. Each pin describes the position of a contact point on a node.

The class `IlvPinLinkConnector` , which is a subclass of the class `IlvLinkConnector`,, manages the link connections to a node. An instance of the class `IlvPinLinkConnector` may be installed on a graphic object. This instance holds a set of pins.

When you create an `IlvPinLinkConnector` instance, it is empty and does not contain any pins. You must provide a set of pins describing the position of the contact points that you need. The pins are defined by the class `IlvGrapherPin`. This class is an abstract class because its `getPosition` method is an abstract method. For this reason, you must first create a subclass of the `IlvGrapherPin` class. Specifying an implementation to the `getPosition (ilog.views.IlvTransformer)` method enables you to indicate the position of the *grapher pin*. The signature of this method follows:

```
IlvPoint getPosition(IlvTransformer t)
```

The position of the pin depends on the transformer used to draw the node. This transformer is passed to the `getPosition` method. To compute the position of the pin, you may need to know the position of the node. For its position, use:

```
IlvGraphic getNode()
```

You may also decide to allow or inhibit the connection of a certain type of link to this pin. To do so, you overwrite the `allow` method of your pin, which is called when you create a link with an interactor:

```
boolean allow(Object oClass, Object dClass, Object linkOrClass, boolean origin)
```

The interactor is authorized to highlight only the specified pin based on the result of this method.

Once the pin classes are created, you nned to add the pins to the previously created instance of `IlvPinLinkConnector` using the following method:

```
void addPin(IlvGrapherPin pin)
```

Example: Defining your connection points

This example defines two classes of pins:

- The class `InPin` that allows links to go to the node.
- The class `OutPin` that allows links to come from the node.

  The pins of class `InPin` are placed on the left border of the object and the pins of type `OutPin` on the right border.

The classes are:

```
final class InPin extends IlvGrapherPin
{
  static final int numberOfPins = 5;
  int index;

  public InPin(IlvPinLinkConnector connector, int index)
  {
    super(connector);
    this.index = index;
  }

  protected boolean allow(Object orig, Object dest,
                          Object linkOrClass,
                          boolean origin)
  {
    return !origin;
  }

  public IlvPoint getPosition(IlvTransformer t)
  {
    IlvRect bbox = getNode().boundingBox(null);
    IlvPoint p = new IlvPoint(bbox.x,
                              bbox.y+(bbox.height/(numberOfPins+1)*
                              (index+1));
    if (t != null) t.apply(p);
    return p;
  }
}
```

In this example, five instances of `InPin` will be created. Each pin has an index giving its position on the node. The `getPosition` method returns the position of the pin on the left side of the node according to its index. The `allow` method returns `true` only for links going to this pin (parameter `origin` is `false`). The `OutPin` class is very similar:

```
final class OutPin extends IlvGrapherPin
{
  static final int numberOfPins = 5;
  int index;

  public OutPin(IlvPinLinkConnector connector, int index)
  {
    super(connector);
    this.index = index;
  }

  protected boolean allow(Object orig, Object dest,
                          Object linkOrClass, boolean origin)
  {
    return origin;
  }
```

```
  public IlvPoint getPosition(IlvTransformer t)
  {
    IlvRect bbox = getNode().boundingBox(null);
    IlvPoint p = new IlvPoint(bbox.x+ bbox.width,
                              bbox.y+(bbox.height/
                                      (numberOfPins+1))*(index+1));
    if (t != null) t.apply(p);
    return p;
   }
}
```

The pins are located on the right side and only allow links leaving the node.

If `node` is a graphic object, the method that adds the pins to the node is:

```
grapher.addNode(node, 1, false);
IlvPinLinkConnector lc = new IlvPinLinkConnector(node);
for (int i = 0; i < 5; i++) {
  new InPin(lc, i);
  new OutPin(lc, i);
}
```

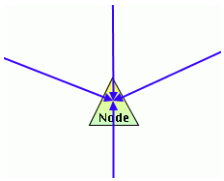
If you want to connect a link to a particular pin, use the method `connectLink` of the class `IlvPinLinkConnector`:

```
public void connectLink(IlvLinkImage link, IlvGrapherPin pin, boolean origin)
```

Other link connectors

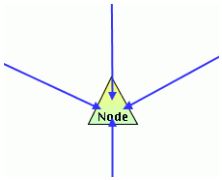
Other link connectors are available as follows:

- `IlvCenterLinkConnector` to connect the link to the center of the node.



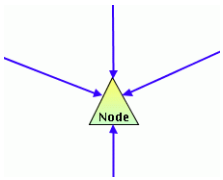
A CenterLinkConnector

- `IlvFreeLinkConnector` to position the link relatively to the node. Any point can be used as the connection point. The connection points are preserved with respect to the bounding box when the node is translated, or when it grows or shrinks.



A FreeLinkConnector

- `IlvClippingLinkConnector` to clip the link at the node border. Like the free link connector, this connector attaches the link to any point inside the node and preserves this attachment point relative to the bounding box of the node. If the node moves, grows, or shrinks, the attachment point moves proportionally. However, unlike the free link connector, the link segment towards the attachment point is clipped at the border of the node by using the method `getIntersectionWithOutline`. This is useful for arrowheads when the shape of the node is nonrectangular.

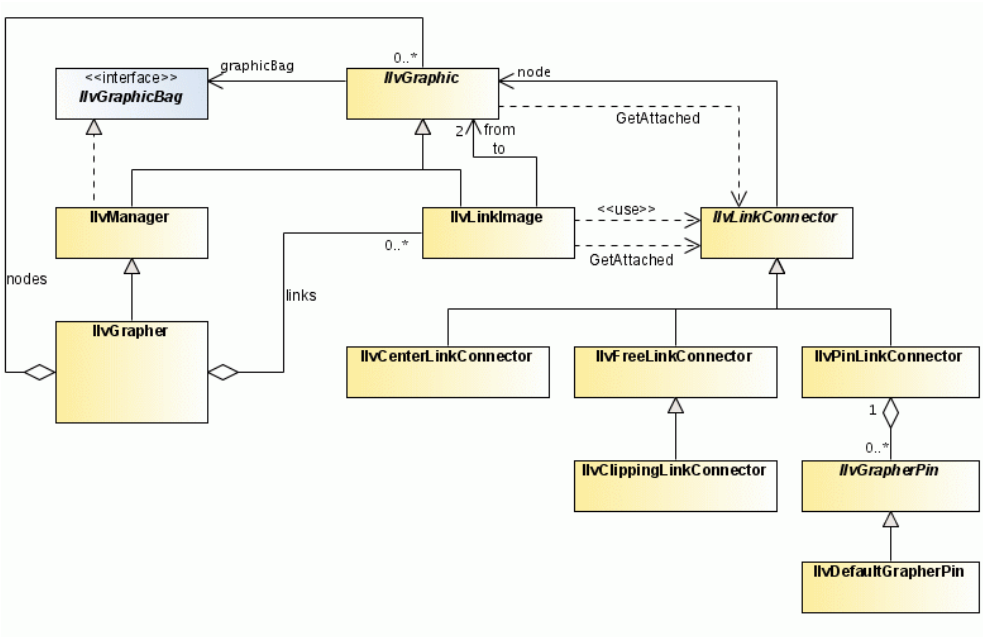


A ClippingLinkConnector

Class diagram for graphers

The following UML class diagram summarizes the relationships between `IlvGrapher`, `IlvGraphic`, `IlvLinkImage`, and `IlvLinkConnector`.

The grapher contains nodes made from `IlvGraphic` objects and links made from `IlvLinkImage` objects. Each link has a "from" node and a “to” node. The link uses the `IlvLinkConnector` class to calculate the contact points. Various link connectors with different behavior are predefined. For example, `IlvPinLinkConnector` is a link connector that specifies a number of `IlvGrapherPin` objects called pins, that can be at the node as contact points.



The Classes Related to IlvGrapher

Grapher interactor class

The library provides several view interactors to create links with the mouse.

The `IlvMakeLinkInteractor` class is an interactor that allows a link of type `IlvLinkImage` to be created by selecting the origin and destination node.

This interactor can be customized so that it creates your own type of link. The link is created by the method `makePolyPoint(ilog.views.IlvPoint[])`. This method uses the `getFrom()` and `getTo()` methods to determine the selected graphic objects:

```
protected IlvGraphic makePolyPoint(IlvPoint[] points)
{
  return new IlvLinkImage(getFrom(), getTo(), isOriented());
}
```

If you override this method, you must also override the method `getLinkClass()` that returns the class of objects created by this interactor.

The class `IlvMakePolyLinkInteractor` is a subclass of the `IlvMakeLinkInteractor` class that allows you to create a link of class `IlvPolylineLinkImage`.

Creating a new class of link

The IlvPolylineLinkImage class

The following example shows the beginning of the class for the new link.

```
public class IlvPolylineLinkImage extends IlvLinkImage
{
  private IlvPoint points[] = null;

  public IlvPolylineLinkImage(IlvGraphic from, IlvGraphic to,
                              boolean oriented, IlvPoint[] points)
  {
    super(from, to, oriented);
    init(points);
  ...
}
}
```

The new link is defined by a polyline, whose starting and ending positions are fixed and are based on the starting and ending positions of the nodes. This link is a subclass of the `IlvLinkImage` class.

The private field `points` will contain all the intermediate points of the link. The origin and destination points are not contained in this array.

The constructor calls the corresponding constructor of `IlvLinkImage` and initializes the object. The `init` method then fills the `points` field as follows.

```
private void init(IlvPoint[] pts)
{
  if (pts == null)
    return;
  int i;
  points = new IlvPoint[pts.length];
  for (i = 0; i < pts.length ; i++)
    points[i] = new IlvPoint(pts[i].x, pts[i].y);
}
```

There is also a copy constructor and a copy method that allow you to copy the object.

```
public IlvPolylineLinkImage(IlvPolylineLinkImage source)
{
  super(source);
  init(source.points);
}

public IlvGraphic copy()
{
```

```
  return new IlvPolylineLinkImage(this);
}
```

getLinkPoints

The `getLinkPoints` method returns the points defining the shape of the link. This method is used by `IlvLinkImage` to draw the object and to define the bounding rectangle of the object. In the class `IlvLinkImage`, this method only returns the origin and destination points of the link. For the new polyline object, the `getLinkPoints` method adds the intermediate points of the link as follows.

```
public IlvPoint[] getLinkPoints(IlvTransformer t)
{
  int nbpoints = getPointsCardinal();
  IlvPoint[] pts = new IlvPoint[nbpoints];
  if (nbpoints > 2)
    for (int i = 1 ; i < nbpoints-1; i++) {
      pts[i] = new IlvPoint(points[i-1]);
      if (t != null) t.apply(pts[i]);
    }
  pts[0] = new IlvPoint();
  pts[nbpoints-1] = new IlvPoint();
  getConnectionPoints(pts[0], pts[nbpoints-1], t);
  return pts;
}
```

The `getConnectionPoints` method is used to get the intermediate points. The `getConnectionPoints` method computes the origin and destination point of the link. These points may depend on the connection pins on the origin or destination object.

getPointCardinal, getPointAt

These methods, originating from the interface `IlvPolyPointsInterface`, are defined like this:

```
public int getPointsCardinal()
{
  if (points == null)
    return 2;
  else
    return points.length +2;
}

public IlvPoint getPointAt(int index, IlvTransformer t)
{
  if (index == 0 || index == getPointsCardinal()-1)
  {
   IlvPoint[] pts = new IlvPoint[2];
   pts[0] = new IlvPoint();
   pts[1] = new IlvPoint();
```

```
   getConnectionPoints(pts[0], pts[1], t);
   return pts[(index == 0) ? 0 : 1];
  }
  else
  {
    IlvPoint p = new IlvPoint(points[index-1]);
    if (t != null)
      t.apply(p);
    return p;
  }
}
```

allowsPointInsertion, allowsPointRemoval

The `allowPointAddition` and `allowPointRemoval` methods are overridden to return `true` to allow the editing interactor associated with links (that is, `IlvLinkImageEditInteractor`), to add and remove points.:

```
public boolean allowsPointInsertion()
{
  return true;
}

public boolean allowsPointRemoval()
{   return points != null && points.length >= 1;
}
```

Since these methods return `true`, the `insertPoint` and `removePoint` methods will be called from the interactor. They are defined as follows.

```
public void insertPoint(int index, float x, float y,
                        IlvTransformer t)
{
  if (points == null && index == 1) {
    points = new IlvPoint[1];
    points[0] = new IlvPoint(x,y);
  }
  else if (index == 0)
    throw new IllegalArgumentException("bad index");
  else if (index >= getPointsCardinal())
    throw new IllegalArgumentException("bad index");
  else index --;
  if (index >= 0 && index <= points.length){
    IlvPoint[] oldp = points;
    points = new IlvPoint[oldp.length+1];
    System.arraycopy(oldp, index, points, index + 1,
                     oldp.length - index);
    points[index] = new IlvPoint(x,y);
    if (index > 0)
      System.arraycopy(oldp, 0, points, 0, index);
  }
```

```
  else throw new IllegalArgumentException("bad index");
}

public void removePoint(int index, IlvTransformer t)
{
  if (index ==0) return;
  if (index == getPointsCardinal()-1)
    return;
  index --;
  if (points != null && index >= 0 && index < points.length)
  {
    IlvPoint[] oldp = points;
    points = new IlvPoint[oldp.length-1];
    if (index > 0)
      System.arraycopy(oldp, 0, points, 0, index);
      int j = oldp.length - index - 1;
      if (j > 0)
      System.arraycopy(oldp, index + 1, points, index, j);
  }
  else throw new IllegalArgumentException("bad index");
}
```

applyTransform

The method `applyTransform` is called when the bounding box is to be modified (when the object is moved or enlarged, for example). The transformation is applied to the intermediate points:

```
public void applyTransform(IlvTransformer t)
{
  if (getPointsCardinal() > 2 && points != null)
    for (int i = 0 ; i < points.length; i++) {
     if (t != null) t.apply(points[i]);
    }
}
```

Input/Output

Input/output methods are needed to allow users to save and read the intermediate points.

The method `write` is defined as follows:

```
public void write(IlvOutputStream stream) throws IOException
{
  super.write(stream);
  stream.write("points", points);
}
```

The corresponding `IlvInputStream` constructor is as follows.

```
public IlvPolylineLinkImage(IlvInputStream stream) throws
                                                IlvReadFileException
{
  super(stream);
  IlvPoint[] points = stream.readPointArray("points");
  init(points);
}
```

Link shapes and crossing

Shape policies

Link shape policies control the shape of an individual link. They are a way to ensure that a link keeps a specific shape. See the class `IlvLinkShapePolicy` for details.

Links may stay orthogonal or they may cross and you can set the aspect of link crossings.

IBM® ILOG® JViews Framework proposes the following predefined link shape policies:

- The class `IlvOrthogonalLinkShapePolicy` keeps links orthogonal.
- The class `IlvCrossingLinkShapePolicy` calculates how crossings are displayed.
- The class `IlvBundleLinkShapePolicy` organizes multilinks and self links in bundles.

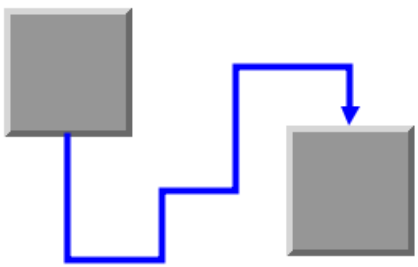
These link shape policies are for internal use by the class `IlvEnhancedPolylineLinkImage`,. This class is a subclass of `IlvPolylineLinkImage` designed to provide support for link shape policies. By hiding the implementation and computation details, this class makes link shape policies easier for you to use.

The orthogonal mode and the crossing mode of the class `IlvEnhancedPolylineLinkImage` are implemented by link shape policies.

By default, orthogonal mode and crossing mode are switched off. In this case, the class `IlvEnhancedPolylineLinkImage` behaves exactly like `IlvPolylineLinkImage`.

Orthogonal links

If you set the method `setOrthogonal(boolean)setOrthogonal`. of the class `IlvEnhancedPolylineLinkImage` to `true`, the link stays orthogonal, even if you try to reshape it interactively. You can add or remove bends, or move bends interactively, the link always reorganizes the adjacent bends so that the link keeps an orthogonal shape, as illustrated in *Orthogonal link between two nodes*.



Orthogonal link between two nodes

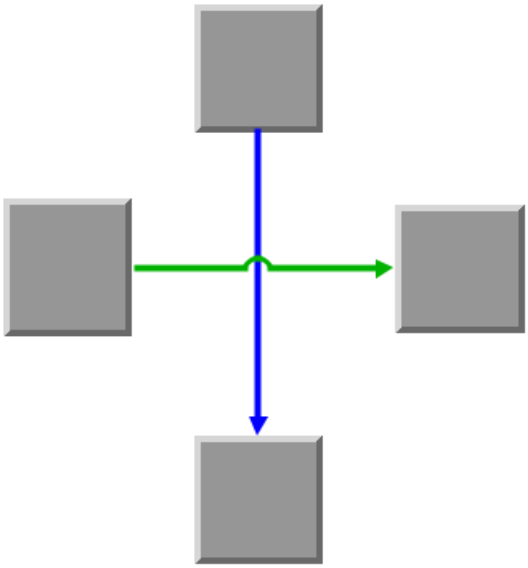
The accessor `isOrthogonal` returns whether the link is in orthogonal mode.

When the orthogonal mode is switched on, the class `IlvOrthogonalLinkShapePolicy` automatically controls the shape policy.

Crossing modes

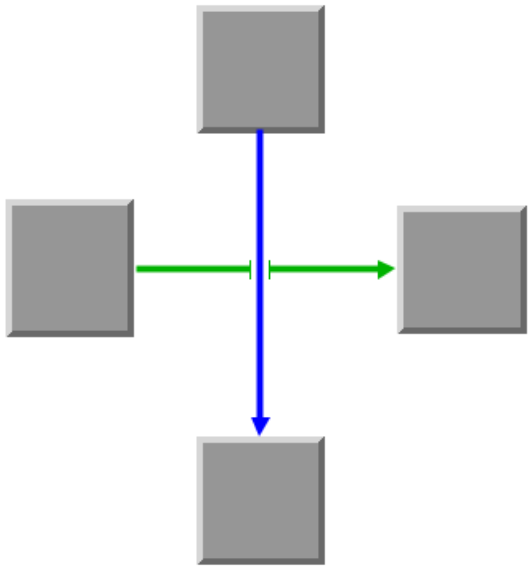
The method `setCrossingMode(int)` of the class `IlvEnhancedPolylineLinkImage` enables you to select the aspect of the image of the link at the place where two links cross. To do so, set the `mode` parameter to one of the following options:

- `NO_CROSSINGS`: crossings are not displayed in any particular way (default).
- `TUNNEL_CROSSINGS`: crossings are displayed with a tunnel shape, as illustrated in *Tunnel-shaped Crossing Mode*.



Tunnel-shaped Crossing Mode

- `BRIDGE_CROSSINGS`: crossings are displayed with a bridge shape, as illustrated in *Bridge-shaped Crossing Mode*.



Bridge-shaped Crossing Mode

The accessor `getCrossingMode` returns the current crossing mode.

When a crossing mode other than `NO_CROSSINGS` is set, the class `IlvCrossingLinkShapePolicy` automatically controls the shape policy.

Composite Graphics

Introduces the Composite Graphics feature and explains how to create a composite graphic.

In this section

Introducing composite graphics
Describes what composite graphics are and the support provided for them.

Creating a composite graphic
Provides a tutorial which explains how to create a simple composite graphic.

© Copyright IBM Corp. 1987, 2009

Introducing composite graphics

With composite graphics, JViews Framework enables you to associate graphic objects in a single object.

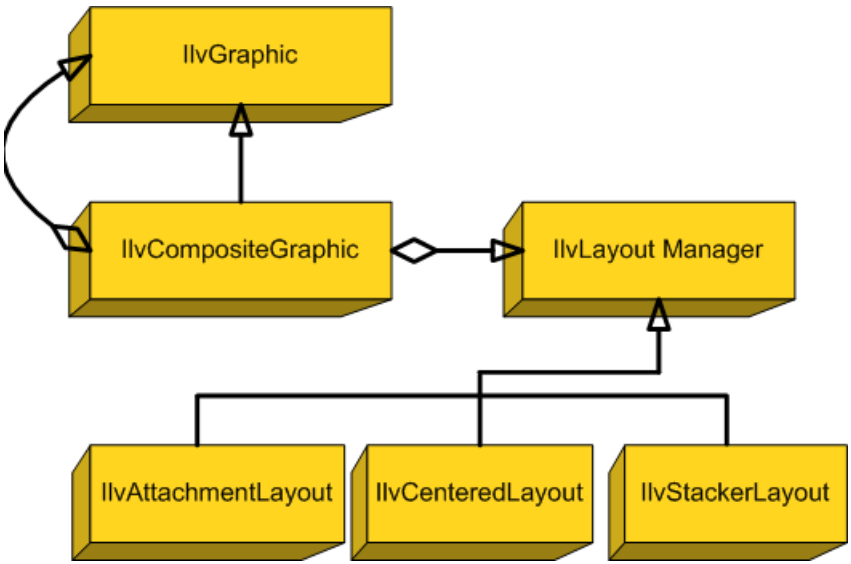
Composite graphics are a set of classes that help you combine `IlvGraphic` objects to build up more complex objects based on simple graphics. Unlike graphic sets (see *Groups*, composite graphics have layout and attachment capabilities.

A composite graphic object is made of child graphics and a layout manager object. There are three possible classes of layout manager: `IlvAttachmentLayout`, `IlvCenteredLayout`, and `IlvStackerLayout`. Each child graphic is a graphic object that can be either a basic `IlvGraphic` or an `IlvCompositeGraphic`.

The API for composite graphics is thus based on the following classes:

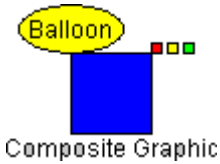
- `IlvCompositeGraphic`: this class holds a set of `IlvGraphic` subclasses. This class composes `IlvGraphic` objects.
- `IlvLayoutManager` subclasses: `IlvAttachmentLayout`, `IlvCenteredLayout` and `IlvStackerLayout`. These classes are responsible for positioning the graphics (see *Creating a composite graphic* for details).

Class Hierarchy for Composite Graphics shows the relationships between these classes.



Class Hierarchy for Composite Graphics

If you follow the steps given in *Creating a composite graphic*, you will obtain the composite graphic illustrated in *Creating a Composite Graphic: Final Result*.



Creating a Composite Graphic: Final Result

To compose this graphic, you just need to reuse existing graphics provided by the JViews Framework. Furthermore, composite graphics can be recursive, that is, a composite graphic may be made up of other composite graphics. In *Creating a Composite Graphic: Final Result*, for example, the balloon object and the three rectangles are themselves composite graphics.

Creating a composite graphic

The corresponding code is supplied in the `codefragments/composite/src/Composite.java` source file of the installed product. For details, see ***<installdir>*** **`/jviews-framework86/codefragments/composite/index.html`**.

This tutorial contains the following stages:

- *Stage 1 - Starting the composite graphic*
- *Stage 2 - Creating an attachment layout*
- *Stage 3 - Creating the first child graphic*
- *Stage 4 - Attaching a child graphic*
- *Stage 5 - Using a stacker layout*
- *Stage 6 - Using a centered layout*

Stage 1 - Starting the composite graphic

To create a composite graphic object:

1. Import the composite graphics package:

```
import ilog.views.graphic.composite.* ;
```

2. Create a composite graphic object.

```
IlvCompositeGraphic composite = new IlvCompositeGraphic();
```

Stage 2 - Creating an attachment layout

The attachment layout enables you to attach the child graphics to the first one by choosing symbolic points of their bounding boxes.

To create the attachment layout:

1. Import the layout package:

```
import ilog.views.graphic.composite.layout.*;
```

2. Create the attachment layout.

```
IlvAttachmentLayout layout = new IlvAttachmentLayout();
composite.setLayout(layout);
```

Stage 3 - Creating the first child graphic

The first child graphic of the `IlvCompositeGraphic` object will be the reference for positioning other child graphics. In this example, the first child graphic is an `IlvRectangle` object.

To create the first child object:

- Create a rectangle object:

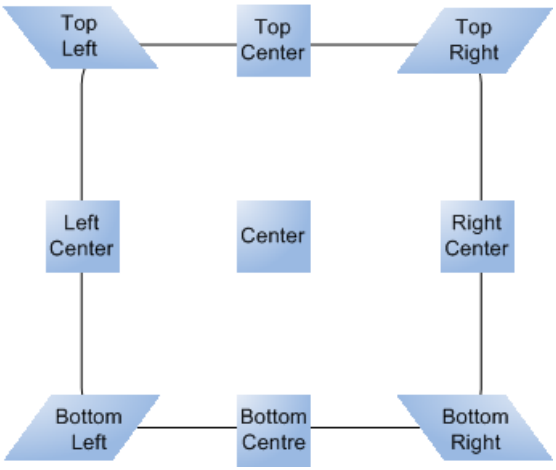
```
IlvRectangle rectangle = new IlvRectangle(new IlvRect(0,0,40,40),true,true);
composite.setChildren(0,rectangle)
```

The first child is at position 0. The attachment layout uses this first child to attach the other children (sibling graphics) of the composite graphic.

Stage 4 - Attaching a child graphic

A label can be attached as a child graphic.

There are nine different attachment locations available, as illustrated in the following figure.



Attachment Locations

To label your composite graphic:

1. Attach an additional child graphic to the first one.

```
IlvText text = new IlvText();
text.setLabel("Composite Graphic");
composite.setChildren(1,text);
```

2. Place the label below the rectangle:

```
composite.setConstraints(1,
```

```
new IlvAttachmentConstraint(IlvAttachmentLocation.TopCenter,
IlvAttachmentLocation.BottomCenter));
```

Here, the top center of the text (an `IlvText`) is anchored to the bottom center of the first child graphic which is, in this example, an `IlvRectangle` object.

If you run your example now, you can see the following composite, in which the “Composite Graphic” label is attached to the blue rectangle with horizontal symmetry:

Composite graphics with attachments

Stage 5 - Using a stacker layout

As well as using the Attachment Layout to position two objects with respect to each other within the composite graphic object, you can use the Stacker Layout to align objects.

To use the Stacker Layout to align three icons:

1. Create one more composite graphic named `rectangles` to hold the three small rectangles. Its role will be to align these rectangles horizontally.

```
IlvCompositeGraphic rectangles = new IlvCompositeGraphic();
```

2. Create a stacker layout and pass it to the Composite Graphic object.

```
IlvStackerLayout stacker = new IlvStackerLayout(SwingConstants.
RIGHT,SwingConstants.BOTTOM,3);
rectangles.setLayout(stacker);
```

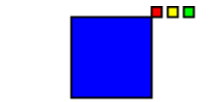
3. Create the three rectangles and set them as child graphics of the `rectangles` composite graphic.

```
IlvRectangle r1 = new IlvRectangle(new IlvRect(0,0,5,5),true,true);
r1.setBackground(Color.red);
rectangles.setChildren(0,r1);

IlvRectangle r2 = new IlvRectangle(new IlvRect(0,0,5,5),true,true);
r2.setBackground(Color.yellow);
rectangles.setChildren(1,r2);

IlvRectangle r3 = new IlvRectangle(new IlvRect(0,0,5,5),true,true);
r3.setBackground(Color.green);
rectangles.setChildren(2,r3);
```

4. Make the `rectangles` composite graphic as a child graphic of the main composite graphic built in the previous stages.

```
composite.setChildren(2,rectangles);
composite.setConstraints(2,new IlvAttachmentConstraint
(IlvAttachmentLocation.BottomLeft,IlvAttachmentLocation.TopRight));
```

At the end of this stage, you should obtain the following result:



Stacker Layout

Stage 6 - Using a centered layout

Unlike center attachment (see *Attachment Locations*), Centered Layout has the specific feature of handling two graphics, the *outer graphic* and the *inner graphic*. More precisely, in Centered Layout, the composite is a container that lays out two children: the first, at index position 0, is the outer graphic, the second one, at index position 1, is the inner graphic. The outer graphic will be resized by the composite in such way that the inner graphic remains at the center of the outer graphic.

This example creates a Balloon made of a yellow ellipse containing the text “Balloon” and adds it as a new child of the composite graphic built in the previous stages. Balloon is a new composite graphic that uses Centered Layout to position the text and the ellipse with respect to each other. In the following steps, you create the composite graphic as a new object (the Balloon), the outer graphic (the ellipse), the inner graphic (the text), then you attach the Balloon to the main composite graphic.

To create the balloon and center the text:

1. Create the Balloon composite graphic with Centered Layout:

```
IlvCompositeGraphic balloon = new IlvCompositeGraphic();
IlvCenteredLayout centered = new IlvCenteredLayout(new Insets(5,5,5,5));
balloon.setLayout(centered);
```

2. Create the outer graphic as an instance of `IlvEllipse`:

   The outer graphic is always the first child, at position 0, of the composite graphic with Centered Layout. This child is resized by the composite to the size of the inner graphic extended by the inset given to the Centered Layout ((5,5,5,5)) at step 1.

```
IlvEllipse ellipse = new IlvEllipse();
ellipse.setFillOn(true);
ellipse.setBackground(Color.yellow);
balloon.setChildren(0,ellipse);
```

3. Create the inner graphic as an instance of `IlvText`:

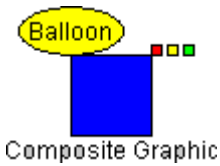
   The inner graphic is always the second child, at position 1, of the composite graphic with Centered Layout.

```
IlvText balloonText = new IlvText(new IlvPoint(0,0),"Balloon");
balloon.setChildren(1,balloonText);
```

4. Attach the Balloon to the main composite graphic using `IlvAttachmentConstraint`:

```
composite.setChildren(3,balloon);
composite.setConstraints(3,new IlvAttachmentConstraint
(IlvAttachmentLocation.BottomCenter,IlvAttachmentLocation.TopLeft));
```

At the end of this stage, you should obtain the composite graphic illustrated in *Centered Layout*:



Centered Layout

Index

A

abortReDraws method
  IlvManager class **118**
accelerators
  handling events **84**
addInteractorListener method
  IlvManagerView class **156**
addLayer method
  IlvManagerLayer class **101**
addLink method
  IlvGrapher class **173**
addManagerChangedListener method
  IlvManager class **104**
  IlvManagerView class **91, 121**
addManagerContentChangedListener method
  IlvManager class **120**
addManagerSelectionListener method
  IlvManager class **129**
addManagerViewsListener method
  IlvManager class **91**
addNode method
  IlvGrapher class **172**
addObject method
  IlvManager class **102, 112**
addPin method
  IlvPinLinkConnector class **182**
addTransformer method
  IlvManagerView class **92**
addTransformerListener method
  IlvManagerView class **93**
addVisibilityFilter method
  IlvManagerLayer class **102**
ADJUSTMENT_END event **120**
allow method
  IlvGrapherPin class **182**
allowsPointInsertion method
  IlvLinkImage class **189**
applet
  initialization **17**
applyToObject method
  IlvManager class **114**
applyTransform method
  IlvGraphic class **66**
  IlvLinkImage class **191**
arcs **59**
arrows **59**
attachment layout **200**
auxiliary view
  repaint request **98**

B

Background manager view property **37**
Beans
  editing properties **36**
  for GUI components **31, 32**
  for main data structure **31**
  for predefined interactors **31**
  installing in an IDE **30**
bounding box **50**
boundingBox method
  IlvGraphic class **50, 66, 69**
  IlvRect class **145**
branch of grapher
  visibility of nodes and links **178**
BRIDGE_CROSSINGS link crossing mode **194**
buffers
  double buffering **95**
  triple buffering **105**

C

centered layout **202**
child, in composite graphics
  attaching **201**
  creating **200**

© Copyright IBM Corp. 1987, 2009

classes, hierarchy **49**
composite graphics **63, 198**
attachment layout **200**
centered layout **202**
creating, step-by-step procedure **200**
introducing **198**
stacker layout **201**
`connectLink` method
`IlvPinLinkConnector` class **182**
`contains` method
`IlvGraphic` class **74**
controlling visibility of nodes and links
independently **177**
together **177**
coupling visibility of nodes and links **177**

D

data structure
Beans **31**
creating **37**
`IlvManager` basic class **11**
`deleteAll` method
`IlvManager` class **112**
`deSelectAll` method
`IlvManager` class **125**
`DIRECT_DRAW`
view repaint mode **98**
double buffering **95**
`draw` method
AWT `Graphics` class **68**
`IlvGraphic` class **50, 66**
`drawGhost` method
`IlvManagerViewInteractor` class **145, 156**
drawing order **108**
drawing, optimizing **118**

E

editing **61**
editing object properties **117**
ellipses **59**
events
handling **84**
managing **141**
examples
creating the `ShadowEllipse` class **67**
defining connection points between nodes **182**
extending the `IlvObjectInteractor` class **145**
implementing the `DragRectangleInteractor` class **156**
importing the JViews library **16**
using double-buffering **95**
zooming a view **92**

F

filled arcs **59**
filled ellipses **59**
`fitTransformer` method
`IlvManagerView` class **92**
functions
user-defined **116**

G

`Get` method
`IlvObjectInteractor` class **144**
`getCardinal` method
`IlvManager` class **112**
`getConnectionPoints` method
`IlvLinkImage` class **189**
`getCrossingMode` method
`IlvEnhancedPolylineLinkImage` class **194**
`getDefaultInteractor` method
`IlvSelection` class **159**
`getGrid` method
`IlvGrid` class **96**
`getInsertionLayer` method
`IlvManager` class **112, 127**
`getInteractor` method
`IlvManagerViewInteractor` class **152**
`getLayer` method
`IlvManager` class **102**
`getLayerCount` method
`IlvManagerLayer` class **101**
`getLinkClass` method
`IlvMakeLinkConnector` class **187**
`getLinkPoints` method
`IlvLinkImage` class **189**
`getNode` method
`IlvGrapherPin` class **182**
`getObject` method
`IlvSelection` class **125**
`getObjectInteractor` method
`IlvManager` class **144**
`getObjects` method
`IlvManager` class **112**
`getPointAt` method
`IlvPolyPointsInterface` **189**
`getPointCardinal`method
`IlvPolyPointsInterface` **189**
`getPosition` method
`IlvGrapherPin` class **182**
`getProperty` method
`IlvGraphic` class **55**
`getSelectedObjects` method
`IlvManager` class **125**
`getSelection` method
`IlvManager` class **125**
graphers

visibility of nodes and links in a branch **178**
graphic bags **58**
graphic objects
groups **63**
IlvGraphic basic class **10**
introduction **47**
predefined, hierarchy **49**
user properties **55**
graphic paths **65**
grid
manager view **96**

H

handles **124**
handling events **84**
hasProperty method
IlvGraphic class **55**
hierarchy of graphic object classes **49**
holes **65**

I

icons **64**
IlvArc class **152**
IlvArrowLine class **152**
IlvArrowPolyline class **152**
IlvAttachmentConstraint class **203**
IlvAttachmentLayout class **198**
IlvBundleLinkShapePolicy class **193**
IlvCenteredLayout class **198**
IlvCenterLinkInteractor class **185**
IlvCircularScale class **64**
IlvClippingLinkInteractor class **185**
IlvCompositeGraphic class **63, 198**
IlvCrossinglLinkShapePolicy class **193**
IlvDoubleLinkImage class **174**
IlvDragRectangleInteractor class **32, 152, 156**
IlvDrawSelection class **124**
IlvEditLabelInteractor class **32, 152**
IlvEllipse class **59, 152, 203**
IlvEnhancedPolylineLinkImage class **193**
getCrossingMode method **194**
isOrthogonal method **193**
setCrossingMode method **194**
setOrthogonal method **193**
IlvFreeLinkInteractor class **185**
IlvGeneralPath class **64**
IlvGrapher class **31, 170, 178**
addLink method **173**
addNode method **172**
makeNode method **172**
IlvGrapherPin class **182**
allow method **182**
getNode method **182**
getPosition method **182**
IlvGraphic class **10, 48, 52, 178, 198**
applyTransform method **66**
boundingBox method **50, 66, 69**
draw method **66**
getProperty method **55**
hasProperty method **55**
makeSelection method **127**
move method **52**
moveResize method **52**
removeProperty method **55**
replaceProperty method **55**
resize method **52**
rotate method **52**
scale method **52**
setObjectInteractor method **144**
setProperty method **55**
translate method **52**
write method **75**
IlvGraphicBag class **58**
IlvGraphicBag interface **166**
IlvGraphicEnumeration class **112**
IlvGraphicPath class **65**
IlvGraphicSet class **63, 166**
IlvGrid class **31**
getGrid method **96**
setGrid method **96**
snapToGrid method **96**
IlvHoverHighligtingImageOperation class **133**
IlvIcon class **25, 64**
IlvInputStream class **56**
IlvJComponentGraphic class **65**
IlvJManagerViewControlBar class **33**
IlvJScrollManagerView class **31, 35, 94**
IlvLabelInterface class **152**
IlvLayerVisibilityFilter class **102**
IlvLine class **152**
IlvLinkImage class **152, 170, 173, 187**
allowsPointInsertion method **189**
allowsPointRemoval method **189**
applyTransform method **191**
getConnectionPoints method **189**
getLinkPoints method **189**
write method **191**
IlvLinkShapePolicy class **193**
IlvLinkVisibilityHandler class **177**
IlvMagnifyInteractor class **32, 152**
IlvMakeArrowLineInteractor class **152**
IlvMakeArrowPolylineInteractor class **152**
IlvMakeLineInteractor class **152**
IlvMakeLinkInteractor class **32, 187**
getLinkClass method **187**
makePolyPoint method **187**
IlvMakePolygonInteractor class **152**
IlvMakePolylineInteractor class **152**

IlvMakePolyLinkInteractor class **152, 187**
IlvMakePolyPointsInteractor class **32, 152**
IlvMakeRectangleInteractor class **32, 152**
IlvMakeSplineInteractor class **152**
IlvManager class **11, 17, 31, 37, 166, 170**
abortReDraws method **118**
addManagerContentChangedListener method **120**
addManagerLayerListener method **104**
addManagerSelectionListener method **129**
addManagerViewsListener method **91**
addObject method **102, 112**
applyToObject method **52, 114**
deleteAll method **112**
deSelectAll method **125**
DXF file
reading into IlvManager **166**
getCardinal method **112**
getInsertionLayer method **112, 127**
getLayer method **102**
getObjectInteractor method **144**
getObjects method **112**
getSelectedObjects method **125**
getSelection method **125**
initReDraws method **118**
invalidateRegion method **118**
isEditable method **117**
isInvalidating method **118**
isManaged method **112**
isMovable method **117**
isSelectable method **103, 117**
isSelected method **125**
isVisible method **102**
mapInside method **116**
mapIntersects method **116**
moveObject method **114**
read method **164**
reDrawViews method **118**
removeObject method **112**
reshapeObject method **114**
selectAll method **125**
setEditable method **117**
setHoverHighlightingImageOperation method **133**
setHoverHighlightingMode method **132**
setInsertionLayer method **112**
setLayer method **102**
setMovable method **117**
setSelectable method **103, 117**
setSelected method **125**
setVisible method **102**
write method **164**
IlvManagerLayer class **101, 102**
addLayer method **101**
addVisibilityFilter method **102**
getLayerCount method **101**
removeLayer method **101**
IlvManagerLayerAdapter class **104**
IlvManagerMagViewInteractor class **32**
IlvManagerView class **11, 31, 35, 36, 87, 92, 98**
addInteractorListener method **156**
addManagerChangedListener method **91, 121**
addTransformer method **92**
addTransformerListener method **93**
fitTransformer method **92**
invalidateTripleBuffer method **105**
isKeepingAspectRatio method **92**
removeInteractorListener method **156**
removeManagerChangedListener method **91**
removeTransformerListener method **93**
setInteractor method **20, 22**
setKeepingAspectRatio method **92**
setTransformer method **92**
setTripleBufferedLayerCount method **105**
Translate method **92**
zoom method **92**
IlvManagerViewControlBar class **33**
IlvManagerViewInteractor class **19, 32, 143, 152**
drawGhost method **145, 156**
getInteractor method **152**
popInteractor method **152**
pushInteractor method **152**
setInteractor method **152**
IlvManagerViewPanel class **31, 95**
isDoubleBuffering method **95**
setDoubleBuffering method **95**
IlvMarker class **63**
IlvMoveRectangleInteractor class **152**
IlvObjectInteractor class **143**
extending **145**
Get method **144**
processEvent method **145**
IlvObjectInteractorContext class **145**
IlvOneLinkImage class **174**
IlvOneSplineLinkImage class **174**
IlvOrthogonalLinkShapePolicy class **193**
IlvOutputStream class **56**
IlvPanInteractor class **32, 152**
IlvPinLinkConnector class
addPin method **182**
connectLink method **182**
IlvPolygon class **152**
IlvPolyline class **152**
IlvPolylineLinkImage class **152, 174, 187, 188**
IlvPolyPoints class **60**
IlvPolyPointsInterface
getPointAt method **189**
getPointCardinal method **189**

IlvPopupMenuManager class **162**
registerView method **162**
IlvReadFileException class **164**
IlvRect class
boundingBox method **145**
IlvRectangle class **152**
IlvRectangularScale class **64**
IlvReliefRectangle class **60, 152**
IlvReshapeSelection class **159**
IlvRotateInteractor class **32, 152**
IlvScrollManagerView class **31, 94**
IlvSelectInteractor class **19, 21, 32, 117, 126, 152, 159**
isDragAllowed method **159**
isEditionAllowed method **159**
isMoveAllowed method **93, 159**
isMultipleSelectionMode method **159**
setDragAllowed method **159**
setEditionAllowed method **159**
setMoveAllowed method **159**
setMultipleSelectionMode method **159**
IlvSelection class **124**
getDefaultInteractor method **159**
getObject method **125**
IlvSelectionFactory class **127**
IlvSpline class **124, 152**
IlvSplineSelection class **124**
IlvStackerLayout class **198**
IlvText class **203**
IlvToolTipManager class **162**
registerView method
IlvToolTipManager class **162**
IlvTransformer class **50**
IlvUnZoomViewInteractor class **21, 152**
IlvZoomViewInteractor class **21, 32, 152**
importing
library and packages **16**
indexing layers **101**
initReDraws method
IlvManager class **118**
inner graphic, in composite graphics/centered layout **203**
input/output operations **56, 75, 85**
interactorChanged method
InteractorListener interface **23, 156**
InteractorChangedEvent class **156**
InteractorListener interface **21**
interactorChanged method **23, 156**
interactors
Beans **31**
grapher **187**
handling events **84**
listener **156**
predefined **152**
selection **126**
view **152**
invalidateRegion method
IlvManager class **118**
invalidateTripleBuffer method
IlvManagerView class **105**
isDoubleBuffering method
IlvManagerViewPanel class **95**
isDragAllowed method
IlvSelectInteractor class **159**
isEditable method
IlvManager class **117**
isEditionAllowed method
IlvSelectInteractor class **159**
isInvalidating method
IlvManager class **118**
isKeepingAspectRatio method
IlvManagerView class **92**
isManaged method
IlvManager class **112**
isMovable method
IlvManager class **117**
isMoveAllowed method
IlvSelectInteractor class **93, 159**
isMultipleSelectionMode method
IlvSelectInteractor class **159**
isOrthogonal method
IlvEnhancedPolylineLinkImage class **193**
isSelectable method
IlvManager class **103, 117**
isSelected method
IlvManager class **125**
isVisible method
IlvManager class **102**

J

jviews-framework-all.jar file **30**

K

KeepAspectRatio manager view property **37**

L

labels **61**
layerInserted method
ManagerLayerListener interface **104**
layerMoved method
ManagerLayerListener interface **104**
layerRemoved method
ManagerLayerListener interface **104**
layers
adding objects to **102**
indexing **101**
managing **99**
selectability of objects **103**
setting up **101**

triple buffering **105**
visibility of objects **102**
layout
aligning **201**
attachment **200**
centering **202**
manager object **198**
library, importing **16**
lines **59**
links
creating **188**
crossing modes **193**
definition **170**
managed **177**
managing **170**
orthogonal **193**
predefined **174**
shape policies **193**
unmanaged **177**
visibility in branch of a grapher **178**
visibility of managed **177**
visibility of unmanaged **177**
listener
some manager layer events **104**
listeners, for changes to
a transformer **93**
an interactor **156**
the content of the manager **120**
the selections in a manager **129**
listening to some, but not all events
manager layer listener **104**

M

main view
repaint request **98**
`makeNode` method
`IlvGrapher` class **172**
`makePolyPoint` method
`IlvMakeLinkConnector` class **187**
`makeSelection` method
`IlvGraphic` class **127**
manager view
auxiliary view **98**
editing properties **36**
main view **98**
repaint delay **98**
`managerChanged` method
`ManagerChangedListener` interface **91**
`ManagerChangedEvent` class **91**
`ManagerChangedListener` interface **91**
`managerChanged` method **91**
`ManagerLayerListener` interface **104**
`layerInserted` method **104**
`layerMoved` method **104**
`layerRemoved` method **104**
managers
binding views **89**
introducing **81**
layers **83**
view grid **96**
`ManagerSelectionListener` interface **129**
`ManagerViewsChangedListener` interface **91**
`viewchanged` method **91**
managing
events **141**
graphic objects **111**
layers **99**
links **170**
nodes **170**
selected objects **125**
`mapInside` method
`IlvManager` class **116**
`mapIntersects` method
`IlvManager` class **116**
markers **63**
`moveObject` method
`IlvManager` class **114**
`MoveObjectInteractor` class **144, 145**

N

`NO_CROSSINGS` link crossing mode **194**
nodes
contact points **180**
definition **170**
managing **170**
visibility in branch of a grapher **178**

O

object model for a graphics application **10**
`OBJECT_ADDED` type of change **120**
`OBJECT_BBOX_CHANGED` type of change **120**
`OBJECT_LAYER_CHANGED` type of change **120**
`OBJECT_REMOVED` type of change **120**
`OBJECT_VISIBILITY_CHANGED` type of change **120**
objects
adding to and removing from manager **112**
adding to layers **102**
bounding box **50**
creating **66, 67, 127**
drawing **50**
editing properties **117**
layers **102**
managing **111**
modifying the geometric properties **114**
moving **52, 114**
nonzoomable **51**
reading **56, 75**
resizing **52**
rotating **52**
saving **56, 75**
scaling **52**

selectability **103**
selection objects **123**
translating **52**
visibility **102**
zoomable **51**
orthogonal links **193**
outer graphic, in composite graphics/centered layout **203**

P

packages, importing **16**
paths **64, 65**
polygons **60, 65**
polylines **60, 65**
pop-up menu manager
`IlvPopupMenuManager` class **162**
`popInteractor` method
`IlvManagerViewInteractor` class **152**
popup menu manager
`IlvPopupMenuManager` class **162**
popup menus **162**
predefined graphic objects **59**
hierarchy **49**
predefined interactors **31**
predefined links **174**
predefined view interactors **152**
`processEvent` method
`IlvObjectInteractor` class **145**
properties
editing **117**
geometric **50, 114**
named **77**
selecting **117**
user **55**
`pushInteractor` method
`IlvManagerViewInteractor` class **152**

Q

quadtree **108**

R

`read` method
`IlvManager` class **164**
`read` superclass constructor **75**
reading
a file **17**
an object in a JViews formatted file **75**
manager contents **164**
rectangles **60**
`reDrawViews` method
`IlvManager` class **118**
refresh delay **98**
`registerMenu` method
`IlvPopupMenuManager` class **162**
`removeInteractorListener` method
`IlvManagerView` class **156**
`removeLayer` method
`IlvManagerLayer` class **101**
`removeManagerChangedListener` method
`IlvManagerView` class **91**
`removeObject` method
`IlvManager` class **112**
`removeProperty` method
`IlvGraphic` class **55**
`removeTransformerListener` method
`IlvManagerView` class **93**
repaint delay
manager view **98**
repaint requests
delay **98**
skipping **98**
`replaceProperty` method
`IlvGraphic` class **55**
`reshapeObject` method
`IlvManager` class **114**
resizing method
`IlvGraphic` class **52**

S

saving
an object in a JViews formatted file **75**
manager contents to file **164**
scales **64**
scrolled manager view **94**
selectability of objects **103**
`selectAll` method
`IlvManager` class **125**
selecting object properties **117**
selection objects **123**
selections
interactor **126**
`setCrossingMode` method
`IlvEnhancedPolylineLinkImage` class **194**
`setDoubleBuffering` method
`IlvManagerViewPanel` class **95**
`setDragAllowed` method
`IlvSelectInteractor` class **159**
`setEditable` method
`IlvManager` class **117**
`setEditionAllowed` method
`IlvSelectInteractor` class **159**
`setGrid` method
`IlvGrid` class **96**
`setHoverHighlightingImageOperation` method
`IlvManager` class **133**
`setHoverHighlightingMode` method
`IlvManager` class **132**
`setInsertionLayer` method
`IlvManager` class **112**
`setInteractor` method
`IlvManagerView` class **20, 22**

IlvManagerViewInteractor class **152**
setKeepingAspectRatio method
IlvManagerView class **92**
setLayer method
IlvManager class **102**
setMovable method
IlvManager class **117**
setMoveAllowed method
IlvSelectInteractor class **159**
setMultipleSelectionMode method
IlvSelectInteractor class **159**
setObjectInteractor method
IlvGraphic class **144**
setOrthogonal method
IlvEnhancedPolylineLinkImage class **193**
setProperty method
IlvGraphic class **55**
setSelectable method
IlvManager class **103, 117**
setSelected method
IlvManager class **125**
setTransformer method
IlvManagerView class **92**
setTripleBufferedLayerCount method
IlvManagerView class **105**
setVisible method
IlvManager class **102**
shadows **61**
shape policies for links **193**
shapes **60**
skipping repaint requests
IlvManagerView **98**
snapToGrid method
IlvGrid class **96**
stacker layout **201**
Swing
tooltip manager **162**

T

text editing **61**
texts **61**
THREADED_REDRAW
view repaint mode **98**
tooltip manager
IlvToolTipManager class **162**
Swing **162**
tooltips **162**
transformations
to define the displayed area of the manager **92**
transformers
listener **93**
Translate method
IlvManagerView class **92**
triple buffering **105**
TUNNEL_CROSSINGS link crossing mode **194**

U

user-defined functions **116**

V

view grid **96**
view interactors
predefined **152**
view repaint mode
DIRECT_DRAW **98**
THREADED_REDRAW **98**
viewchanged method
ManagerViewsChangedListener interface **91**
viewport
IlvManagerView basic class **11**
views
binding to a manager **89**
creating **17**
of a manager **82**
predefined view interactors **152**
scrolled manager view **94**
transformations **92**
zooming **92**
visibility
of nodes and links in branch of a grapher **178**
visibility of objects **102**

W

write method
IlvGraphic class **56, 75**
IlvLinkImage class **191**
IlvManager class **164**
wysiwyg editing **62**

Z

Z-order **108**
zoom method
IlvManagerView class **92**
zoomable objects **51**
zooming a view **92**