IBM ILOG JViews Framework V8.6

IBM ILOGJViews Framework Advanced Features

Table of contents

Nested managers and nested graphers

Describes nested managers and graphers. Nesting allows you to add a manager to another manager or a grapher to another grapher.

In this section

Submanagers
Explains the nesting feature for managers and graphers and further features of nested graphs.

Using nested managers
Describes nested managers and explains how to create them and work with them.

Adding a manager frame
Explains how to add and customize a frame around the objects in a nested manager.

Expanding and collapsing
Explains how to expand and collapse a nested manager and how to manage the collapsed representation and the expand/collapse events.

Using nested graphers
Describes nested graphers and explains how to create them and work with them.

Selection in nested managers
Explains how to select objects in a nested manager and in a hierarchy of nested managers.

Selection interactor in nested managers
Explains how to use the selection interactor.

Content-change events in nested managers
Explains how to be notified of content-change events.

Hierarchy events in nested managers
Describes the two types of hierarchy event and explains how to be notified of them and how they propagate.

Interactors for nested managers and graphers
Describes the graphic object interactors that can be used with nested managers and graphers.

Class diagram for nested managers
Describes the relationships between the classes for nested managers with a class diagram.

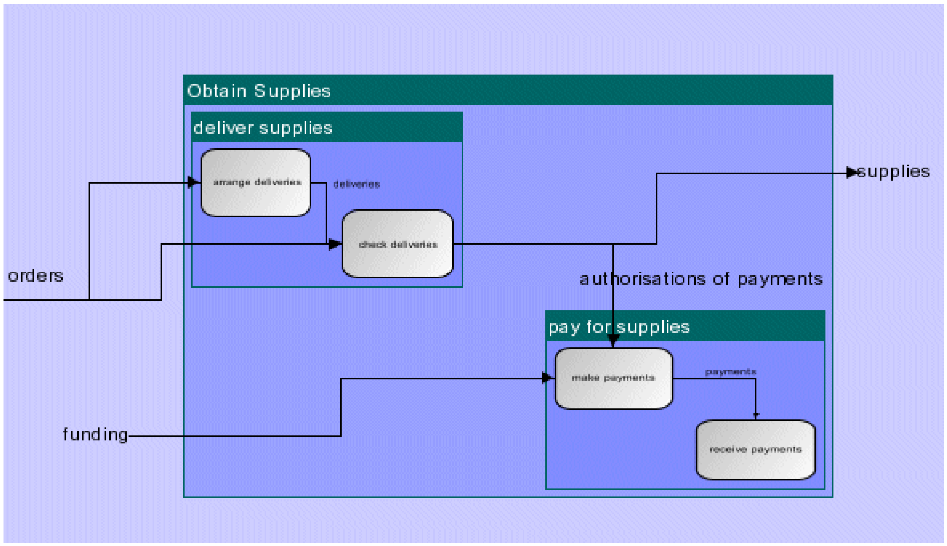
Submanagers

The manager and the grapher are graphic objects that can be embedded inside another manager or grapher. This nesting feature of IBM® ILOG® JViews allows you to create applications that display a graph inside another graph.

The manager and the grapher are the main classes that can contain graphic objects for displaying and manipulating in several views.

For an introduction to the manager (`IlvManager`) and its subclass the grapher (`IlvGrapher`), see Managers and Graphers of The Essential JViews Framework.

The following figure shows an example of nested graphs.



Nested graphs

In this figure, the object entitled “Obtain Supplies” is a grapher (instance of `IlvGrapher`) that itself contains two other graphers, “deliver supplies” and “pay for supplies.” The figure shows that a manager (and a grapher) can be surrounded by a frame; in this example the three (blue) frames each display the name of the manager as a title. This type of decoration as well as the background of the submanager can be completely customized. Another feature shown in this example is the fact that links between nodes can cross subgraph boundaries. Such links are called *intergraph* links.

Each grapher or manager embedded inside another manager or grapher can also be displayed in several views, just like top-level managers.

A manager or grapher embedded inside another manager has two different representations: an expanded state where all the objects contained in the submanager are visible, and a collapsed state where the manager is drawn with a collapsed representation that can also be customized.

Using nested managers

Describes nested managers and explains how to create them and work with them.

In this section

Adding a nested manager

The `IlvManager` class inherits from the `IlvGraphic` class; as a consequence, a manager and a grapher can be added to another manager or grapher just like any other graphic object. To add a manager to a manager, you use the `addObject(ilog.views.IlvGraphic, boolean)` method of the `IlvManager` class.

Example: Adding a nested manager

The following code shows a simple nested manager.

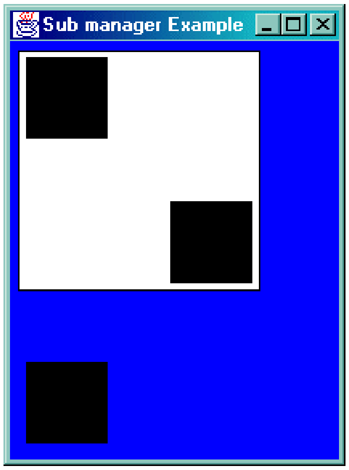
```
import ilog.views.*;
import ilog.views.graphic.*;
import javax.swing.*;
import java.awt.*;

public class SubManagerExample
{
  public static void main(String[] args) {
    IlvGraphic obj;
    IlvManager toplevel = new IlvManager();
    IlvManager subManager = new IlvManager();

    obj = new IlvRectangle(new IlvRect(10,10,50,50), false, true);
    subManager.addObject(obj, false);
    obj = new IlvRectangle(new IlvRect(100,100,50,50), false, true);
    subManager.addObject(obj, false);

    toplevel.addObject(subManager, false);

    obj = new IlvRectangle(new IlvRect(10,200,50,50), false, true);
    toplevel.addObject(obj, false);

    final IlvManagerView view = new IlvManagerView(toplevel);
    view.setBackground(Color.blue);
    SwingUtilities.invokeLater(
      new Runnable() {
        public void run() {
          JFrame frame = new JFrame("Sub manager Example");
          frame.getContentPane().add(view);
          frame.setSize(200,200);
          frame.setVisible(true);
        }
      });
}
```

This simple example creates two `IlvManager` objects, the top-level manager (variable `toplevel`) that will be displayed in the view and the submanager (variable `subManager`). The submanager is added to the top-level manager by the line:

```
toplevel.addObject(subManager, false);
```

Two rectangles are also added to the submanager. Another rectangle is added at the top level.

The following figure shows the resulting application.



Submanager example

The white area is the submanager containing two rectangles.

Note: Adding a manager to a manager can be done to an infinite level. The library will just make sure that you do not create cycles in the hierarchy of nested managers.

Optimizing the addition of nested managers

It is possible to add submanagers to a manager in two different ways:

This first approach requires multiple recalculations of the bounds of the parent manager while filling the submanager. Adding an object to the submanager might not only change the bounds of the submanager but also the bounds of all its ancestor managers.

1. Add the submanager to its parent manager.
2. Fill the submanager

The second approach avoids changing the bounds of all its ancestor managers and is always faster. Generally, it is faster to add a subobject to a manager after all the customizations of the subobject are completed.

1. Fill the submanager.
2. Add it to its parent manager.

However, it is not always possible to control the order in which objects are added to managers. In this case, insertion adjusting sessions help:

```
manager.setInsertionAdjusting(true);
try {
    ... add nested managers and nodes recursively in any order ..
} finally {
    manager.setInsertionAdjusting(false);
}
```

This ensures that the performance is optimal even when submanagers and their contents are added in the wrong order. It ensures that the bounding boxes of the manager and its all submanagers are only recalculated when setting the insertion adjusting flag to `false`. Events related to the change of the bounds of the managers are also delayed until this time.

The insertion adjusting flag needs only to be set at the top level manager, and it will be automatically propagated to all existing and newly created submanagers. The insertion adjusting flag affects only the performance when objects are added. It has no effect when objects are removed.

Traversing nested managers

Once a manager has been added in a manager, it is no different from any other graphic object. Nevertheless, the manager contains a set of methods that allow you to have quick access to the managers added in a manager and to the manager hierarchy in general.

You can obtain an enumeration of the managers present in a submanager using the method `getManagers()` and a count of how many managers are present with the method `getManagersCount()`.

The same methods can be applied to a particular layer: `getManagers(int)` and `getManagersCount(int)`.

Note that these methods will return the submanagers at the first level of the hierarchy only. Using these methods is much more efficient than traversing the list of all objects present in a manager.

You can also traverse up the hierarchy of managers, using the method `getParent()`.

In the simple example (see *Example: Adding a nested manager*) the following line would return `toplevel`:

```
subManager.getParent()
```

For a manager at the top level, this method returns `null`.

Coordinate system in nested managers

The position and size of a submanager depend on the position and size of the objects that are contained in the submanager. As a consequence, all objects that are contained in the submanager are always displayed, and the size and position of the submanager may change when a graphic object contained in this manager changes position or size.

Note that in the simple example (see *Example: Adding a nested manager*), the position and size of the submanager has not been specified.

As for any graphic object, you can move and resize a submanager using the `moveObject (ilog.views.IlvGraphic, float, float, boolean)` or `reshapeObject(ilog.views. IlvGraphic, ilog.views.IlvRect, boolean)` methods. In the example you could do something like:

```
toplevel.moveObject(subManager, 100,100, true)
```

When a nested manager is resized or moved, the objects contained in the manager do not change position in the submanager coordinate system. The submanager will move because an affine transformation (an `IlvTransformer`) will be specified in the submanager to define a new relative coordinate system for the submanager. In the line for moving the `subManager` to `(100,100)`, the transformation is a simple translation of `(100,100)`.

To obtain the affine transform that defines the coordinate system of the submanager, you can use the method `getTransformer()`.

If the submanager has not been moved or reshaped, this returns the identity transformation.

To know what transformation is used to draw a submanager in a specified view, you can use the following method of the `IlvManager` class: `getDrawingTransformer(ilog.views. IlvManagerView)`.

This method returns the affine transformation used to draw the objects in a manager. If the specified view is a view of the manager, then it simply returns the affine transform of the view; otherwise, the method will compose the transformation of all the parents of the manager and also the transformation of the view to give the result.

Working with graphic objects in nested managers

The API you use to manipulate graphic objects stored in a nested manager is the same as the one you use when working on a top-level manager. For example, you use the `moveObject(ilog.views.IlvGraphic, float, float, boolean)` or `reshapeObject(ilog.views.IlvGraphic, ilog.views.IlvRect, boolean)` method of the `IlvManager` class to move or reshape an object in a nested manager. You also use the `applyToObject(ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)` method of `IlvManager` when modifying a property of a graphic object that changes the bounding box of a graphic object. The only difference for a nested manager is that when a graphic object is stored in a nested manager, changing its size or moving it can change the size of the manager itself, proceeding recursively up the hierarchy of the manager.

The `IlvManager` class provides some convenient methods for working with graphic objects in nested managers; these methods have a `traverse` parameter that when set to `true` means that the method applies also to nested managers in this manager.

You can access all the objects in the hierarchy of managers using the following methods with the `traverse` parameter set to `true`:

♦ To return the total number of objects in the hierarchy use the method:

  ```
  int getCardinal(boolean traverse)
  ```

♦ To return an enumeration of all objects use the method:

  ```
  IlvGraphicEnumeration getObjects(boolean traverse)
  ```

♦ You can locate an object under a certain point using:

  ```
  IlvGraphic getObject(IlvPoint p, IlvManagerView view, boolean traverse)
  ```

♦ Finally, you traverse the hierarchy of objects to apply a function:

  ```
  void map(IlvApplyObject f, Object arg, boolean traverse)
  ```

  This method applies the function `f` to all objects of the hierarchy when the `traverse` parameter is set to `true`.

  ```
  void mapIntersects(IlvApplyObject f, Object arg,IlvRect rect, IlvTransformer

  t, boolean traverse)
  ```

  This method applies the function to all graphic objects that intersects the specified rectangle in the hierarchy of nested managers.

  ```
  void mapInside(IlvApplyObject f, Object arg, IlvRect rect, IlvTransformer
  ```

```
t,
boolean traverse)
```

This method applies the function to all graphic objects that are inside the specified rectangle in the hierarchy of the nested manager.

Some methods allow you to deal with selection and deselection of objects in a hierarchy of nested managers. This is explained in *Selection in nested managers*.

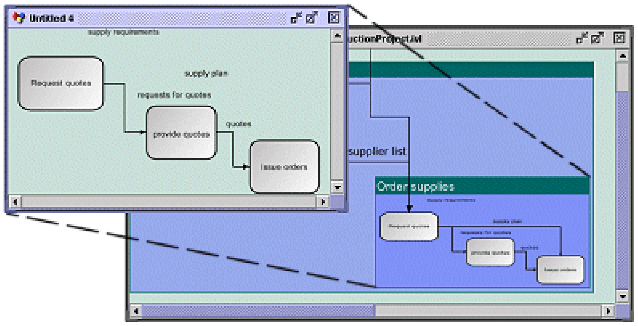
Views on a nested manager

It is possible to view the contents of a nested manager in several views (`IlvManagerView`) just as you can do for a top-level manager.

Any modification of an object in a nested manager will be reflected in all the views in which the object appears, which are:

- ♦ All the views of the submanager that contain the object.
- ♦ All the views of the parent manager, and proceeding recursively.

The association of the view with a submanager is no different from the association of the view for the top-level manager.



Views on nested manager

Adding a manager frame

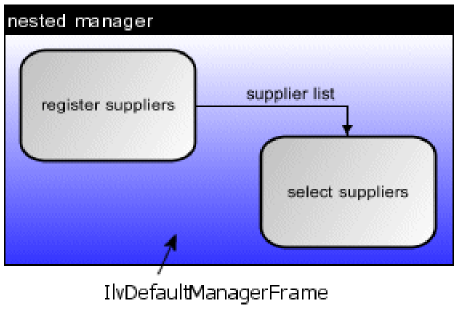
Explains how to add and customize a frame around the objects in a nested manager.

In this section

Defining and drawing a default frame and its margins

When a manager is nested, a frame can be drawn around the objects it contains.

A default frame is provided when you create a nested manager; the default frame is defined by the class `IlvDefaultManagerFrame`.



Default manager frame

Defining the frame

The frame around a nested manager is defined by the interface `IlvManagerFrame`. The `IlvManagerFrame` interface defines the margin that will be added around the manager. It defines how the frame is drawn and how hit testing on the frame is performed.

To specify the frame that must be drawn around the manager, you use the following methods of the `IlvManager` class:

```
void setFrame(IlvManagerFrame frame)
```

```
IlvManagerFrame getFrame()
```

Note that you can remove the frame from the manager using the `setFrame` method with a `null` parameter.

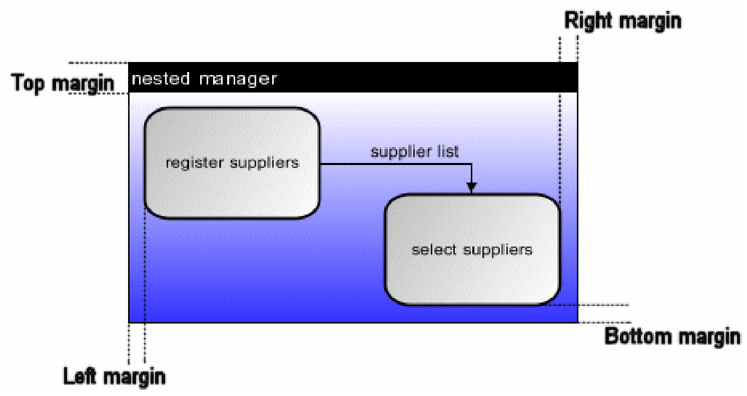
Defining the margins

The following methods of the `IlvManagerFrame` interface define the margins that are added around the manager:

```
float getBottomMargin(IlvManager manager, IlvTransformer t)
```

```
float getLeftMargin(IlvManager manager, IlvTransformer t)
float getRightMargin(IlvManager manager, IlvTransformer t)
```

```
float getTopMargin(IlvManager manager, IlvTransformer t)
```



Manager frame margins

Drawing the default frame

The `IlvManagerFrame` interface provides a method to draw the frame:

```
void draw(IlvManager manager, IlvRect bbox, Graphics g, IlvTransformer t)
```

This method provides the manager with its bounding box in the view coordinate system that already takes the margins into account. The `IlvDefaultManagerFrame` fills the background and makes a border around the manager. This class also puts a title at the top of the manager that corresponds to the name of the manager, as you can see in *Manager frame margins*.

If the frame implementation fills the background of the manager, the frame is an opaque frame and the following method must return `true`:

```
boolean isOpaque(IlvManager manager)
```

The `IlvDefaultManagerFrame` object can be opaque or transparent.

- ♦ If the frame is opaque, graphic objects under the nested manager are hidden. This disallows manipulation of these graphic objects.
- ♦ If the frame is transparent, graphic objects under the nested manager are visible. This allows manipulation of the graphic objects, whether covered by the frame or not.

Using the default frame

Hit testing

To allow hit testing on the frame, the interface provides the `contains` method:

```
boolean contains(IlvManager manager, IlvPoint p, IlvPoint tp, IlvTransformer
t)
```

If the frame is transparent, this method must return `true` for a possible border and not for the background.

Saving the frame to an IVL file

To be able to save the frame to an IVL file, the implementation must be a public class that also implements the `IlvPersistentObject` interface.

Copying a frame

To be able to copy a manager that has a frame, the frame must implement the `copy` method of the interface in such a way that it creates a clone of the frame:

```
IlvManagerFrame.copy()
```

Using the constant mode frame

Instead of the `IlvDefaultManagerFrame`, you can use the `IlvConstantModeManagerFrame`, a manager frame supplied with IBM® ILOG® JViews that handles the resize event automatically when a user reshapes a submanager. `IlvConstantModeManagerFrame` keeps the zoom level of the submanager unchanged by adjusting the frame margins to match the required size.

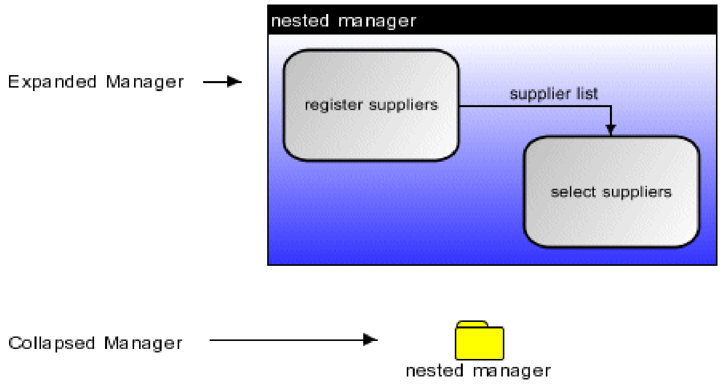
A submanager with the `IlvConstantModeManagerFrame` frame set has the following behavior:

- ♦ Margins are never negative.
- ♦ A minimum margin size can be set.
- ♦ The default margin size is 0.
- ♦ The submanager can only be translated.
- ♦ The frame reacts to the `IlvManager.moveResize()` method but not `IlvManager.applyTransform()`. Call `IlvManager.applyTransform()` to apply a specific transformation only.
- ♦ While reshaping the submanager by dragging a handle, the margin next to the handle is adjusted first. If this margin is reduced to zero, the margin on the other side of the frame is adjusted.
- ♦ When both left and right margins of a `IlvConstantModeManagerFrame` object become zero, the submanager stops the reshape operation.
- ♦ `IlvConstantModeManagerFrame` objects listens for submanager or grapher content changes. When the bounding box of the submanager grows, in order to keep the global bounding box constant the frame first tries to reduce margins.

For an example of how to use a manager frame, see ***<installdir>*** **`/jviews-framework86/codefragments/subgraphs/index.html`**.

Expanding and collapsing

A nested manager can be expanded or collapsed. When collapsed, a nested manager has a different representation, as illustrated in the following figure.



Expanded and collapsed manager

To expand/collapse a manager, the `IlvManager` class provides the following methods:

`setCollapsed(boolean)`

`isCollapsed()`

When a nested manager is collapsed, the contents of the manager and the frame are no longer drawn. Only the new graphic representation is shown.

Defining the collapsed representation

The graphic representation that defines the collapsed manager is a graphic object, an instance of `IlvGraphic`. This allows you to define any kind of collapsed representation for your manager. The default graphic object used to draw a collapsed manager is an instance of the class: `IlvDefaultCollapsedGraphic`, which represents a folder above the name of the manager. To change this default representation, you use the following methods of the class `IlvManager`:

```
void setCollapsedGraphic(IlvGraphic graphic)
```

```
IlvGraphic getCollapsedGraphic()
```

Note that the graphic object used for the collapsed representation cannot be shared by several managers.

When the manager is collapsing, the collapsed graphic will be placed at the center of the area of the manager. When the manager is expanding, it will move so that its center will be placed at the center of the collapsed graphic.

Expand/collapse events

When the manager is expanded or collapsed, it fires an event to notify you of this change. The event is defined by the class `ManagerExpansionEvent`.

To listen to such events, you must create a listener that implements the `ManagerExpansionListener` interface, which defines the two methods:

♦ `managerCollapsed(ilog.views.event.ManagerExpansionEvent)` which is called after the manager is collapsed.

♦ `managerExpanded(ilog.views.event.ManagerExpansionEvent)` which is called before the manager is expanded.

You will then register your listener using the following method of the `IlvManager` class:

```
void addManagerExpansionListener(ManagerExpansionListener listener)
```

Using nested graphers

Describes nested graphers and explains how to create them and work with them.

In this section

Nested graphers

As a subclass of a manager, a grapher (instance of `IlvGrapher`) inherits all the behavior of managers, including nesting.

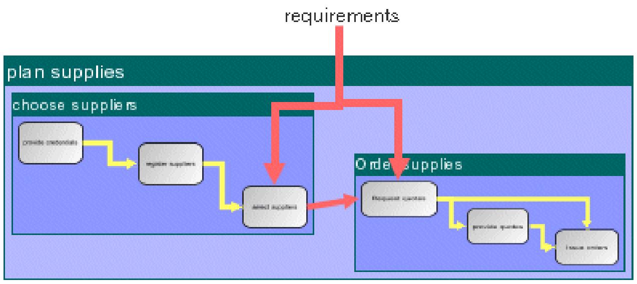
A grapher can be nested in another grapher or in a manager. It can be collapsed or expanded, and a frame can be set on it.

In addition to having the features inherited from the `IlvManager` class, nested graphers allow you to create applications that define graphs containing subgraphs, with links crossing the graph boundaries (intergraph links).

Intergraph links

An intergraph link is a link that crosses grapher boundaries. In other words, an intergraph link is a link whose origin and destination are in different graphers.

Some examples of intergraph links are shown in the following figure.



Intergraph Links and regular links

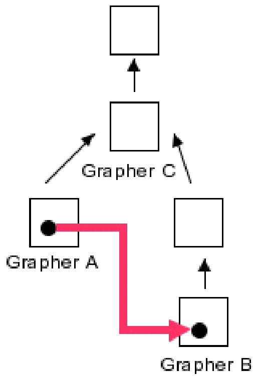
In this picture the red (darker) links are intergraph links and the yellow (light) links are regular links.

Creating and accessing an intergraph link

To be able to create intergraph links, you must have a hierarchy of nested graphers (`IlvGrapher`). The hierarchy should not contain a manager (`IlvManager`); otherwise the library will not let you create an intergraph link.

Creating an intergraph link

Given that an intergraph link has an origin and a destination in different graphers, the question that arises is in which grapher the link is stored. The intergraph link between a node stored in grapher A and a node stored in grapher B must be stored in the first common ancestor of A and B, as shown in the following figure.



Intergraph link in a hierarchy of graphers

In this figure, the red intergraph link that connects an object of A and an object of B must be stored in grapher C, the first common ancestor of A and B.

The `IlvGrapher` class provides a static utility method that allows you to determine the first common grapher:

```
static IlvGrapher getLowestCommonGrapher(IlvGraphic obja, IlvGraphic objb)
```

To create an intergraph link, the code will look like this (assuming that the origin and destination variables have been created and added in different graphers):

```
IlvGraphic origin, destination;
IlvLinkImage link;

...

link = new IlvLinkImage(origin, destination, false);
```

```
IlvGrapher common = IlvGrapher.getLowestCommonGrapher(origin, destination);

common.addLink(link, false);
```

The `IlvGrapher`.addInterGraphLink static utility method allows you to add the link directly to the common parent grapher:

```
static void addInterGraphLink(IlvLinkImage link, boolean redraw)
```

The code shown in the previous example is equivalent to:

```
IlvGraphic origin, destination;
IlvLinkImage link;

...

link = new IlvLinkImage(origin, destination, false);

IlvGrapher.addInterGraphLink(link, false);
```

Accessing an intergraph link

The `IlvGrapher` class also provides methods that let you access intergraph links stored in a grapher in an efficient way:

```
IlvGraphicEnumeration getInterGraphLinks()
```

```
int getInterGraphLinksCount()
```

Since an intergraph link is stored in the same way as other links in the grapher, it is also part of the list of all objects contained in this grapher returned by the `getObjects` method of the class `IlvManager`:

```
IlvGraphicEnumeration getObjects()
```

Nevertheless, calling the `getInterGraphLinks` method is much more efficient than traversing all objects of the grapher.

To distinguish an intergraph link from other objects in the grapher, you can use the following method of the `IlvGrapher` class:

```
boolean isInterGraphLink(IlvGraphic obj)
```

This method returns `true` if the specified graphic object is a link stored in the grapher instance with the origin or the destination stored somewhere else. That is, if the graphic object is an intergraph link.

The `IlvGrapher` class also gives you access to intergraph links that are leaving or entering a grapher. You can access such links using the methods:

```
IlvGraphicEnumeration getExternalInterGraphLinks()
```

```
int getExternalInterGraphLinksCount()
```

The difference between the methods `getInterGraphLinks` and is the following:

♦ The first method, `getInterGraphLinks`, returns the intergraph links stored in a grapher with an origin or destination in *another* grapher

♦ The second method, `getExternalInterGraphLinks`, returns the intergraph links stored in another grapher but with the origin or destination in *this* grapher

If the grapher has no subgraphers, the external intergraph links obtained by `getExternalInterGraphLinks()` are all links that are leaving or entering the grapher.

If the grapher has subgraphers, the intergraph links that leave the grapher have one of the following:

♦ An end node in this grapher

♦ An end node in a subgrapher of this grapher

Calling `getExternalInterGraphLinks()` on the grapher gives only the links with an end node in the grapher, not the links with an end node in a subgrapher of the grapher.

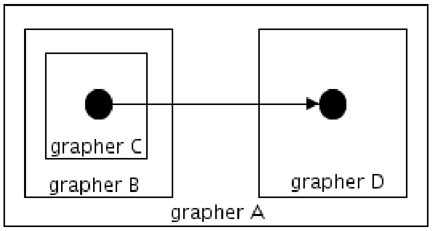
In this case, links that are leaving or entering the nesting hierarchy of the grapher can be obtained by examining all external intergraph links of the subgraphers recursively. To help in this task the following convenience methods are supplied: `getTreeExternalInterGraphLinks` and `getTreeExternalInterGraphLinksCount`.

When a grapher is moved, it is possible that the grapher and all its nested subgraphers appear at a new location on the screen. The grapher displacement causes the shape of the following links to change:

♦ All links that are directly connected to the grapher. That is, have the grapher as origin or destination.

♦ All links that are obtained by `grapher.getTreeExternalInterGraphLinks()`.

The following figure illustrates interlinked nested graphers:



External intergraph link

Grapher A contains two graphers, B and D. Grapher B contains another grapher, C. The intergraph link from an object of C to an object of D is then stored in A.

This link is an intergraph link of A (returned in `getInterGraphLinks()` called on A) and is also an external intergraph link of C and D (returned by `getExternalInterGraphLinks()` called on C or D).

Neither the origin nor destination of the intergraph link ends in grapher B, and thus this link is not returned by calling `getExternalInterGraphLinks` on B. However, the link is returned by calling `getTreeExternalInterGraphLinks()` on B, since its origin node is nested inside B.

Coordinate system of Intergraph Links

Since an intergraph link has its origin and destination in different graphers, it may be difficult to determine the coordinate system in which the bend points of an intergraph link are defined.

The coordinate system of the link and its bend points is always the coordinate system of the grapher to which the link has been added. The only difference compared to regular links is the way an intergraph link computes its end points: the connection points of the link to its origin and destination. The link itself, in fact the base class of the `IlvLinkImage` class, computes its connection points by calling:

```
void getConnectionPoints(IlvPoint src, IlvPoint dst, IlvTransformer t)
```

This method determines whether the link connector (`IlvLinkConnector`) is installed on the destination or origin node, by calling:

```
boolean getLinkConnectorConnectionPoint(boolean origin, IlvPoint p,
IlvTransformer t)
```

The `getConnectionPoints` method stores the result in the `src` and `dst` parameters. If the link is an intergraph link, this method will compute the connection points in the coordinate system of the origin and destination nodes.

In this method the `origin` parameter is `true` for computing the connection point at the origin of the link and `false` for the destination of the link, and the `t` parameter is the transformer used to draw the link.

This method returns `true` if a link connector is installed. If no link connector is installed, a default connection point is computed. If a link connector is installed, the link connector then computes the connection point with the method:

```
IlvPoint getConnectionPoint(IlvLinkImage link, boolean origin,IlvTransformer
t)
```

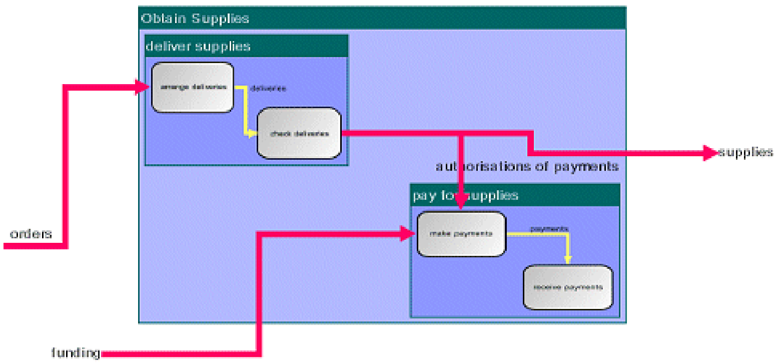
- When the link is a regular link, the `t` parameter is the transformer used to draw the link. This is the same as the transformer used to draw the origin and destination.
- When the link is an intergraph link, then the`t` parameter is the transformer used to draw the origin or destination.

So when developing a new class of links or link connectors, there is no special work to be done to take into account the specific case of intergraph links.

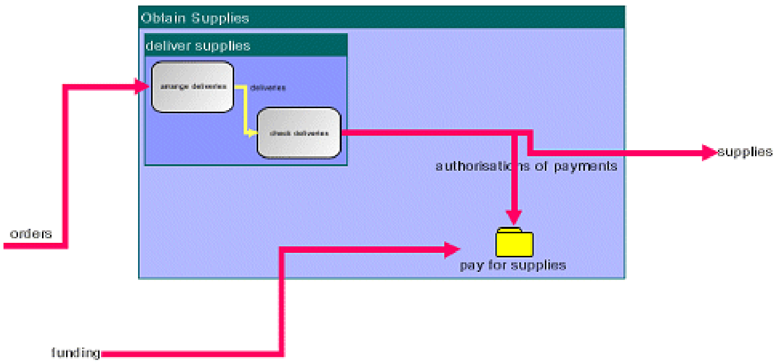
Collapsed grapher and intergraph links

When a grapher is collapsed, the graphic objects that it contains are no longer visible on the screen. The nodes that are the destination and origin of an intergraph link are not visible on the screen, and thus such intergraph links cannot visually point to their end nodes. These intergraph links will instead point to the collapsed representation of the grapher on the screen.

The following figures show an example of intergraph links to a grapher when it is expanded and then when it is collapsed.



Intergraph links to an expanded Grapher



Intergraph links to a collapsed grapher

Although the links point visually to the collapsed representation of the manager once the manager is collapsed, the real origin and destination of the links do not change. The methods `getFrom()` and `getTo()` of the link (`IlvLinkImage`) still return the same object. Only the graphical representation changes. The link visually points to the first noncollapsed parent manager.

You could consider that the methods `getFrom` and `getTo` of the link return the real end nodes of the link. The nodes that appear visually as end nodes on the screen can also be obtained by using the methods `getVisibleFrom()` and `getVisibleTo()` of the link. If there are no collapsed managers, the visible end nodes and the real end nodes are always the same

For a given node, all links that have the node as visible end node can be obtained by the methods:

```
IlvGrapher.getLinksVisibleFrom(IlvGraphic node)
```

```
IlvGrapher.getLinksVisibleTo(IlvGraphic node)
```

The links that have it as real end node are obtained by the methods:

```
IlvGrapher.getLinksFrom(IlvGraphic node)
```

```
IlvGrapher.getLinksTo(IlvGraphic node)
```

In *Intergraph links to a collapsed grapher*, two links have the visible destination node `pay for supplies`, but their real destination node is the inner node of `pay for supplies`, which is not visible in *Intergraph links to a collapsed grapher*, but can be seen in *Intergraph links to an expanded Grapher*.

If a manager is collapsed, the position of the connection point of the link is determined by the link connector installed on the collapsed manager and not by the ones installed on the real end nodes.

Creating a link using IlvMakeLinkInteractor

As well as creating an intergraph link by code, it is possible to create intergraph links (and also regular links) using the `IlvMakeLinkInteractor`.

When this interactor is installed, it allows you to interactively create a link from any graphic object to any other graphic object. This allows you to create intergraph links and also links to or from a nested grapher.

Selection in nested managers

The `IlvManager` class provides the methods that allow you to query the selection status of objects in a hierarchy of nested managers and also to listen for selection events in such a hierarchy of managers.

Selection method for a nested manager

An `IlvManager` object allows you to select objects that it contains. See Selection in a manager for more information about object selection. When a manager contains other managers, objects located in a nested manager can be selected using the same method of `IlvManager` : `setSelected(ilog.views.IlvGraphic, boolean, boolean)`.

Selection methods for a hierarchy of nested managers

The following methods allow you to deal with the specific case of selections in a hierarchy of nested managers.

```
void selectAll(boolean traverse, boolean redraw)
```

This method selects all the objects in the manager and also all the objects in nested managers when the traverse parameter is `true`.

```
void deSelectAll(boolean traverse,boolean redraw)
```

This method deselects all the selected objects in the manager and also all the selected objects in nested managers when the traverse parameter is `true`.

```
IlvGraphicEnumeration getSelectedObjects(boolean traverse)
```

This method returns an enumeration that contains all the selected objects in this manager and in nested managers if the traverse parameter is set to true.

```
int getSelectedObjectsCount(boolean traverse)
```

This method returns the number of selected objects of this manager and in the nested managers if the traverse parameter is set to `true`.

```
void deleteSelections(boolean redraw, boolean traverse, boolean redraw)
```

This method removes the selected objects in the manager and also removes the selected objects in nested managers when the traverse parameter is set to `true`.

```
IlvSelection getSelection(IlvPoint p, IlvManagerView view, boolean traverse)
```

This method returns the selection object under the specified point. The method will search for selection objects in nested managers if the traverse parameter is set to `true`.

Selection Events

When a graphic object is selected or deselected, the manager fires a selection event. This is described in Listener for the selections in a manager.

You can register a selection listener in the manager using the method:

```
void addManagerSelectionListener(ManagerSelectionListener listener)
```

The listener will only receive selection events for selections and deselections that occur in the manager where the listener was registered.

To listen to selections that are taking place throughout a hierarchy of nested managers, the `IlvManager` class provides the following methods:

```
void addManagerTreeSelectionListener(ManagerSelectionListener listener)
```

```
void removeManagerTreeSelectionListener(ManagerSelectionListener listener)
```

When you register a selection listener using these methods, whenever an object is selected or deselected from a manager that is a submanager in the hierarchy of this manager, the listener will receive the event.

Such a listener placed on the top-level manager of a hierarchy will receive all the selection events of the hierarchy. To distinguish which submanager has sent an event, you can use the method `getManager` on the event; the event is an instance of the class `ManagerSelectionChangedEvent` and contains the method:

```
IlvManager getManager()
```

Selection interactor in nested managers

The selection interactor (the class `IlvSelectInteractor`) allows you to select and edit objects interactively in a hierarchy of nested managers. Using this interactor, you can select objects in several managers that are part of the hierarchy of nested managers.

Selecting multiple objects

You can select several objects in one of these ways:

- ♦ Shift-Click on each object
- ♦ Drag a rectangle around the objects

  When you drag a rectangle, graphic objects that are inside the rectangle will be part of the selection even if the submanager that contains these objects is not fully inside the rectangle.

Moving a nested manager

To move a nested manager, click and drag in the background of the nested manager. You can start a multiple selection with a selection rectangle in one of these ways:

- ♦ Click and drag in the background of the view
- ♦ Click and drag in the background of a nested manager with the Control (Ctrl) key pressed.

  In this case the Control key distinguishes between the beginning of dragging a selection rectangle and the beginning of a move operation on a manager.

Content-change events in nested managers

When the content of the manager changes, for example, when an object is added or removed or when the bounding box of an object changes, the manager fires a `ManagerContentChangedEvent` event. Any class can listen for the modification of the content of the manager by implementing the `ManagerContentChangedListener` interface. This mechanism is described in Listener for the content of the manager.

Registering such a listener in a manager using the method `addManagerContentChangedListener(ilog.views.event.ManagerContentChangedListener)` of the manager allows the listener to receive Content Changed events only for modifications that are taking place in the manager where the listener is registered. A listener registered using this method will not be notified, for example, when a new graphic object is added to a submanager.

Note: Such a listener can nevertheless receive events indirectly due to some modifications in submanagers. For example, when a new graphic object is added in a submanager B of a manager A, the nested manager B may change size, so a listener registered on A may receive an `ObjectBBoxChangedEvent` due to the insertion of a new graphic object in B.

In order to receive all Content Changed events of a hierarchy of nested managers, the `IlvManager` class allows you to register a global listener, with the methods:

```
void addManagerTreeContentChangedListener (ManagerContentChangedListener
listener)
```

```
void removeManagerTreeContentChangedListener (ManagerContentChangedListener
listener)
```

Such a listener registered on the top-level manager of a hierarchy will receive all the Content Changed events of the hierarchy. To distinguish which submanager has sent an event, you can use the method `getManager` on the event; the event is an instance of the class `ManagerContentChangedEvent` and contains the method:

```
IlvManager getManager()
```

Note: You can register a global Content Changed listener on the top-level manager to detect all insertions of submanagers in the hierarchy of managers. If you do this, you must take into account the possibility that a manager that already contains some managers might be added to a manager. In this case the listener will not receive an event for each of these managers.

Hierarchy events in nested managers

Listeners registered with `addManagerTreeContentChangedListener(ilog.views.event.ManagerContentChangedListener)` receive events from all changes to children and their descendents in the hierarchy of nested managers.

An event is created on the manager that contains the changed object and it is propagated upwards in the hierarchy to the ancestors. Typically, the event listener is added to the root of the hierarchy.

Conversely, hierarchy events are propagated downwards in the hierarchy to the descendents. They are similar to `java.awt.event.HierarchyEvent` and notify the descendents about a change made to their ancestors. Typically, hierarchy event listeners are added to the leaves of the hierarchy.

Two types of hierarchy events exist:

♦ The `GraphicBagHierarchyEvent` installed by `addGraphicBagHierarchyListener(ilog.views.event.GraphicBagHierarchyListener),`

♦ The `ManagerViewsChangedEvent` installed by `addManagerViewsHierarchyListener(ilog.views.event.ManagerViewsChangedListener).`

Graphic bag hierarchy events

When a submanager is added to or removed from its manager, a Graphic Bag Hierarchy event is fired. It notifies all objects contained in the submanager recursively that the root of the submanager nesting hierarchy has changed. The objects that can receive this event are `IlvManager`, `IlvGraphicSet`, `IlvGraphicHandleBag` and their subclasses.

In order to add a listener for Graphic Bag Hierarchy events or remove one, you can call the following methods on a submanager (or on `IlvGraphicSet` or `IlvGraphicHandleBag`):

```
void addGraphicBagHierarchyListener (GraphicBagHierarchyListener
listener)
```

```
void removeGraphicBagHierarchyListener (GraphicBagHierarchyListener
listener)
```

The `GraphicBagHierarchyEvent` contains the following information:

```
getGraphicBag() - this is the object whose graphic bag has changed,
```

```
getOldRootBag() - this is the root graphic bag of the entire hierarchy before

the change happend,
getNewRootBag() - this is the root graphic bag of the entire hierarchy after
the change happened.
```

Note: The event is fired on the manager, graphic set or graphic handle bag whose graphic bag has changed. It is propagated recursively to all children that implement the `GraphicBagHierarchyEventReceiver` interface. It is not propagated to children whose ancestors do not implement the `GraphicBagHierarchyEventReceiver` interface, because the recursive propagation of the event stops on the object that does not implement that interface. For example, if a manager contains an object of type `IlvGraphicHandle`, which contains an `IlvGraphicSet`, then the event is not received by the `IlvGraphicSet` because the `IlvGraphicHandle` does not implement the interface. However, if the manager contains an object of type `IlvFullZoomingGraphic`, which contains an `IlvGraphicSet`, then the event is received by the `IlvGraphicSet` because both `IlvFullZoomingGraphic` and `IlvGraphicSet` implement the interface.

Manager views changed events

`ManagerViewsChangedEvent` events are fired when a view is added to or removed from a manager. To install listeners for these events on a manager, use the following methods:

- ♦ `addManagerViewsListener(ilog.views.event.ManagerViewsChangedListener)`
- ♦ `removeManagerViewsListener(ilog.views.event.ManagerViewsChangedListener)`

For details, see Listener for the views of a manager.

Listeners installed with this API will only receive events when a view is added directly to the manager. In a nested hierarchy of managers, a submanager does not receive the event if the view is added to the root manager of the hierarchy, even though the submanager will be visible in that view.

It is possible to install listeners of the `ManagerViewsChangedEvent` events so that they also receive the events from the ancestor managers. Use the following API to achieve this:

- ♦ `addManagerViewsHierarchyListener(ilog.views.event.ManagerViewsChangedListener)`
- ♦ `removeManagerViewsHierarchyListener(ilog.views.event.ManagerViewsChangedListener)`

This API can be used on `IlvManager`, `IlvGraphicSet`, `IlvGraphicHandleBag` and their subclasses. If a view is added to the root manager, the event is propagated to all the contained objects that have a listener added with this API.

Note: The event is fired by the manager where a view is added or removed. It is propagated recursively to all children that implement the

`ManagerViewsHierarchyEventReceiver` interface. It is not propagated to children whose ancestors do not implement the `ManagerViewsHierarchyEventReceiver` interface, because the recursive propagation of the event stops at an object that does not implement that interface. For example, if a manager contains an object of type `IlvGraphicHandle`, which contains an `IlvGraphicSet`, then the event is not received by the `IlvGraphicSet` because the `IlvGraphicHandle` does not implement the interface. However, if the manager contains an object of type `IlvFullZoomingGraphic`, which contains an `IlvGraphicSet`, then the event is received by the `IlvGraphicSet` because both `IlvFullZoomingGraphic` and `IlvGraphicSet` implement the interface.

Interactors for nested managers and graphers

All interactors that are part of the IBM® ILOG® JViews library and that work with individual graphic objects can work with objects embedded in a nested manager or grapher.

It is therefore possible to:

♦ Select and edit objects in nested managers using the `IlvSelectInteractor` (see *Selection interactor in nested managers*).

♦ Create links and intergraph links in nested managers using the `IlvMakeLinkInteractor` (see *Creating a link using IlvMakeLinkInteractor*).

It is also possible to have object creation interactors and specific object interactors.

Creation interactors

The same interactors that allow you to create a graphic object interactively also allow you to create objects directly in a submanager.

Such interactors all follow the same scheme: when the interaction starts in a nested manager (the first mouse click), the created object is placed in this manager. This is the case for all subclasses of `IlvMakeRectangleInteractor` and `IlvMakePolyPointsInteractor`, which are the base classes for creating an object with a rectangular shape and a set of points respectively. It is also the case for the `IlvEditLabelInteractor` which allows you to create labels.

Object interactors

Object interactors (instances of `IlvObjectInteractor`) are objects that provide interaction for a specific graphic object. For details, see Object interactors of The Essential JViews Framework.

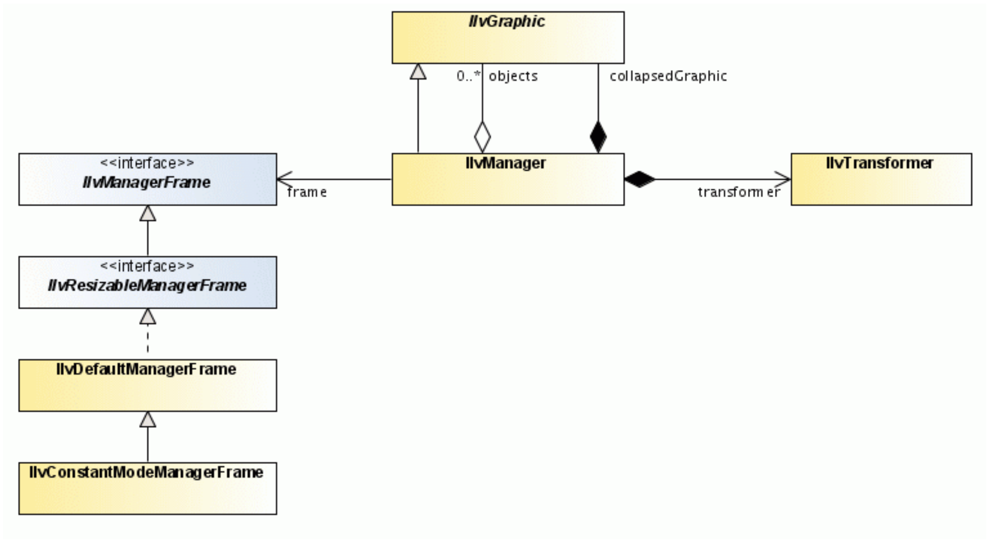
An object interactor that is set on a graphic object contained in a hierarchy of nested managers will still receive events coming from the view attached to a parent manager. There is no difference in writing an object interactor that can be used on such a graphic object.

Class diagram for nested managers

The following UML class diagram summarizes the class structure for nested managers.

`IlvManager` is a subclass of `IlvGraphic`; since `IlvManager` can contain `IlvGraphic` instances as objects, it is possible to nest managers in other managers. When you display the nested submanagers, the content is drawn and transformed using a local `IlvTransformer` instance in the submanager. For the top-level manager displayed in the manager view, the local transformer plays no role since the top-level manager is drawn and transformed by that view's `IlvTransformer`.

A manager can be collapsed or expanded. In its collapsed state, it is drawn as a representative collapsed graphic and its contents are not drawn. In its expanded state, its contents are drawn inside a manager frame.



The classes related to a nested IlvManager

Link shape policies

Describes each supplied link shape policy and explains the details of the link shape mechanism and how to extend a predefined policy or implement your own.

In this section

Overview of link shape policies

Link shape policies allow you to manipulate and to constrain the shape of a polyline link. For instance, you can use a link shape policy if you want a link always to remain orthogonal. You can set link shape policies on links of type `IlvPolicyAwareLinkImage`, which is a subclass of `IlvPolylineLinkImage`.

Important: A link shape policy affects only the shape of the link on which it is set. In other words, link shape policies are different from link layouts, which analyze and reshape all links together globally.

The package `ilog.views.graphic.linkpolicy` contains two predefined link shape policies:

- `IlvOrthogonalLinkShapePolicy` to keep the shape of a link orthogonal.
- `IlvCrossingLinkShapePolicy` to display a tunnel or bridge crossing shape when links cross.

The class `IlvEnhancedPolylineLinkImage` uses the predefined link shape policies internally. The class has a simple API that offers you all the benefits of link shape policies while hiding the details of the link shape mechanism. See Link shapes and crossing of The Essential JViews Framework.

Orthogonal link shape policy

The orthogonal link shape policy forces links to keep at right angles to one another. When a link bend is moved, the adjacent link bends automatically to follow the move so that the link always remains orthogonal.

Setting the Link Shape Policy

The orthogonal link shape policy can be set on links of class `IlvPolicyAwareLinkImage`. The policy can be shared by several links. The following code shows how to set the orthogonal link shape policy on two links.

```
IlvOrthogonalLinkShapePolicy policy =
    new IlvOrthogonalLinkShapePolicy();

IlvPolicyAwareLinkImage link1 =
    new IlvPolicyAwareLinkImage(node1, node2, true, null);
link1.setLinkShapePolicy(policy);
IlvPolicyAwareLinkImage link2 =
    new IlvPolicyAwareLinkImage(node1, node2, true, null);
link2.setLinkShapePolicy(policy);
...
// insert the links into the grapher only AFTER installing the policy
grapher.addLink(link1, redraw);
grapher.addLink(link2, redraw);
...
```

After the link shape policy is set and added to the grapher, the link remains orthogonal during all the operations that try to reshape the link. To disable the link shape policy, call:

```
link.setLinkShapePolicy(null);
```

This does not change the shape of the link immediately, but it enables subsequent operations on the link to reshape it in a nonorthogonal way again.

Chaining of link shape policies

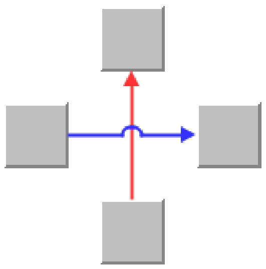
It is possible to apply multiple link shape policies to the same link, if these policies do not contradict each other. In particular, it is possible to add a crossing link shape policy onto an orthogonal link shape policy, as illustrated in the following code.

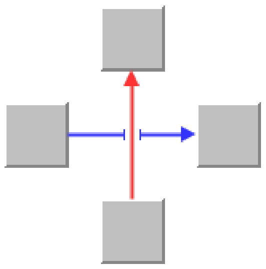
```
policy = new IlvOrthogonalLinkShapePolicy();
policy.setChildPolicy(new IlvCrossingLinkShapePolicy());
link.setLinkShapePolicy(policy);
```

In this case, the orthogonal policy first forces the link to an orthogonal shape, and then the crossing policy calculates the display of the link crossings.

Crossing link shape policy

The crossing link shape policy calculates the shape of a link that crosses other links. The crossing mode can be a tunnel or a bridge.





Tunnel crossing and bridge crossing

Setting the link shape policy

You can set the crossing link shape policy to links of class `IlvCrossingAwareLinkImage`. Use the following code:

```
IlvCrossingAwareLinkImage link =
    new IlvCrossingAwareLinkImage(node1, node2, true, null);
link.setLinkShapePolicy(new IlvCrossingLinkShapePolicy());
grapher.addLink(link, redraw);
...
```

This policy can be shared among several links. The policy parameters are valid for all links that share the same policy.

Crossing graphics

The crossing link shape policy calculates the positions of the crossings, but it is the crossing graphics that actually draw the crossings. If no crossing graphic is associated with a link, then the crossing is not displayed and a gap appears instead. A crossing graphic is an instance of a subclass of `IlvGraphic` that implements the `IlvCrossingGraphic` interface.

The package `ilog.views.graphic.linkpolicy` contains two predefined crossing graphics:

- `IlvTunnelCrossings` for tunnel crossings
- `IlvBridgeCrossings` for bridge crossings

Unlike the shape policy itself, crossing graphics cannot be shared by several links. Once a crossing graphic is associated with a link, it is fully maintained by the crossing link shape policy. Therefore, it is not necessary to move or to reshape the crossing graphic, as all this is done by the link shape policy automatically.

To set a tunnel crossing graphic, call:

```
link.setCrossingGraphic(new IlvTunnelCrossings(link));
```

To set a bridge crossing graphic, call

```
link.setCrossingGraphic(new IlvBridgeCrossings(link));
```

Example

The following code shows how to set the crossing link shape policy and the crossing graphics for two links.

```
IlvCrossingLinkShapePolicy policy = new IlvCrossingLinkShapePolicy();
IlvCrossingAwareLinkImage link1 =
    new IlvCrossingAwareLinkImage(node1, node2, true, null);
link1.setCrossingGraphic(new IlvTunnelCrossings(link1));
link1.setLinkShapePolicy(policy);
IlvCrossingAwareLinkImage link2 =
    new IlvCrossingAwareLinkImage(node3, node4, true, null);
link2.setCrossingGraphic(new IlvTunnelCrossings(link2));
link2.setLinkShapePolicy(policy);
...
// insert the links into the grapher only AFTER installing the policy
grapher.addLink(link1, redraw);
grapher.addLink(link2, redraw);
...
```

Parameters of the link shape policy

Graphers may be nested. In this case, links of different subgraphers may cross each other. Calculating the crossings therefore requires a complete traversal of the nested subgraphers. If the grapher is not nested, or if crossings of links of different subgraphers should not be displayed as tunnels or bridges, then it is useful to disable the nesting traversal of the link shape policy. This also speeds up the calculation of crossings.

To disable the nesting traversal, call

```
crossingPolicy.setNestingTraversal(false);
```

The nesting traversal is enabled by default.

To enable or disable the nesting traversal of all crossing link shape policies that occur in a nested grapher, it is more convenient to call:

```
IlvCrossingLinkShapePolicy.SetNestingTraversal(grapher, flag, traverse,
redraw);
```

If the `traverse` parameter is set to `true`, this method iterates over all links of all subgraphers and sets the flag at all crossing shape policies of the links. If the `redraw` parameter is `true`, a redraw is automatically performed afterwards.

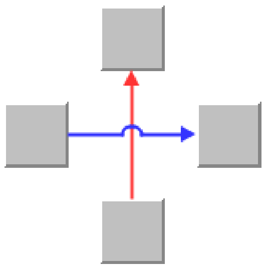
Tunnel and bridge crossing shapes can be horizontally or vertically oriented. To enable a horizontal orientation, call:

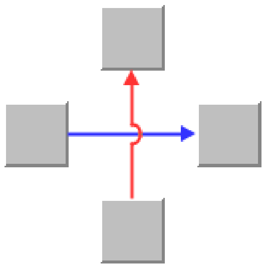
```
crossingPolicy.setHorizontalPreferred(true);
```

To enable a vertical orientation, call:

```
crossingPolicy.setHorizontalPreferred(false);
```

The horizontal orientation is the default.





Crossing orientations: horizontal and vertical

Crossing links only look nice if all link crossings are the same. This can be achieved by making all links share the same shape policy. Alternatively, you can also achieve this by calling the following method on the grapher:

```
IlvCrossingLinkShapePolicy.SetHorizontalPreferred(grapher, flag, traverse,
redraw);
```

If the `traverse` parameter is true, the same preference is applied to all crossing shape policies of all the links in all the nested subgraphers. Changing the preference changes the shape of the links; therefore it may be necessary to redraw the links. If the `redraw` parameter is true, the redraw is automatically performed.

Gaps at crossings

A link of class `IlvCrossingAwareLinkImage` draws a gap at the link crossings. The crossing graphic of the link fills this gap with a bridge or tunnel shape. You can modify the size of the gap using the method:

```
crossingAwareLinkImage.setGap(width);
```

The corresponding bridge or tunnel drawing is automatically adapted when the gap changes.

The gap between links can follow the zoom level or be independent of the zoom level. If the gap is zoomable, its size changes as you zoom in and out of the view. If the gap is nonzoomable, it remains constant during zooming. In particular with nested graphers that have different zoom levels, it may be visually more appealing if the gap is not zoomable.

To make gaps nonzoomable, call:

```
crossingAwareLinkImage.setGapZoomable(false);
```

Gaps are zoomable by default.

Obtaining the link shape policy of IlvEnhancedPolylineLinkImage instances

All instances of `IlvEnhancedPolylineLinkImage` that are in the same grapher use the same link shape policy internally.

To obtain this link shape policy, call

```
IlvLinkShapePolicy policy = IlvLinkShapePolicyUtil.GetLinkShapePolicy (mode,
grapher);
```

The parameter `mode` can be:

- `CROSSING_POLICY_MODE` retrieves the link shape policy of all links that have the crossing mode enabled and the orthogonal mode disabled.
- `ORTHOGONAL_POLICY_MODE` retrieves the link shape policy of all links that have the crossing mode disabled, that is, the mode is `NO_CROSSINGS`, but the orthogonal mode enabled.
- A bitwise Or combination of both – the link shape policy that has the crossing mode and the orthogonal mode enabled.

This allows you to access the link shape policies that are used internally by the class `IlvEnhancedPolylineLinkImage` so as to change the parameters of these policies.

Defining your own link shape policy

Describes the callback methods of the class `IlvLinkShapePolicy` and uses an example to explain how you can define your own link shape policy.

In this section

Callback methods of IlvLinkShapePolicy

In addition to the predefined link shape policies, the JViews Framework allows you to define your own link shape policies.

In general, when a method of `IlvPolicyAwareLinkImage` is called that may change the shape of the link, the following procedure is used by the policy-aware link image:

1. The link shape policy is temporary disabled to avoid endless recursion when calling methods on the link inside the link shape policy.
2. The link shape policy is asked whether the change is allowed. For instance, when the method `link.insertPoint` is called, the policy is asked by method `policy.allowInsertPoint` whether inserting the point is allowed.
3. If the change is allowed, it is done (for example, the point is inserted).
4. The "after change" callback method of the link shape policy is called. For instance, inside `link.insertPoint`, the method `policy.afterInsertPoint` is called. This allows the policy to react to the change.
5. The link shape policy is enabled again.
6. The method `afterAny` is called at the end. Since, at this point, the link shape policy is enabled, this method should not reshape the link any further. It can however perform cleanup operations.

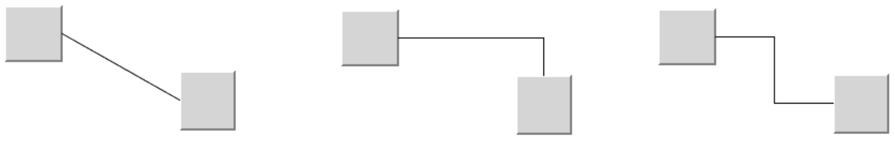
Callback methods of IlvLinkShapePolicy used by IlvPolicyAwareLinkImage

Method of IlvPolicyAwareLinkImage	Test Method in IlvLinkShapePolicy	After Method in IlvLinkShapePolicy	Comment
setLinkShapePolicy		onInstall	Applied to the new policy
setLinkShapePolicy		onUninstall	Applied to the old policy
insertPoint	allowInsertPoint	afterInsertPoint	
removePoint	allowRemovePoint	afterRemovePoint	
movePoint	allowMovePoint	afterMovePoint	
setIntermediateLinkPoints	allowSetIntermediateLinkPoints	afterSetIntermediateLinkPoints	
applyTransform	allowApplyTransform	afterApplyTransform	
		afterAdd	Applied after a link has been added to a grapher
		beforeRemove	Applied before a

Method of IlvPolicyAwareLinkImage	Test Method in IlvLinkShapePolicy	After Method in IlvLinkShapePolicy	Comment
			link is removed from a grapher
		afterFromNodeMoved	Applied when the source node of the link has been moved
		afterToNodeMoved	Applied when the destination node of the link has been moved
		afterAny	Applied after any of the methods mentioned above

Creating a link shape policy with up to two bends

The example creates a link shape policy that allows links to have 0, 1 or 2 bend points. With 1 or 2 bends, the link will have an orthogonal shape. The link shape policy prohibits the creation of more than 2 bends.



Link shape policy for 0, 1 or 2 bends

In principle, this link shape policy is a combination of the functionality of the classes `IlvOneLinkImage` and `IlvDoubleLinkImage` (see Links of The Essential JViews Framework). However, using a link shape policy is more flexible, because the link shape policy can be enabled and disabled on the link, and can change from a 1-bend image to a 2-bend image.

The class `IlvAbstractLinkShapePolicy` is a suitable base class for the new link shape policy because it defines all the methods of the interface `IlvLinkShapePolicy` as empty methods.

```
public class MyLinkShapePolicy
    extends IlvAbstractLinkShapePolicy
{
    ...
}
```

You can concentrate on the few methods that you need to override.

To create your link shape policy:

1. Create an auxiliary method that reshapes the link according to the policy. Depending on the number of bends and the horizontal and vertical distance, you can decide which shape the link should have. For instance:

```
private void verifyLinkPoints(IlvLinkImage link)
{
    if (link == null)
      return;

    IlvPoint[] pts = link.getLinkPoints(null);
    int n = pts.length;
    if (n <= 2) return;

    IlvRect fromrect = link.getFromBoundingBox(null);
    IlvRect torect = link.getToBoundingBox(null);
    float fx = fromrect.x + 0.5f * fromrect.width;
    float fy = fromrect.y + 0.5f * fromrect.height;
    float tx = torect.x + 0.5f * torect.width;
    float ty = torect.y + 0.5f * torect.height;
    float dx = (fx < tx ? 0.5f : -0.5f);
    float dy = (fy < ty ? 0.5f : -0.5f);

    if (n == 3) {
      link.movePoint(0, fx + dx * fromrect.width, fy, null);
      link.movePoint(1, tx, fy, null);
      link.movePoint(2, tx, ty - dy * torect.height, null);
      return;
    }

    if (Math.abs(fx - tx) > Math.abs(fy - ty)) {
      float middleX = 0.5f * (fx + tx +
                        dx * (fromrect.width - torect.width));
      pts[1].move(middleX, fy);
      pts[2].move(middleX, ty);
      link.movePoint(0, fx + dx * fromrect.width, fy, null);
      link.movePoint(n-1, tx - dx * torect.width, ty, null);
    } else {
      float middleY = 0.5f * (fy + ty +
                        dy * (fromrect.height - torect.height));
      pts[1].move(fx, middleY);
      pts[2].move(tx, middleY);
      link.movePoint(0, fx, fy + dy * fromrect.height, null);
      link.movePoint(n-1, tx, ty - dy * torect.height, null);
    }

    link.setIntermediateLinkPoints(pts, 1, 2);
}
```

2. When the link policy is set on a link, or when a link is added to the grapher, you have to ensure that the link shape is correct. Call the method `verifyLinkPoints` inside the appropriate callback methods of the link shape policy.

   The callback method `afterAdd` is called when the link is already in the grapher. Since the link shape policy will reshape the link and might change the bounding box of the link, you must call the method `applyToObject`:

```
public void afterAdd(IlvLinkImage link)
{
    if (link.getGraphicBag() != null) {
        link.getGraphicBag().applyToObject(link,
            new IlvApplyObject() {
                public void apply(IlvGraphic obj, Object arg) {
                    verifyLinkPoints((IlvLinkImage)obj);
                }
            }, null, true);
        }
        else verifyLinkPoints(link);
    super.afterAdd(link);
}
```

3. When you set the link shape policy, you can rely on the user to call `applyToObject`, therefore you do not need to use this method again inside the link shape policy. Hence, the code of the method `onInstall` is simpler.

```
public void onInstall(IlvLinkImage link)
{
    verifyLinkPoints(link);
    super.onInstall(link);
}
```

4. The link shape policy allows the link to have 0, 1, or 2 bends. Since the two end points of the link are counted in the link cardinality, you just have to forbid insertion of more than four points, so that there will never be more than two bends:

```
public boolean allowInsertPoint(IlvLinkImage link,
                                int index,
                                float x, float y,
                                IlvTransformer t)
{
    int n = link.getPointsCardinal();
    if (n >= 4) return false;
    return true;
}
```

5. When bends are inserted or removed, the link policy must reshape the link so that it has the shape desired for this number of bends. Therefore, you override the methods `afterInsertPoint`, `afterRemovePoint`, and `afterSetIntermediateLinkPoints` to call `verifyLinkPoints`. Note that all these methods are always called inside the method `applyToObject`. Thus, the code can omit an additional call to `applyToObject` similar to `onInstall`:

```
public void afterInsertPoint(IlvLinkImage link, int index,
                                          IlvTransformer t)
{
    verifyLinkPoints(link);
    super.afterInsertPoint(link, index, t);
}

public void afterRemovePoint(IlvLinkImage link, int index,
                                         IlvTransformer t)
{
    verifyLinkPoints(link);
    super.afterRemovePoint(link, index, t);
}

public void afterSetIntermediateLinkPoints(IlvLinkImage link)
{
    verifyLinkPoints(link);
    super.afterSetIntermediateLinkPoints(link);
}
```

6. Changing the transformation of a polyline link image usually also modifies the bends of the link. Therefore the link points need to be verified here as well.

```
public void afterApplyTransform(IlvLinkImage link, IlvTransformer t)
{
    verifyLinkPoints(link);
    super.afterApplyTransform(link, t);
}
```

7. When an end node of the link is moved, the 1 or 2 bends of the link must adjust accordingly. Therefore it is necessary to override the corresponding callback methods of the link shape policy.

```
public void afterFromNodeMoved(IlvLinkImage link)
{
    verifyLinkPoints(link);
    super.afterFromNodeMoved(link);
}

public void afterToNodeMoved(IlvLinkImage link)
{
    verifyLinkPoints(link);
    super.afterToNodeMoved(link);
}
```

8. The link shape policy defines the positions of the bends completely. Therefore the policy forbids modification of the positions of the bends; that is, you are not allowed to move a point.

```
public boolean allowMovePoint(IlvLinkImage link,
                              int index,
                              float x, float y,
                              IlvTransformer t)
{
    return false;
}
```

The callback method `afterMovePoint` of the link shape policy could be used to adjust the link shape after moving points. However, since this link shape policy does not allow point movement, the method `afterMovePoint` need not be overridden because point movements will never be executed.

The new link shape policy works for all subclasses of `IlvPolicyAwareLinkImage`. It can be set on a link as follows:

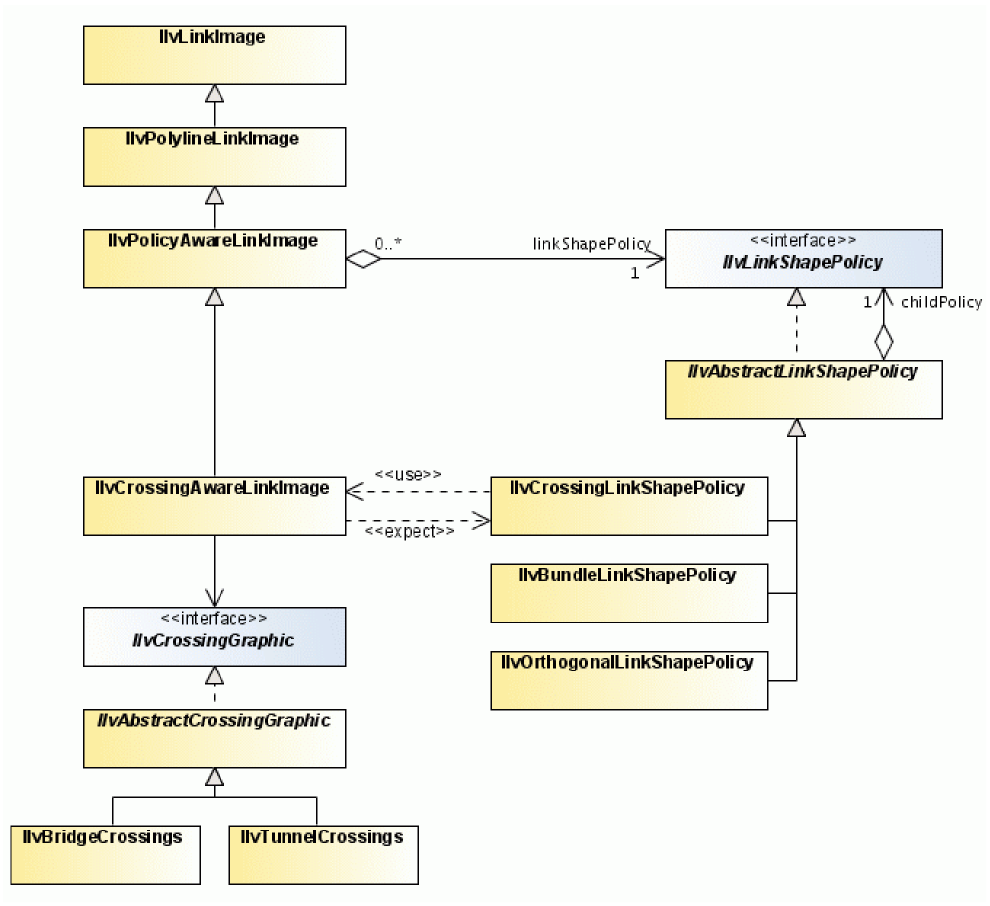
```
link.setLinkShapePolicy(new MyLinkShapePolicy());
```

Note: For simplicity, the example class `MyLinkShapePolicy` is designed for zoomable, rectangular end nodes that have no link connector or a free link connector (`IlvFreeLinkConnector`). If this is not the case, the method `verifyLinkPoints` needs to be adjusted to analyze the current transformer, the shape of the end nodes, and the link connector.

Class diagram for link shape policies

The following UML class diagram summarizes all classes related to `IlvLinkShapePolicy`.

Subclasses of `IlvPolicyAwareLinkImage` can have link shape policies. Normally, link shape policies can be shared by different links. Subclasses of `IlvAbstractLinkShapePolicy` allow you to link the policies using the child policy handle. This allows you to install multiple policies on a link. The `IlvCrossingLinkShapePolicy` calculates the crossing points of an `IlvCrossingAwareLinkImage`. The crossings are drawn using a crossing graphic; that is, an `IlvGraphic` that additionally implements the `IlvCrossingGraphic` interface.



The classes related to IlvLinkShapePolicy

The generic printing framework

Describes the generic printing framework.

In this section

Overview of the support for printing

IBM® ILOG® JViews provides the following support for printing:

♦ A generic printing framework

♦ Extensions to the generic printing framework

You can use the generic printing framework to implement printing capabilities in your Java™ applications. You can specify printing parameters, preview the document before printing it, and print the document by sending it to the specified printer.

Note: The printing framework is independent of any IBM® ILOG® JViews product, and therefore can be used to print other types of documents.

The printing framework classes are defined in the package `ilog.views.util.print`.

IBM® ILOG® JViews includes several extensions of the printing framework to help you print the content of various components, such as 2D graphics in the JViews Framework (documented in *Printing framework for manager content*), Gantt diagrams, or Charts. See the documentation of these products for more information.

Java print package and printing API

Describes the interface for printing and explains how to use the main class in the printing API to print a component.

In this section

Overview

The printing framework is implemented by using the printing APIs which have been available since JDK™ 1.3 and JDK 1.4.

The printable interface

The `Printable` interface is the most important interface of the `java.awt.print` package. To make a UI component printable, you need to implement this interface, which has a single method:

```
public int print(Graphics graphics,
                 PageFormat pageFormat,
                  int pageIndex) throws PrinterException
```

This method is called by the Java™ printing system when it sends the printing instructions to a printer. To print your component, you need to call the drawing methods of the `Graphics` object passed as the first argument.

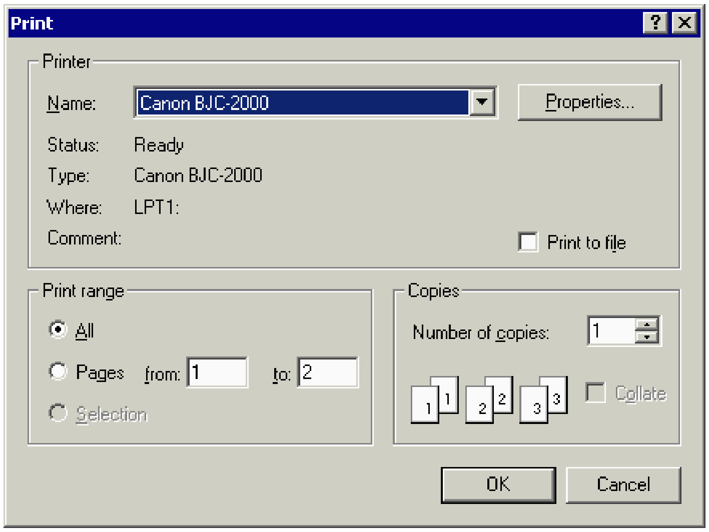
While printing your component, you can get more information on the type of paper being currently used from the second and the third arguments of this method.

After implementing the `Printable` interface, you need to create a `PrinterJob` object to print your component.

Using the PrinterJob class to print a component

The `PrinterJob` class is the root of the `java.awt.print` package. To print your component, you must create a `PrinterJob` object. The `PrinterJob` class has a static method called `getPrinterJob()`.

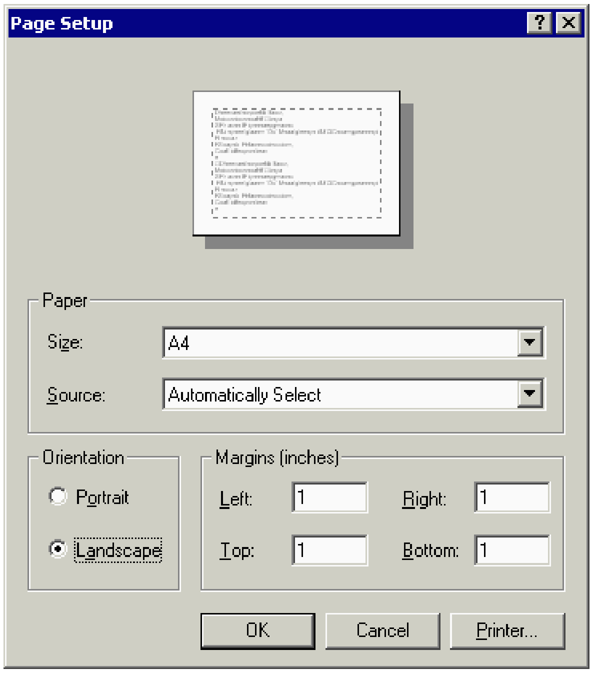
1. Call method `getPrinterJob()` to create a printer job.
2. Call the `setPrintable(Printable painter)` method to specify the `Printable` object you have implemented.
3. To change the printer parameters you can open the `Print` dialog box by calling the `printDialog()` method.



Name: Canon BJC-2000
Status: Ready
Type: Canon BJC-2000
Where: LPT1:
Comment:
Print range: All; Pages from: 1 to: 2
Number of copies: 1

Print dialog box of Java SE

4. If you want to change the page format, open the `Page Setup` dialog box by calling the `pageDialog()` method.



Size: A4

Source: Automatically Select

Portrait ( ) Landscape (●)

Left: 1 Right: 1

Top: 1 Bottom: 1

Page Setup Dialog Box of Java SE

5. To send the printing job to the selected printers, call the `print()` method of the printer job.

See the documentation of the `PrinterJob` class for more information.

The printing framework

Describes the printing framework supplied with the JViews Framework, specifically the Document Model classes, and provides examples of its use.

In this section

The PrintingController class
Describes how to use the PrintingController class.

Creating an IlvDocument with printable objects
Describes how to print a simple example document.

Creating an IlvDocument with a flow of text
Describes how to print an example document with text.

Features of the printing framework

The generic printing framework is designed and implemented to meet the printing needs of IBM® ILOG® JViews packages but you can also use it to make other printing applications. Based on the Model-View-Controller (MVC) architecture, the printing framework consists of the following features:

- ♦ *The Document Model classes*

  The framework provides a set of classes that you can use to create a printable document. You can use the built-in document class and page class to easily create the document you want to print.

- ♦ *Printing user interface components*

  The printing framework provides user interface components, such as preview frame and toolbar, that allow you to preview the document you have created. It also provides customizable dialog boxes for setting page properties such as paper size, orientation, header and footers.

- ♦ *The PrintingController class*

  The printing controller is a high-level class that manages the document to print and the preview frame. You can use it to invoke the Print dialog box, or the Page Setup dialog box, to preview the document, and to send the document to the selected printer.

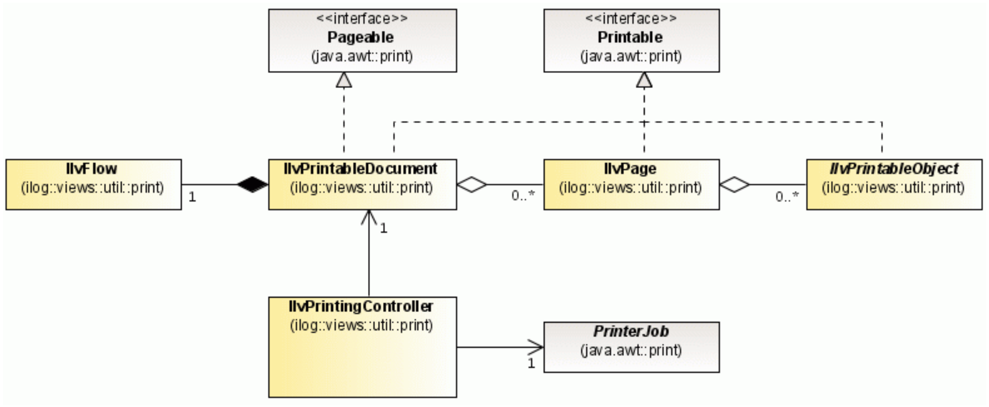
The process for printing multiple pages is as follows:

1. Create and fill an `IlvPrintableDocument` object by using the document model classes.
2. Create an `IlvPrintingController` object for the `IlvPrintableDocument` object you created. Then you can call its methods to preview and print the `IlvPrintableDocument`.

The Document Model classes

The document model provides a structure for defining a multipage document to be printed. The document model allows you to concentrate your efforts on creating your printable document without worrying about how the document is previewed and printed. The printing framework previews and prints the document for you.

Print document model shows the relationship between the main classes in the document model.



Print document model

The PrintableDocument class

The `IlvPrintableDocument` class is the top-level class of the document model. It is designed as a page holder to which you can add pages defined by the `IlvPage` class.

There are two ways to fill a document:

♦ Simply create pages (`IlvPage` class) and add them to the document using the `addPage (ilog.views.util.print.IlvPage)` and `removePage(ilog.views.util.print.IlvPage)` methods.

♦ Associate an `IlvFlow` object with the document.

  The `IlvFlow` object allows you to add styled text to a document; if you decide to use an `IlvFlow` object, it will create and manage the pages of the document for you depending on the text, the alignments, and the styling properties that you specify in the `IlvFlow` object.

  Later in this topic you will see how to associate an `IlvFlow` object with a document.

Note: Some IBM® ILOG® JViews packages provide subclasses of the `IlvPrintableDocument` class that manage the pages for you. For example, IBM® ILOG® JViews Gantt provides a subclass that can automatically create its pages to display a chart in multiple pages. For more information see the IBM® ILOG® JViews Gantt documentation.

The Page class

An `IlvPage` object represents a physical page to be printed. It must be added to a printable document so that the page can be printed by a printer, or previewed by the preview framework. The `IlvPage` is implemented as a collection of printable objects. It has the `addPrintableObject(ilog.views.util.print.IlvPrintableObject)` and `removePrintableObject(ilog.views.util.print.IlvPrintableObject)` methods, which you can use to manage the printable objects in a page.

Once a page is added to a printable document, you can get the index of the page in that document by calling the `getPageIndex()` method. The index of the page is determined by the order in which it is added. The `getDocument()` method allows you to know the owner of the printable page.

To know the page format of the page, you can call the `getPageFormat()` method. By default, this method returns the page format of the document.

The Printable class

The Printable objects are the basic elements in the printable document model. They represent concrete objects such as label, line, and rectangle printed on a page. Therefore, printable objects must be added to pages so that they can be previewed and printed.

A printable object class must implement the java.awt.print.Printable interface. In other words, you have to write the print() method of this interface if you want to implement a new printable object. See *Java print package and printing API* and the JDK™ documentation for more information on this interface.

The definition of the print() method is as follows:

```
public int print(Graphics graphics, PageFormat pageFormat, int pageIndex)
          throws PrinterException
```

To print, you need to call the methods of the specified Graphics object. You are responsible for setting the correct clip and transformation to position the printing results as desired.

Some commonly used printable objects are provided as built-in classes in the package.

Classes for common printable objects

Class	Purpose
IlvPrintableLine	To print a line.
IlvPrintableRectangle	To print a rectangle.
IlvPrintableLabel	To print a label.
IlvPrintableTable	To print a portion of the JTable.
IlvPrintableTableHeader	To print a portion of the JTableHeader.

.

Note: Some IBM® ILOG® JViews packages provide additional subclasses of IlvPrintableObject. For example, IBM® ILOG® JViews Gantt provides the subclass IlvPrintableTimeScale to print its time scale. For more information, see the IBM® ILOG® JViews Gantt documentation.

The header and footer classes

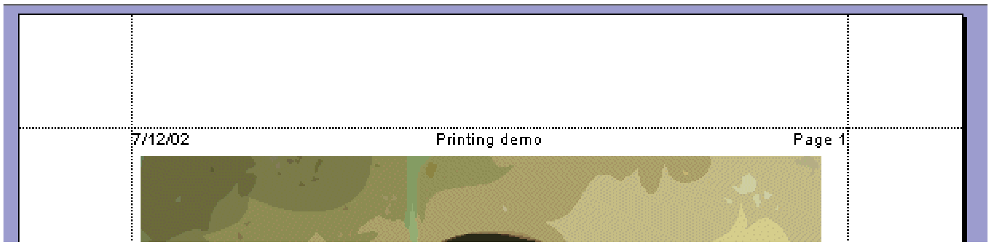
To manage page headers and footers, the printing framework also contains two additional subclasses of `IlvPrintableObject`: `IlvHeader` and `IlvFooter`. Although these two classes are subclasses of the `IlvPrintableObject`, you do not add them to the page like other printable objects. The header and footer are common to all pages of a document, and thus are set on the instance of the `IlvDocument`class.

Here are the methods of the class `IlvPrintableDocument` to set a header or a footer:

```
public void setFooter(IlvFooter footer)
public IlvFooter getFooter()

public void setHeader(IlvHeader header)
public IlvHeader getHeader()
```

The `IlvHeader` and `IlvFooter` classes are very similar. A header or footer is defined by three text sections. Each section can have a specified font.

Here is an example of a header:



Example of a header

Each of the three text sections of a header or footer can contain the text that you specify in the constructor of the object. For the header shown in *Example of a header* it would be:

```
new IlvHeader("7/12/02", "Printing demo", "Page 1");
```

Since the header and footer are defined on the document, you should not specify the page number as in the previous example. The `IlvHeader` and `IlvFooter` classes provide a certain number of keys that will be translated to values from the document, when the document is printed.

The list of keys that you can use is as follows:

- `static String AuthorKey` - The key for the author.
- `static String DateKey` - The key for the date.
- `static String FileKey` - The key for the file name.
- `static String PageKey` - The key for the page number.

- `static String PagesKey` - The key for the number of pages in the document.
- `static String TimeKey` - The key for the printing time

To create the header in *Example of a header*, use the following `new` statement:

```
new IlvHeader(IlvHeader.DateKey, "Printing demo", "Page " + IlvHeader.PageKey)
```

Note: The printing framework provides a page dialog box that also allows you to change the header and footer of a document.

The IlvFlow class for creating a document with styled text

A document can be filled in one of two ways: either you decide to create pages, fill them with `IlvPrintableObject` instances, and add them to a document, or you create a document that contains styled text. In the second case you use the `IlvFlow` class.

The `IlvFlow` class represents a flow of text with styling attributes that constitutes the content of the document to be printed. A document can have only one `IlvFlow` object, which is created using:

```
public IlvFlow IlvPrintableDocument.getFlow()
```

Once the flow is created you can specify its content using the following methods:

- `void add(String text)`

  Adds the specified text to the current paragraph.

- `void add(Image image, int alignment)`

  Adds the image to the current line of text using the specified alignment on the line.

- `void add(IlvFlowObject object, int alignment)`

  Adds the object in the current line of text using the specified alignment on the line.

The `IlvFlowObject` interface allows you to add virtually any kind of drawing to a flow. Some IBM® ILOG® JViews packages provide classes that implement this interface, so that you can add an IBM® ILOG® JViews Charts component to a flow of text.

You can also control the paragraphs and the pages of the flow using the following methods:

- `void newLine()`

  Starts a new paragraph in the flow of text.

- `void newPage()`

  Starts a new page in the flow of text.

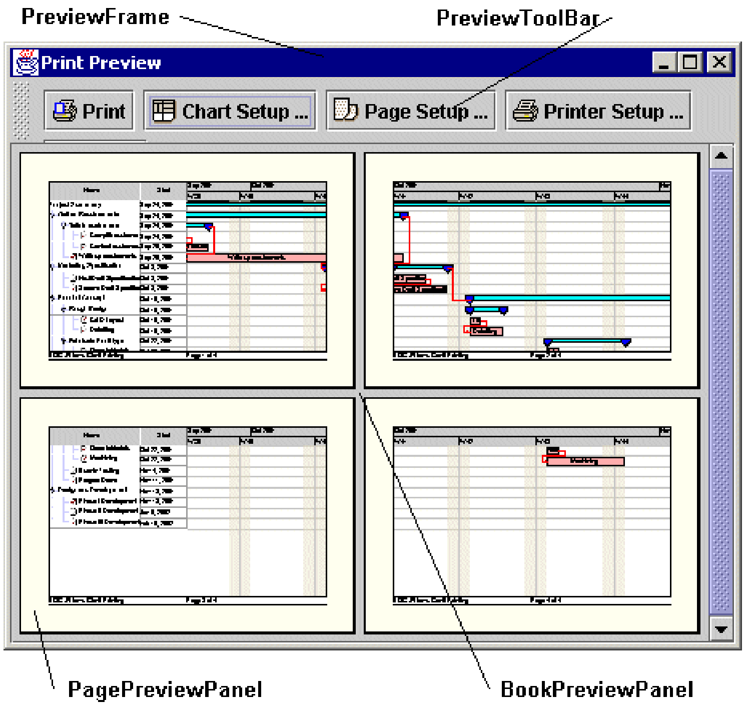
Each text added in the flow can be styled using the following method:

- `void setTextStyle(IlvFlow.TextStyle style)`

By calling this method, you change the current text style that will be used for the next text added to the flow. The `TextStyle` object allows you to change the colors (background and foreground color) of the text, the font used, and also the paragraph alignment.

Printing user interface components

The `print` package provides user interface components for previewing a document and for setting document properties such as the page format, the headers, or the footers.

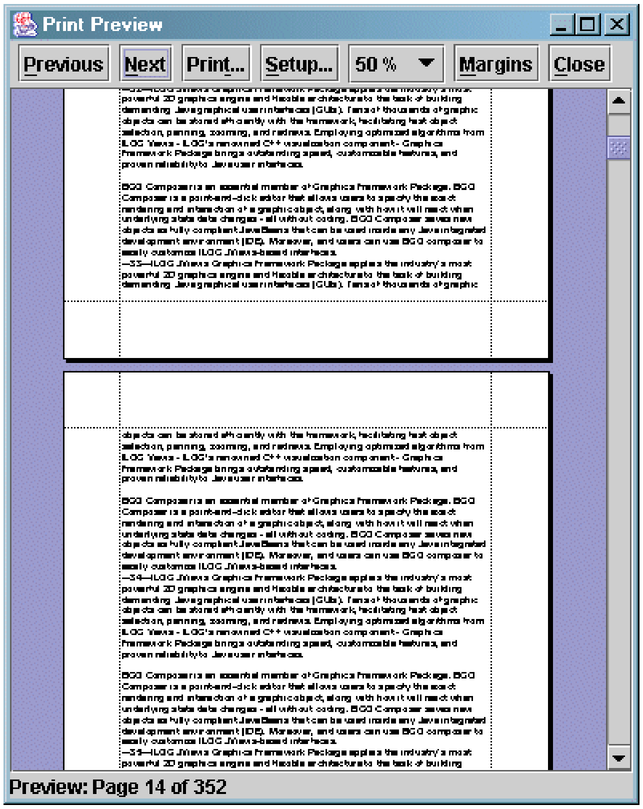


Print Preview components

When you want to preview a document to print, the following classes are involved.

- `IlvPrintPreviewPanel` - A Swing panel that displays the document to preview.
- `IlvPrintPreviewDialog` - A Swing dialog that contains an `IlvPrintPreviewPanel`.

Here is an image showing an `IlvPrintPreviewDialog` containing an `IlvPrintPreviewPanel`:



powerful 2D graphics engine and flexible architecture to the task of building demanding Java graphical user interfaces (GUIs). Tens of thousands of graphic objects can be stored efficiently with the framework, facilitating fast object selection, panning, zooming, and redraws. Employing optimized algorithms from ILOG Views - ILOG's renowned C++ visualization component - Graphics Framework Package brings outstanding speed, customizable features, and proven reliability to Java user interfaces.

BGO Composer is an essential member of Graphics Framework Package. BGO Composer is a point-and-click editor that allows users to specify the exact rendering and interaction of a graphic object, along with how it will react when underlying state data changes - all without coding. BGO Composer saves new objects as fully compliant JavaBeans that can be used inside any Java integrated development environment (IDE). Moreover, end users can use BGO composer to easily customize ILOG JViews-based interfaces.
—33—ILOG JViews Graphics Framework Package applies the industry's most powerful 2D graphics engine and flexible architecture to the task of building demanding Java graphical user interfaces (GUIs). Tens of thousands of graphic

objects can be stored efficiently with the framework, facilitating fast object selection, panning, zooming, and redraws. Employing optimized algorithms from ILOG Views - ILOG's renowned C++ visualization component - Graphics Framework Package brings outstanding speed, customizable features, and proven reliability to Java user interfaces.

BGO Composer is an essential member of Graphics Framework Package. BGO Composer is a point-and-click editor that allows users to specify the exact rendering and interaction of a graphic object, along with how it will react when underlying state data changes - all without coding. BGO Composer saves new objects as fully compliant JavaBeans that can be used inside any Java integrated development environment (IDE). Moreover, end users can use BGO composer to easily customize ILOG JViews-based interfaces.
—34—ILOG JViews Graphics Framework Package applies the industry's most powerful 2D graphics engine and flexible architecture to the task of building demanding Java graphical user interfaces (GUIs). Tens of thousands of graphic objects can be stored efficiently with the framework, facilitating fast object selection, panning, zooming, and redraws. Employing optimized algorithms from ILOG Views - ILOG's renowned C++ visualization component - Graphics Framework Package brings outstanding speed, customizable features, and proven reliability to Java user interfaces.

BGO Composer is an essential member of Graphics Framework Package. BGO Composer is a point-and-click editor that allows users to specify the exact rendering and interaction of a graphic object, along with how it will react when underlying state data changes - all without coding. BGO Composer saves new objects as fully compliant JavaBeans that can be used inside any Java integrated development environment (IDE). Moreover, end users can use BGO composer to easily customize ILOG JViews-based interfaces.
—35—ILOG JViews Graphics Framework Package applies the industry's most powerful 2D graphics engine and flexible architecture to the task of building

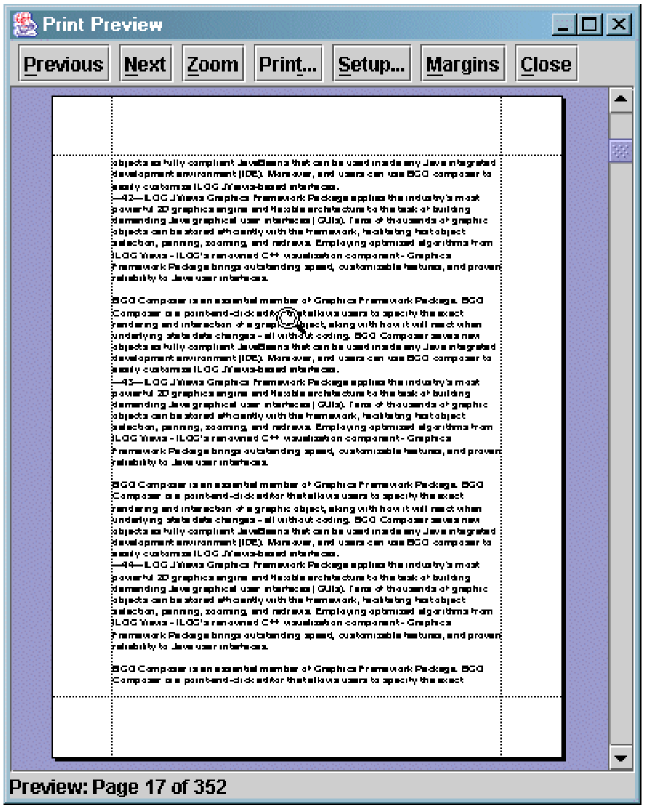
Preview: Page 14 of 352

Print Preview in Continuous Mode

The Print Preview user interface components can work in two main modes. *Print Preview in Continuous Mode* shows a Print Preview panel in continuous mode.

In this mode, the preview component displays several pages at once, and you can scroll the pages using the scroll bar. In this mode, you can also change the zoom level using the zoom level combo box.

The second mode is the single page mode. In this mode, you can see only one page at a time, but you can still use the scroll bar to move from page to page.



Print Preview in Single Page Mode

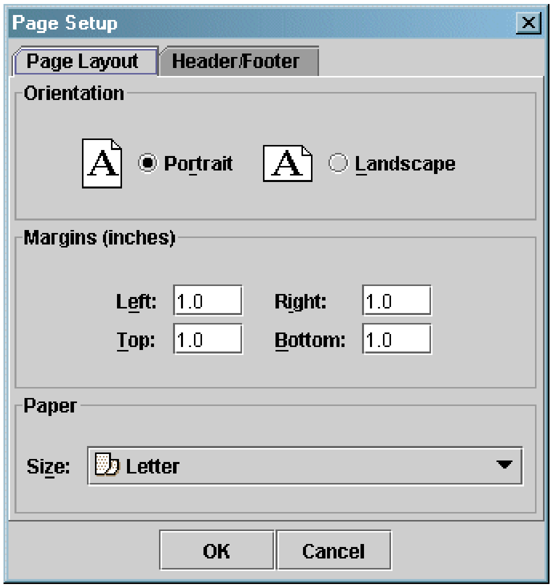
The single page mode also allows you to zoom the page. Using the zoom button (or clicking the view), you can switch from a zoom mode where the page fits the component to the mode where the page is at real size.

In most cases, you do not need to create instances and manage these components by yourself. The high-level class `IlvPrintingController` allows you to perform most of the printing actions and settings.

Page Setup component

The class `IlvDocumentSetupDialog` is a dialog box that allows you to edit various parameters of your document such as the page format, the header and the footer.

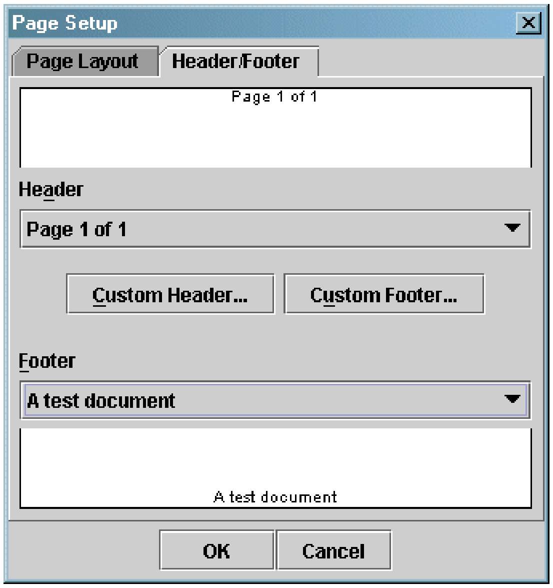
This dialog box is composed of a `JTabbedPane` with two tabs: one to edit the page format, and one to edit the header and footer.



Orientation: [✓ Portrait] Landscape

Left: 1.0 Right: 1.0
Top: 1.0 Bottom: 1.0

Size: Letter



Page 1 of 1

Header
Page 1 of 1

Footer
A test document

A test document

Page Setup

This component can be customized for your particular needs. You may add a new tab by using the `addTab(java.lang.String, java.awt.Component)` method, if you need to edit additional parameters for a specialized class of document.

In most cases, you do not need to create instances and manage the `IlvDocumentSetupDialog` by yourself. The high-level class `IlvPrintingController` allows you to perform most of the printing actions and settings.

The PrintingController class

After you create your `IlvPrintableDocument`, you will have to create an instance of `IlvPrintingController`, a high-level class that manages the document to print and the preview frame. Then you can perform various actions on the document to be printed:

♦ Call the `printDialog()` method to invoke a dialog box to select a printer, or to specify other printer-related information.

♦ Call the `setupDialog(java.awt.Window, boolean, boolean)` method to open the Page Setup dialog box, which allows you to change the page format, the header and footers. You can also click the **Setup** button while you preview the document.

♦ Call the `printPreview(java.awt.Window)` method to preview your document.

♦ Call the `print(boolean)` method of the print controller to print your document. You can also click the print button while you preview the document.

The printing controller also offers methods that you can overwrite, so that you can customize the Setup dialog, or the Print Preview dialog created by the printing controller.

Creating an IlvDocument with printable objects

This is a simple example to explain how to use the printing framework.

To print the simple example:

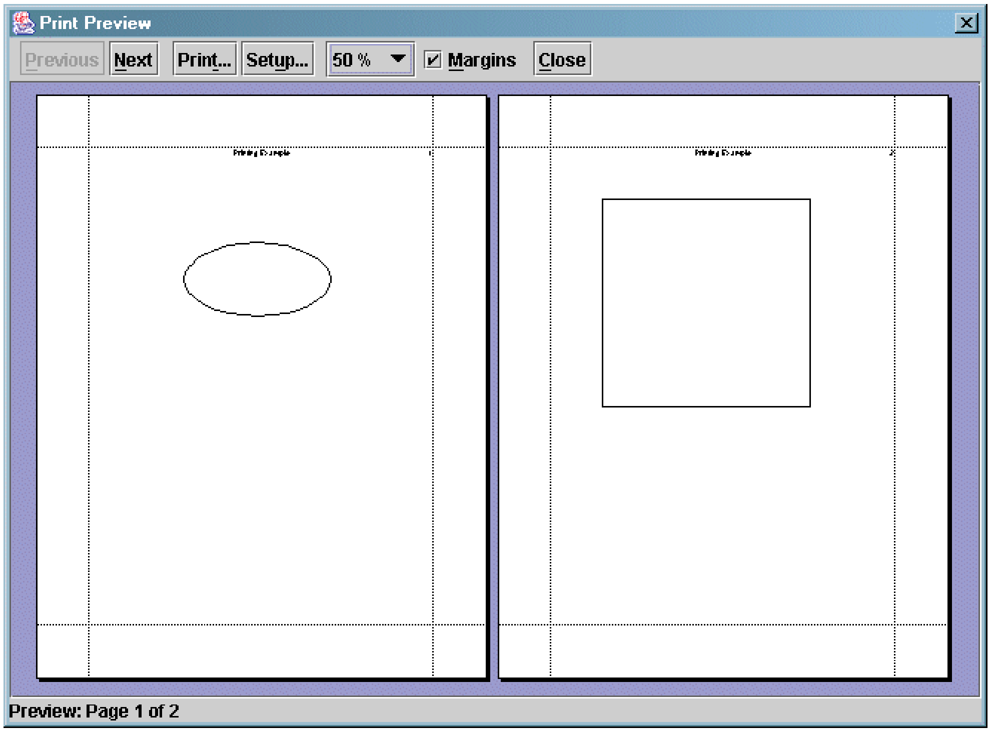
1. Create an `IlvPrintableDocument` object.
2. Add pages to the document.
3. Set header and footer to the document.
4. Implement an `IlvPrintableObject` object.
5. Create and use the printing controller.
6. Use the user interface components to preview the document you created.

The example code is as follows.

Creating a document with printable objects

```
IlvPrintableDocument document = new IlvPrintableDocument("A test document");

    // Sets a header in the document.
    document.setHeader(new IlvHeader(null, "Printing Example", IlvHeader.
PageKey));

    // Creates the first page.
    IlvPage page1 = new IlvPage();

    // Prints an oval on the first page.
    IlvPrintableObject oval = new IlvPrintableObject() {

   /**
     * Overrides the <code>print</code> method to print an oval.
     * It shows how to implement a <code>PrintableObject</code>.
     * @param dst The output <code>Graphics</code> object.
     * @param format The page format, which is ignored here.
     * @param pageIndex The page index, which is ignored here.
     */
       public int print(Graphics dst, PageFormat format, int pageIndex)
      throws PrinterException {
          dst.drawOval(200, 200, 200, 100);
          return Printable.PAGE_EXISTS;
       }
      };

    // Adds this new printable to the page.
    page1.addPrintableObject(oval);

    // Creates the second page.
    IlvPage page2 = new IlvPage();

    // Prints a rectangle on the second page.
    IlvUnit.Rectangle area = new IlvUnit.Rectangle(5, 5, 10, 10, IlvUnit.CM);

    IlvPrintableRectangle rectangle = new IlvPrintableRectangle(area);
    page2.addPrintableObject(rectangle);

    // Adds the pages to the document.
    document.addPage(page1);
    document.addPage(page2);

    // Creates the print manager.
    IlvPrintingController controller = new IlvPrintingController(document);

    // Previews the document.
    controller.setPreviewMode(IlvPrintPreviewPanel.CONTINUOUS_MODE);
    controller.printPreview(JOptionPane.getRootFrame());
```

Here is what you get on the screen:



Document with printable objects

Creating an IlvDocument with a flow of text

This is an example to explain how to use the printing framework with a text flow.

To print the text flow example:

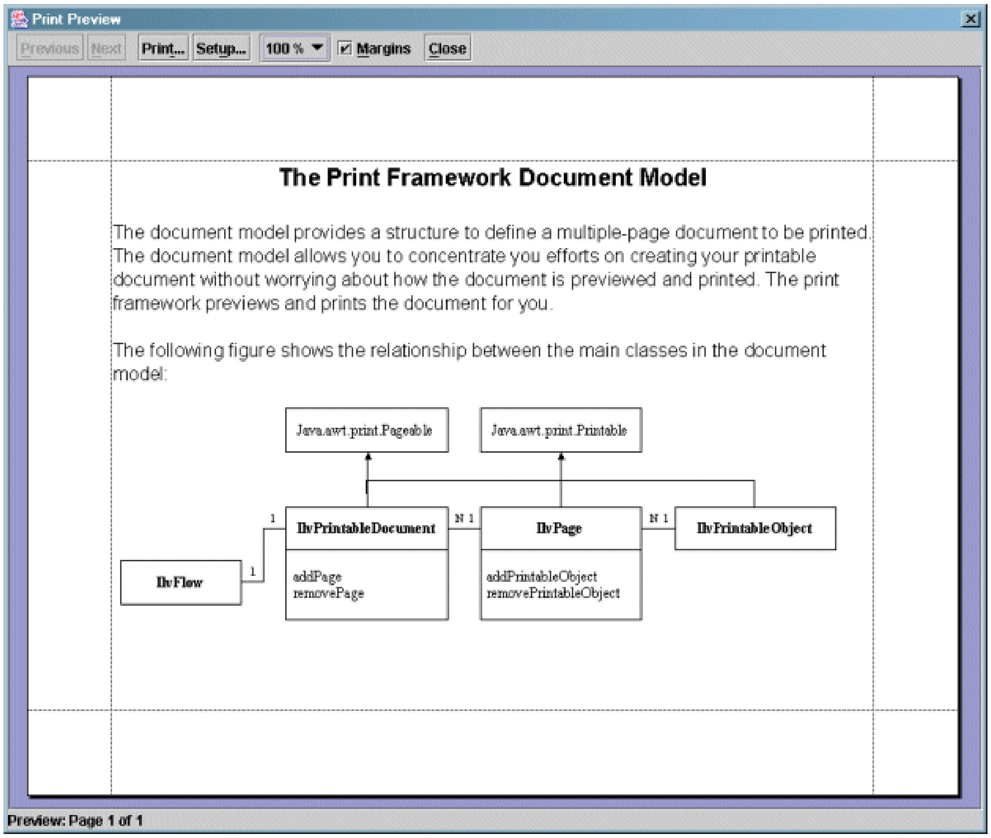
1. Create an `IlvPrintableDocument` object.
2. Create the `IlvFlow` object.
3. Add text and image to the `IlvFlow` object.
4. Set the text style and paragraph alignment.
5. Create and use the printing controller.
6. Use the user interface components to preview the document you created.

The example code is as follows.

Creating a document with a text flow

```
 // Creates the document.
 IlvPrintableDocument document
= new IlvPrintableDocument("A Test Document");

// Gets the flow to add text and images.
IlvFlow flow = document.getFlow();

// Sets the style of text and paragraph alignment
// for the title.
IlvFlow.TextStyle style = new IlvFlow.TextStyle();
style.setFont(new Font("Helvetica", Font.BOLD, 20));
style.setAlignment(IlvFlow.TextStyle.CENTER_ALIGNMENT);
flow.setTextStyle(style);

// Adds the title text.
flow.add("The Printing Framework Document Model");

flow.newLine();

// Sets the style of text for the next paragraph.
style.setAlignment(IlvFlow.TextStyle.LEFT_ALIGNMENT);
style.setFont(new Font("Helvetica", Font.PLAIN, 16));
flow.setTextStyle(style);

flow.newLine();
flow.add("The document model provides a structure to define a multiple-page
document to be printed. The document model allows you to concentrate your
efforts on creating your printable document without worrying about how the
document is previewed and printed. The printing framework previews and prints
 the document for you.");

flow.newLine();
flow.newLine();
flow.add("The following figure shows the relationship between the main classes
 in the document model:");
flow.newLine();

try {

    // Loads an image to add.
    Image image = IlvUtil.GetImageFromFile(Class.forName("demos.print.
PrintExample"), "model.gif");

    // Adds the image to the flow.
    flow.add(image, IlvFlow.TOP_ALIGNMENT);

  } catch (Exception e) {
}

// Creates the print controller.
IlvPrintingController controller = new IlvPrintingController(document);

// Previews the document.
controller.setPreviewMode(IlvPrintPreviewPanel.CONTINUOUS_MODE);
controller.printPreview(JOptionPane.getRootFrame());
```

Here is what you get on the screen:



The Print Framework Document Model

The document model provides a structure to define a multiple-page document to be printed. The document model allows you to concentrate you efforts on creating your printable document without worrying about how the document is previewed and printed. The print framework previews and prints the document for you.

The following figure shows the relationship between the main classes in the document model:

Document with text flow

Printing framework for manager content

Describes the printing framework used by JViews Framework for printing the content of managers.

In this section

Overview of the printing framework for managers

For details of the generic print framework, see *The generic printing framework*.

The Framework printing classes are located in the `ilog.views.print` package.

The architecture of this package is based on four main components:

- **A configuration document:** The document containing the printing configuration.
- **The manager:** The `IlvManager` to print.
- **The print UI:** Some specialized dialog boxes and interactors that allow you to configure printing properties such as the page format, the orientation, the header and footer, the number of pages, and so on.
- **The printing controller**: A high-level class that manages the document to print. You can use the printing controller to invoke the Print dialog box or the Page Setup dialog box, to preview the document, and to send the document to the selected printer.

There are three ways to print the contents of an `IlvManager` object in a document:

- *Printing the contents of a manager on multiple pages*

  You can print a manager (or an area of a manager) in multiple pages. Using the predefined document class `IlvManagerPrintableDocument`, you simply have to specify the manager object you want to print and some printing parameters such as the number of pages and the area to print.

- *Printing a manager as a flow of text*

  You can print a manager (or an area of a manager) in a flow of text.

- *Printing a manager in a custom document*

  You can print a manager (or an area of a manager) in a custom document structure. In this mode, you create your own document and insert a printable object that represents a manager in a page.

Printing the contents of a manager on multiple pages

Describes the classes in the `ilog.views.print` package that allow you to print the contents of an `IlvManager` object on multiple pages.

In this section

The IlvManagerPrintableDocument class

JViews Framework provides a document class, `IlvManagerPrintableDocument`, which is a subclass of the generic `IlvPrintableDocument` class. The `IlvManagerPrintableDocument` class is dedicated to printing the contents of a manager on multiple pages.

When using the `IlvManagerPrintableDocument` class, you do not have to create pages and add them to the document. This class will create the pages for you, depending on the parameters you specify for the document.

In addition to the generic parameters defined in the superclass `IlvPrintableDocument` such as the name and author of the document, the page format, the header and footer, and the page order, the `IlvManagerPrintableDocument` class allows you to specify the following options:

- ♦ The number of pages
- ♦ The area of the manager to be printed
- ♦ The zoom level used for printing

The following code creates an instance of `IlvManagerPrintableDocument` to print the area (0,0,500,500) of a manager in five columns:

```
IlvManagerPrintableDocument document = new IlvPrintableManagerDocument
                                       ("My Document", view);
document.setColumnCount(5);
document.setPrintArea(new IlvRect(0,0,500,500));
```

Number of pages

The number of pages is determined by the number of rows and columns that you specify as follows:

- ♦ If you specify the number of rows, the document computes the number of columns necessary to cover the area to print.
- ♦ If you specify both the number of rows and the number of columns, then the document class will choose to use the number of rows or the number of columns to produce the minimum number of pages.

To print the manager on one page, set the number of rows and the number of columns to 1.

Area to print

The area of the manager to print is specified by the `setPrintArea` and `getPrintArea` methods. When no print area has been specified, then the printed area will be the full area of the manager. To reset the area to print to the full area of the manager, call:

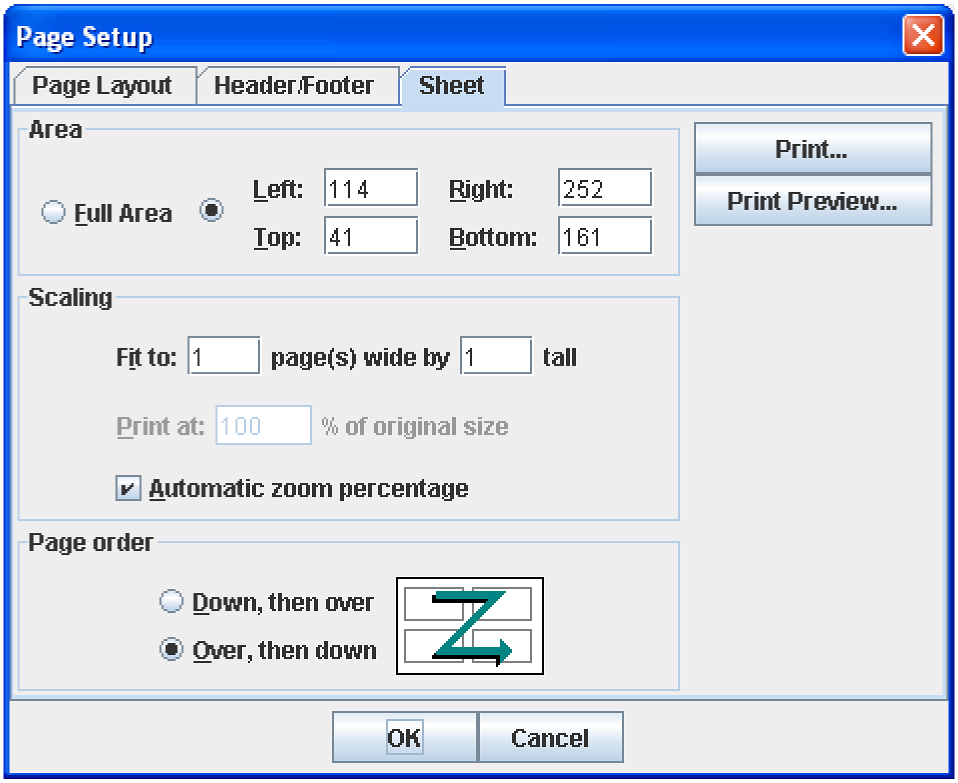
```
document.setPrintArea(null);
```

Zoom Level for Printing

The contents of the manager may be graphically different when a different zoom level is used, in particular when the manager contains nonzoomable objects. Thus, when printing the manager, you may need to specify the zoom level used for printing. By default, the contents of the manager are printed using the identity affine transform (that is, zoom level 1).

The IlvManagerDocumentSetupDialog class

All properties of the `IlvManagerPrintableDocument` class can be specified by a dialog box (class `IlvManagerDocumentSetupDialog`). This dialog box, a subclass of the generic `IlvDocumentSetupDialog`, contains an additional page in the tabbed pane that allows you to specify the area to print, the number of columns and rows (that is, the number of pages), the zoom level at which to print, and the page order for the numbering of pages.



Area

( ) Full Area (●) Left: 114 Right: 252 Top: 41 Bottom: 161

Scaling

Fit to: 1 page(s) wide by 1 tall

Print at: 100 % of original size

[✓] Automatic zoom percentage

Page order

( ) Down, then over

(●) Over, then down

Page Setup dialog box

You may not want to allow the user to change the zoom level, or you may need to specify a range of zoom level that is allowed for this specific manager. To do this, use the following methods in the `IlvManagerPrintableDocument` class:

- Enable or disable the modification of the zoom level from the dialog box:

  ```
  is/setZoomLevelModificationEnabled()
  ```

- Set the minimum or maximum zoom level that can be used for printing.

```
setMaximimumZoomLevel(double zl)
```

```
void setMinimumZoomLevel(double zl)
```

Note: You do not have to create this dialog box yourself; the printing controller will manage an instance of this class for you.

The IlvManagerPrintingController class

The `IlvManagerPrintingController` is a subclass of the generic `IlvPrintingController` that controls the printing of an `IlvManagerPrintableDocument` object.

After creating your `IlvManagerPrintableDocument` object, you must create an instance of `IlvManagerPrintingController` for the document. Then you will be able to perform various actions on the document to be printed; for example, you can:

♦ Call the `printDialog` method to invoke a dialog box to select a printer or to specify other printer-related information.

♦ Call the `setupDialog` method to open the Page Setup dialog box.

♦ Call the `printPreview` method to preview your document.

♦ Call the `print` method of the print controller to print your document. You can also click the Print button while you preview the document.

The IlvManagerPrintAreaInteractor class

The `ilog.views.print` package also contains a specific interactor (`IlvPrintAreaInteractor`) that allows you to specify the area to print on the manager by dragging a rectangle.

A Swing application that prints the contents of a manager

In this example, a full application using the IlvManagerPrintingController and IlvManagerPrintableDocument classes prints the contents of a manager on multiple pages.

This example has a Swing frame containing an instance of IlvManagerView that displays the contents of a manager. The application has also a menu bar with standard printing menu commands such as Print..., Print Preview, Page Setup..., and so on.

```
import java.awt.*;
import java.awt.print.*;
import java.awt.event.*;
import javax.swing.*;
import ilog.views.print.*;
import ilog.views.util.print.*;
import ilog.views.*;
import ilog.views.interactor.*;
import ilog.views.swing.*;

/**
 * This is a very simple example to show how to use
 * the IlvManagerPrintableDocument class.
 */
public class ExamplePrint extends JFrame
{
  /**
   * The manager to print.
   */
  IlvManager manager;

  /**
   * An IlvManagerView to display the content of a manager.
   */
  IlvManagerView mgrview;

  /**
   * The printing controller.
   */
  IlvManagerPrintingController controller;

  /**
   * The interactor that allows you to specify the area to print.
   */
  IlvPrintAreaInteractor printAreaInteractor;

  /**
   * Creates and initializes the example.
   */
  public ExamplePrint() {

    super("Printing Example");
    getContentPane().setLayout(new BorderLayout());
    // Creates the manager to print.
    manager = new IlvManager();

    // Fills the manager with some data.
    try {
       manager.read("data.ivl");
    } catch (Exception e) {
    }


    // Creates a view of the manager.
    mgrview  = new IlvManagerView(manager);
    mgrview.setBackground(Color.white);
    mgrview.setKeepingAspectRatio(true);

    // Creates the printing controller
    controller = new IlvManagerPrintingController(mgrview) ;

    // Initializes the document with some parameters.
    IlvPrintableDocument document = controller.getDocument();

    document.setName("data.ivl");
    document.setAuthor("My name");

    // Creates the interactor.
    printAreaInteractor = new IlvPrintAreaInteractor(controller);

    // Adds a toolbar to edit/zoom/pan.
    IlvJManagerViewControlBar toolbar
          = new IlvJManagerViewControlBar();
    toolbar.setView(mgrview);

    // Creates a scroll manager view.
    IlvJScrollManagerView scrollview
             = new IlvJScrollManagerView(mgrview);
    getContentPane().add(scrollview, BorderLayout.CENTER);
    getContentPane().add(toolbar, BorderLayout.NORTH);
    scrollview.setPreferredSize(new Dimension(300,300));
    setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);

    addWindowListener(new WindowAdapter () {
        public void windowClosed (WindowEvent e) {
          System.exit(0);
        }
    });

    // Creates the menu bar.
    setJMenuBar(createMenu());

  }

  private JMenuBar createMenu() {
    JMenuBar bar = new JMenuBar();
    JMenu file = new JMenu("File");
    JMenu parea = new JMenu("Print Area");
    JMenuItem preview = new JMenuItem("Print Preview...");
    JMenuItem setup = new JMenuItem("Page Setup...");
    JMenuItem setarea = new JMenuItem("Set Print Area");
    JMenuItem cleararea = new JMenuItem("Clear Print Area");
    JMenuItem print = new JMenuItem("Print...");


    // Action to open the print preview dialog.
    preview.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          controller.printPreview(ExamplePrint.this);
        }
      });

    // Action to open the setup dialog box.
    setup.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          controller.setupDialog(ExamplePrint.this, true, true);

        }
      });

    // Action to print the document.
    print.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          try {
            controller.print(true);
          } catch (Exception e) { }

        }
      });

    // Action to install the print area interactor.
    setarea.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          mgrview.setInteractor(printAreaInteractor);
        }
      });


    // Action to reset the print area to full manager size.
    cleararea.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {
          ((IlvManagerPrintableDocument)controller.
                getDocument()).setPrintArea(null);
        }
      });


    file.add(setup);
    file.add(parea);
    parea.add(setarea);
    parea.add(cleararea);
    file.add(preview);
    file.add(print);

    bar.add(file);

    return bar;

  }

public static void main(String[] args)
{
  SwingUtilities.invokeLater(
    new Runnable() {
      public void run() {
        ExamplePrint example = new ExamplePrint();
        example.pack();
        example.setVisible(true);
      }
    });
}
```

Printing a manager as a flow of text

The IlvManagerPrintableDocument class allows you to print the contents of a manager. You may also need to add a view of a manager as a flow of text. To do this, you use an instance of the IlvPrintableDocument class and the IlvFlow class described in *The IlvFlow class for creating a document with styled text*.

The IlvFlow class not only allows you to control the flow of text in a textual document, but also allows you to insert images in the text. The objects you can insert in the text are implemented by the interface IlvFlowObject.

The ilog.views.print package provides an implementation of the IlvFlowObject interface for printing the manager. The class is named IlvSimplePrintableManager.

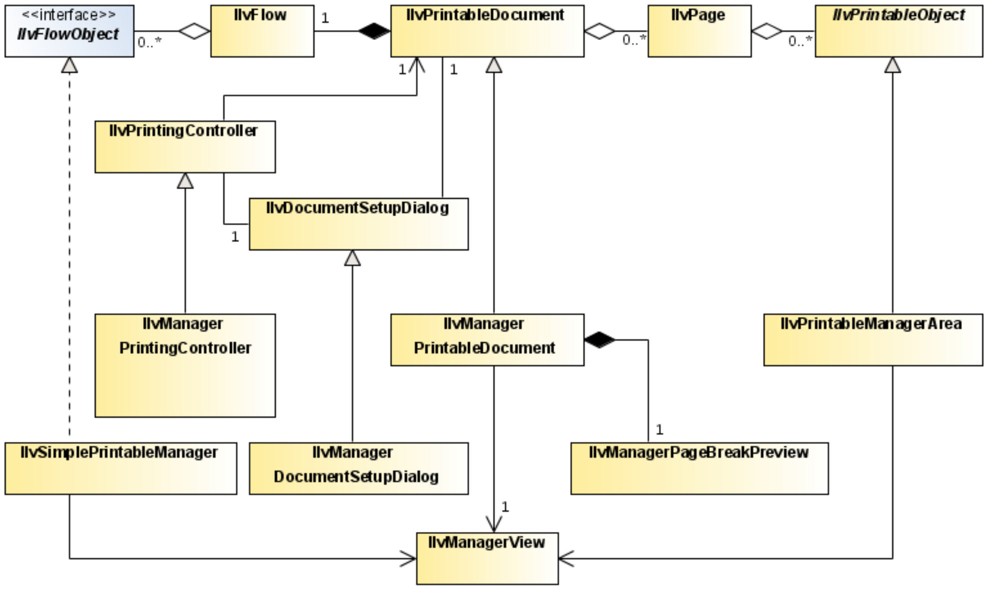
Printing a manager in a custom document

IBM® ILOG® JViews printing framework allows you to create your own document structure by creating pages (`IlvPage` class) and adding printable objects to those pages (`IlvPrintableObject` class). For more information see *The PrintableDocument class*.

The `ilog.views.print` package also provides a printable object (subclass of `IlvPrintableObject`) that you can insert in a page of your document to print an area of a manager. This class is named `IlvPrintableManagerArea`.

Class diagram for printing the contents of managers

The following UML class diagram summarizes all classes related to `IlvManagerPrintingController`. The printing controller contains an `IlvManagerPrintableDocument` that represents either the entire manager view or an area of the manager view to be printed. If an area of a document is to be printed, an `IlvPrintableManagerArea` represents the area of the manager printed on the page. If the manager view should be printed as a flow of text, an `IlvSimplePrintableManager` represents the view in the text flow.



The classes related to printing a manager

Scalable Vector Graphics

Explains how to configure IBM® ILOG® JViews to be able to read and write SVG files, how to use the SVG thin-client features of IBM® ILOG® JViews, and how to personalize IBM® ILOG® JViews SVG features for your context and to translate your own graphic objects.

In this section

Overview of SVG support

The SVG graphics format is based on the eXtensible Markup Language (XML) which gives it great interoperability.

SVG support in the JViews Framework class library

The JViews Framework class library provides the ability to load Scalable Vector Graphics (SVG) files into an `IlvManager` object. Conversely, the contents of an `IlvManager` can be translated into an SVG document and saved to a file.

These features allow you to interoperate with other SVG software such as SVG viewers or generators. When applicable, in the thin-client context, you can also replace bitmap generation by SVG generation to gain time and interactivity.

Uses of SVG Files

SVG can be used by IBM® ILOG® JViews applications as an exchange format for exchanging data with third-party software that supports this format. You will be able to import data from such software to the IBM® ILOG® JViews library. Conversely, you will be able to load SVG files generated by IBM® ILOG® JViews into third party software such as the Batik SVG Browser (Squiggle) from Apache™ , which you can download free at *http://xml.apache.org/batik*.

You can also think of SVG as a means of replacing bitmaps when generating data from a server and displaying the data in a Web browser. In this case, the browser must be able to display an SVG document. For the moment, the easiest solution is to use the Adobe® SVG Plug-in, which can be downloaded free at *http://www.adobe.com/svg*. You can find an example of such an SVG thin-client application in *Deploying IBM® ILOG® JViews applications as SVG thin clients*.

The contents of an SVG File

An SVG file describes a set of two-dimensional graphics. The following are examples of such graphics that could be found in an SVG file:

♦ Images, through the `image` element.

♦ Rectangles, through the `rect` element.

♦ Circles and ellipses, through the `circle` and `ellipse` elements.

♦ Lines, through the `line` and `polyline` elements.

♦ Polygons, through the `polygon` element.

♦ Arbitrary paths (curves, arcs, lines, and so on), through the `path` element.

♦ Groups of other graphic elements, through the `g` element.

♦ Text, through the `text` element.

These elements can be styled by setting XML presentation attributes on them or by linking cascading style sheets (CSS) to them.

To better understand SVG and its possibilities, see the SVG specification at the following URL: *http://www.w3.org/TR/SVG*. You will see that SVG provides many more features than those introduced here, such as transformations on graphic elements, filter effects, in-line animation, and scripting capabilities.

SVG file example

A typical example of an SVG file would be as follows.

```
<svg width="640" height="480">
  <defs>
    <!-- the style on path elements and element with id "myid" -->
    <!-- is defined through a style sheet -->
    <style type="text/css">
      path {stroke-width:3;stroke:blue;fill:none}
      .dash {stroke-dasharray:5 2}
      #myid {fill:rgb(205,5,5);fill-opacity:0.5}
    </style>
    <!-- style can be complex such as a gradient... -->
    <linearGradient id="grad" x1="0%" y1="0%" x2="100%"
         y2="100%">
       <stop offset="0" stop-color="yellow"/>
       <stop offset="0.2" stop-color="green"/>
       <stop offset="1" stop-color="red"/>
     </linearGradient>
  </defs>
  <!-- the style on the rectangle is defined through XML -->
  <!-- attributes                                        -->
  <rect x="0" y="0" width="100%" height="100%"
```

```
        fill="url(#grad)"/>
  <!-- paths use a particular syntax to defined their shape -->
  <path d="M0 0L640 480"/>
  <path class="dash" d="M640 0L0 480"/>
  <!-- the style on the ellipse is defined through an inline -->
  <!-- style sheet                                           -->
  <ellipse cx="320" cy="240" rx="40" ry="30"
       style="fill:rgb(180,10,10)"/>
  <circle id="myid" cx="320" cy="240" r="50"/>
</svg>
```

This SVG file will be rendered as a 640 x 480 rectangle filled with a linear green, yellow, and red gradient. On top on this gradient there will be two lines of thickness 3, one of which will be a dashed line. There will also be an ellipse and a circle; the color on the circle is semi-transparent (`fill-opacity:0.5`) and lets you see the color of the ellipse underneath.

Loading and saving SVG files with IBM® ILOG® JViews

In order to read and write SVG files, the JViews Framework library defines a new instance of `IlvStreamFactory` specialized for SVG.. You will see how to set the `SVGStreamFactory` on the `IlvManager` and how to use it.

To configure the SVG stream factory:

1. Create the manager you will work on and the stream factory

```
import ilog.views.*;
import ilog.views.svg.*;

IlvManager manager = new IlvManager();
SVGStreamFactory factory = new SVGStreamFactory();
manager.setStreamFactory(factory);
```

2. Set the options on the stream factory appropriately to change the way IBM® ILOG® JViews loads and saves SVG files: how the file size should be reduced, whether the reader should ignore some data, and so on.

3. After the stream factory is configured, it must be attached to the manager by using:

```
manager.setStreamfactory(factory);
```

More options can be found in the *Java API Reference Manual* for the class, such as other compaction techniques (remove invisible graphic objects, and so forth) or the ability to choose between CSS or XML styling.

The following example shows a stream factory configuration:

```
// When reading SVG, the parsing of the CSS style will include
// the definitions contained in the user.css file.
factory.getReaderConfigurator().setUserStyleSheetURL(“user.css”);
// When writing SVG, the following compaction techniques will be used:
//   - style on graphic element will be factored
//   - an algorithm will be applied to polyline to remove some points
factory.getBuilderConfigurator().
    setCompactMode(SVGStreamFactory.COMPACT_STYLE |
                   SVGStreamFactory.COMPACT_POLY);
// With the option set to true on the reader configurator, when reading SVG,
// non-processible elements will be memorized
// and with the option set to true on the builder configurator, this will
// allow regenerating them later.
factory.getReaderConfigurator().setFullDocumentOn(true);
factory.getBuilderConfigurator().setFullDocumentOn(true);
```

You can load an SVG file once a stream factory has been set up.

To load an SVG File:

- ♦ Call the `IlvManager` method to load an SVG file instead of a regular IBM® ILOG® JViews file.

The following code is an example of loading an SVG file.

```
try {
  manager.read("mysvgfile.svg");
} catch (ilog.views.io.IlvReadFileException rfe) {
  System.err.println("The SVG file is badly formatted");
} catch (java.io.IOException ioe) {
  System.err.println("Cannot access the SVG file");
}
```

Once an `IlvManager` object has been filled dynamically or by reading an SVG or an IVL file, it is possible to save the objects to an SVG file.

To save to an SVG file:

- ♦ Call the `IlvManager` method to save an SVG file instead of a regular IBM® ILOG® JViews file.

The following code is an example of writing to an SVG file:

```
try {
  manager.write("mysvgfilemodified.svg");
} catch (java.io.IOException ioe) {
  System.err.println("Cannot access the SVG file");
}
```

Deploying IBM® ILOG® JViews applications as SVG thin clients

Explains how to use the SVG generation mechanism of IBM® ILOG® JViews to deploy your JViews Framework applications as SVG thin clients.

In this section

Overview of the IBM® ILOG® JViews SVG thin-client feature

The IBM® ILOG® JViews SVG thin-client support, like the IBM® ILOG® JViews DHTML thin-client support (see LINK) is based on the Java™ Servlet technology.

SVG thin-client support

The IBM® ILOG® JViews SVG Framework thin-client support contains the following:

♦ An abstract servlet class than can generate SVG documents from an IBM® ILOG® JViews display

♦ A set of SVG scripts written in ECMAScript for use on the client side to display and interact with the document created on the server side

An IBM® ILOG® JViews SVG thin-client application provides the following features:

♦ Main View of the server side `IlvManager` object displayed in SVG with some predefined behavior: zooming, panning, single-line tooltips on top of graphics objects, fixed-size graphics management, and the availability of visibility filters with the possibility of load-on-demand for layers that are invisible at initialization time.

♦ Overview providing the ability to navigate on the Main View.

♦ Layer View allowing you to display the layers of the Main View and to switch their visibility on or off.

SVG thin-client sample

The sample code gives you additional information on how to code for SVG both server side and client side. For details, see ***<installdir>*** **`/jviews-framework86/samples/svg-servlet/index.html`**.

See the sample documentation for information on how to run the sample. The sample can be viewed inside a Web browser such as Internet Explorer® or Netscape Communicator with the Adobe® SVG Plug-in that can be downloaded for free at *http://www.adobe.com/svg*.

Developing the server side of an SVG thin-client application

The server side of an IBM® ILOG® JViews SVG thin-client application is composed of two main parts: the IBM® ILOG® JViews application itself, which can be any type of complex two-dimensional display built on top of the JViews Framework API, and a Servlet that produces SVG documents for the client.

In the example both server-side parts are written inside a single class defined in the file `SVGDynamicServlet.java` located at `samples/svg-servlet/src/svg/SVGDynamicServlet.java` in the installed product. For details, see ***<installdir>*** **`/ jviews-framework86/samples/svg-servlet/src/svg/SVGDynamicServlet.java`**.

The IBM® ILOG® JViews Application

The example IBM® ILOG® JViews application is a simplified version of a supplied sample. It is very simple and all described in the `getManager(HttpServletRequest)` method.

The application consists of an IBM® ILOG® JViews `IlvManager` object filled with the contents of an IBM® ILOG® JViews IVL file (`data/map.ivl`) as follows:

```
manager = new IlvManager();
try {
  // Read its contents from an IVL file.
  manager.read(getServletConfig().getServletContext().getResource("/data/map.
ivl"));
} catch (java.io.IOException e) {
} catch (IlvReadFileException rfe) {
}
```

The IBM® ILOG® JViews SVG Servlet

The SVG servlet inherits from the `IlvSVGManagerServlet` class which is in the `ilog.views.svg.servlet` package. The code for this class is as follows.

```
import javax.servlet.*;
import javax.servlet.http.*;

import ilog.views.*;
import ilog.views.io.IlvReadFileException;
import ilog.views.svg.SVGDocumentBuilder;
import ilog.views.svg.SVGDocumentBuilderConfigurator;
import ilog.views.svg.servlet.IlvSVGManagerServlet;

public class SVGDynamicServlet extends IlvSVGManagerServlet
{
  private IlvManager manager = null;

  private static final SVGDocumentBuilderConfigurator CONFIGURATOR =
    new SVGDocumentBuilderConfigurator();
  static {
    CONFIGURATOR.setCompactMode(SVGDocumentBuilderConfigurator.COMPACT_LOD);
    CONFIGURATOR.setViewBox(new IlvRect(-2200, 4600, 3600, 3600));
  }
```

```

  /**
   * Creates a manager servlet.
   */
  public SVGDynamicServlet()
  {
    super(CONFIGURATOR);
  }


  public IlvManager getManager(HttpServletRequest r)
  {
    if (manager == null) {
      manager = new IlvManager();
      try {
        // Read its contents from an IVL file.
       manager.read(getServletConfig().getServletContext().getResource("/data/
map.ivl"));
      } catch (java.io.IOException e) {
      } catch (IlvReadFileException rfe) {
      }
      } catch (IlvReadFileException rfe) {
      }
      manager.setVisible(manager.getManagerLayer("Rivers").getIndex(),
                         false, false);
      manager.getManagerLayer("Areas").
         addVisibilityFilter(new IlvZoomFactorVisibilityFilter(5,
            IlvZoomFactorVisibilityFilter.NO_LIMIT));
    }
    return manager;
  }
}
```

There is a sample SVG servlet provided at `samples/svg/SVGDynamicServlet.java` in the installed product. For details, see ***<installdir>*** **`/jviews-framework86/samples/svg-servlet/src/svg/SVGDynamicServlet.java`**.

The following describes the SVG servlet code in more detail:

♦ The `javax` `import` statements are required to use the Java Servlet API:

```

import javax.servlet.*;
import javax.servlet.http.*;
```

♦ The IBM® ILOG® `import` statements are required to use IBM® ILOG® JViews and the IBM® ILOG® JViews SVG thin-client support.

```

import ilog.views.*;
import ilog.views.io.IlvReadFileException;
import ilog.views.svg.SVGDocumentBuilder;
import ilog.views.svg.SVGDocumentBuilderConfigurator;
import ilog.views.svg.servlet.IlvSVGManagerServlet;
```

- The `IlvSVGManagerServlet` class is an abstract Java™ subclass of the `HTTPServlet` class from the Java Servlet API. The example `SVGDynamicServlet` class inherits from `IlvSVGManagerServlet` and defines only two methods: its constructor and the `getManager` method.
- The constructor initializes the base class by providing an `SVGDocumentBuilderConfigurator` instance to be used by IBM® ILOG® JViews classes to configure the way the SVG document is generated.

```
private static final SVGDocumentBuilderConfigurator CONFIGURATOR =
  new SVGDocumentBuilderConfigurator();
// ...
public SVGDynamicServlet()
{
   super(CONFIGURATOR);
}
```

- In this example the `SVGDocumentBuilderConfigurator` is configured not to send invisible layers of the JViews display to the client at generation time so that such layers are only loaded on demand. This is done through the following line:

```
CONFIGURATOR.setCompactMode(SVGDocumentBuilderConfigurator.COMPACT_LOD);
```

  The configurator is also configured to generate the SVG document with a particular `viewBox` in order to see a specific part of the IBM® ILOG® JViews display on the client. This is done through the following line:

```
CONFIGURATOR.setViewBox(new IlvRect(-2200, 4600, 3600, 3600));
```

- The `getManager` method is the only abstract method of the `IlvSVGManagerServlet` class and should return an `IlvManager` that will be used by the generated SVG document. Here we simply return the manager object we created after adding some visibility filters on it to allow some layers to not always be generated.

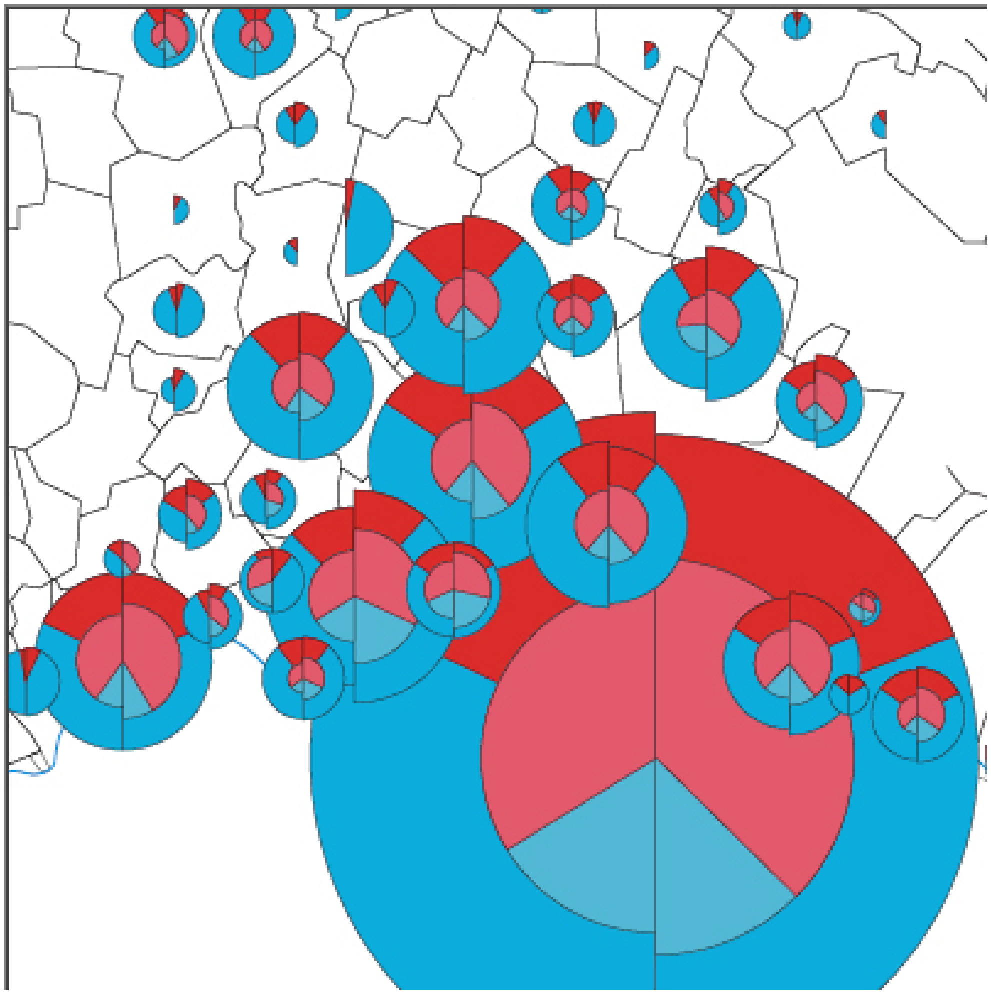
  The first line makes the `Rivers` layer invisible in the SVG generated document.

  The second line makes the `Areas` layer invisible if the zoom factor on the display is less than five times the initial zoom factor.

```
manager.setVisible(manager.getManagerLayer("Rivers").getIndex(),
                        false, false);
manager.getManagerLayer("Areas").
        addVisibilityFilter(new IlvZoomFactorVisibilityFilter(5,
           IlvZoomFactorVisibilityFilter.NO_LIMIT));
```

The example SVG servlet can answer HTTP requests from a client by sending SVG documents. If you have installed the example, you can try the following HTTP request:

This produces the following image:



This image is the (-2200, 4600, 3600, 3600) area of the JViews manager mapped to a 400, 400 image.

In most cases you do not need to know the servlet parameters because the SVG scripts provided by IBM® ILOG® JViews for the client side will set the HTTP request parameters for you.

After creating the server side, you can create the client side.

Developing the client side of an SVG thin-client application

The client-side display is an SVG file that will contain several things:

♦ SVG elements describing the structure of the client-side display

♦ JViews XML elements as metadata on the SVG elements describing how JViews client-side scripts should customize the SVG elements

♦ A call to an initialization function

♦ References to cascading style sheets to style the client-side display

SVG file

The example SVG file is a simplified version of the `index.svg` file of the supplied sample. The contents are as follows::

```
<?xml-stylesheet title="default JViews style sheet"
      href="default.css" type="text/css"?>
<?xml-stylesheet title="style sheet" href="style.css"
      type="text/css"?>
<svg width="640" height="480"
     xmlns:ilv="http://xmlns.ilog.com/JViews/SVGToolkit"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:a3="http://ns.adobe.com/AdobeSVGViewerExtensions/3.0/"
     a3:scriptImplementation="Adobe"
     onload="JViewsSVG.Init(evt)">
  <defs>
    <!-- alternatively you could import each single file
         as in the distribution example -->
    <script xlink:href="SVGUtil.es" language="text/ecmascript"
       a3:scriptImplementation="Adobe"/>
  </defs>
  <svg id="myView" x="0%" y="0%" width="100%" height="100%">
    <metadata>
      <ilv:view type="manager" enableTooltips="true"
                xlink:href="SVGDynamicServlet"/>
    </metadata>
  </svg>
  <svg id="overview" x="-70" y="-30" width="240" height="240">
    <title>Over View</title>
    <metadata>
      <ilv:view type="over" disableZoom="true"
                enableBackground="true" xlink:href="#myView"/>
    </metadata>
  </svg>
  <svg id="legend" x="0" y="240" width="240" height="240"
      viewBox="0 0 240 240">
    <title>The View Legend</title>
    <metadata>
      <ilv:view type="layer" disableZoom="true" enableDrag="true"
```

```
               showTitle="true" enableBackground=”true” xlink:href="#myView"/
>
    </metadata>
  </svg>
</svg>
```

SVG elements

The main `svg` element in the SVG file contains several SVG elements as follows:

```
 <svg id="myView" x="0%" y="0%" width="100%" height="100%">
    <metadata>
      <ilv:view type="manager" enableTooltips="true"
                xlink:href="SVGDynamicServlet"/>
    </metadata>
  </svg>
  <svg id="overview" x="-70" y="-30" width="240" height="240">
    <title>Over View</title>
    <metadata>
      <ilv:view type="over" disableZoom="true"
                enableBackground="true" xlink:href="#myView"/>
    </metadata>
  </svg>
  <svg id="legend" x="0" y="240" width="240" height="240"
      viewBox="0 0 240 240">
    <title>The View Legend</title>
    <metadata>
      <ilv:view type="layer" disableZoom="true" enableDrag="true"
               showTitle="true" enableBackground=”true” xlink:href="#myView"/
>
    </metadata>
  </svg>
```

SVG element attributes

Each `svg` element corresponds to a view that will be displayed on the client. The `x`, `y`, `width`, and `height` attributes on the `svg` elements correspond to the rectangular region of the SVG client area in which the view will be displayed. Each view has an SVG `metadata` element that describes through the `ilv:view` element how the JViews SVG client-side scripting will display that view.

COMMENT: Make this a description list

The following attributes are allowed on the `ilv:view` element:

- `type`: the possible values are `manager` (the view will display the main manager view), `overview` (the view will display an overview) and `layer` (the view will display a view of the layers).
- `disableZoom`: if `true`, the view will remain fixed in size when the user performs a zoom operation. The default value is `false`.

♦ enableBackground: if true, a background will be displayed under the view. The default value is false.

♦ enableDrag: if true, the user will be able to move the view by dragging it. The default value is false.

In addition to these attributes, depending on the value of the type attribute, some additional attributes may be recognized. These are shown in *Additional SVG element attributes*

Additional SVG element attributes

Attribute	Recognized with the Following Types	Meaning	Default Value
showTitle	Layer View	If true, displays as a title of the View the contents of the 'title' element child of the 'svg' element.	False
disableClip	Manager View	If true, does not clip the contents of the manager view to the bounds of the 'svg' element.	False
xlink:href	Layer and OverView	Provides a reference to the Manager View by using its ID prefixed with a hash sign (#). For the OverView, the reference can point to a Manager View that belongs to another SVG file, provided that both SVG files are references from the same HTML file. The reference is then of the form: idofthemainSVGfile#idoftheManagerView This feature works only with Adobe® Viewer on Microsoft® Internet Explorer®.	None. Mandatory.
xlink:href	Manager View	Provides a reference to the Servlet that will provide the contents of the Manager View.	None. Optional. The contents can be inlined if needed.

The main SVG element

The svg elements and their metadata need to be interpreted by the client JViews SVG scripts. This is done by calling the JViewsSVG.Init() method at loading time on the main svg element. The main svg element is as follows:

```
<svg width="640" height="480"
     xmlns:ilv="http://xmlns.ilog.com/JViews/SVGToolkit"
     xmlns:xlink="http://www.w3.org/1999/xlink"
     xmlns:a3="http://ns.adobe.com/AdobeSVGViewerExtensions/3.0/"
     a3:scriptImplementation="Adobe"
     onload="JViewsSVG.Init(evt)">
  <defs>
    <!-- alternatively you could import each single file... -->
    <script xlink:href="SVGFramework.es" language="text/ecmascript"
       a3:scriptImplementation="Adobe"/>
  </defs>
</svg>
```

Main SVG element attributes

The main `svg` element contains several attributes:

- The `width` and `height` attributes correspond to the size the SVG document will take in the Web browser. These can be either absolute values or percentage values. If you use percentage values, the size of the image will be rescaled when the browser size is resized.
- The `xmlns:ilv` attribute allows you to import the IBM® ILOG® JViews namespace to use IBM® ILOG® JViews XML elements and attributes.
- The `xmlns:xlink` attribute allows you to import the XLink namespace to use XLink.
- The `xmlns:a3` attribute allows you to import the Adobe® SVG Viewer namespace to use its proprietary functionality.
- The `onload` attribute references the `JViewsSVG.Init(evt)` method to allow the JViews SVG scripts to run on the client.

Script files in the SVG file

The main `svg` element has a reference to the script code as a child of the `defs` element. This reference can either be to the concatenated version (`SVGFramework.es`) or to the single files needed by your application. The scripting files can be found in *<installdir>* `/jviews-framework86/lib/thinclient/svg/`.

Script files needed for SVG thin client shows a summary of the script files you need depending on the SVG thin-client features you are using.

Script files needed for SVG thin client

Feature	ECMAScript Files Needed
Any	`SVGUtil.es`, `SVGAbstractView.es`
Main View	`SVGTooltipManager.es`, `SVGLayer.es`, `SVGView.es`
Over View	`SVGOverview.es`
Layer View	`SVGTitledView.es`, `SVGCheckBox.es`, `SVGLayerView.es`
External Overview	`SVGExternalOverview.es`

Cascading style sheets in the SVG file

The first two lines of the SVG file are:

```
<?xml-stylesheet title="default JViews style sheet"
      href="default.css" type="text/css"?>
```

```
<?xml-stylesheet title="style sheet" href="style.css"
      type="text/css"?>
```

These are processing instructions that refer to cascading style sheets (CSS) to give the client-side display its look. The first style sheet referred to is the default one; it is mandatory to put this processing instruction in your SVG file. After this instruction, you can add your own style sheet (`style.css` in this case) to customize the default styling; the second style sheet will be cascaded with the first one according to CSS rules.

The style.css file

The contents of the `style.css` file are the following:

```
#legend > .backgroundRect {stroke:red;fill-opacity:0.6}
#overview > .backgroundRect {fill-opacity:1}
.overRect {fill:yellow;fill-opacity:0.8}
.title {fill-opacity:0.8}
```

This means that:

- The background rectangle (the SVG elements with the `backgroundRect` CSS class) of elements that are children of the SVG element with the `legend` ID have to be stroked with the red color and filled with an opacity of 60%.
- The background rectangle of elements that are children of the SVG element with the `overview` id have to be filled with full opacity.
- The overview rectangle (the SVG elements with the `overRect` CSS class) have to be filled with the yellow color and an 80% opacity.
- The title areas (the SVG elements with the title CSS class) have to be filled with an 80% opacity.

The following are the CSS classes that are recognized by the JViews SVG thin client and the display element to which they correspond:

CSS classes and corresponding display elements

CSS Class	Corresponds to
`backgroundRect`	Background Rectangle of a View (main view, overview, layer view).
`overRect`	OverView Rectangle.
`tooltip`	Tooltip Rectangle on a JViews Graphics Object.
`tooltipText`	Tooltip Text on a JViews Graphics Object.
`title`	Title Rectangle of a View.
`titleText`	Title Text of a View.
`enableCheckBox`	A Check Box in the enabled state.
`disableCheckBox`	A Check Box in the disabled state.

For more information about CSS in general, see the CSS specification at *http://www.w3.org/TR/REC-CSS2* and, more specifically for CSS in SVG, see the SVG 1.1 specification at *http://www.w3.org/TR/SVG11*.

Deployment of the thin-client application

Once the SVG file (client side) is deployed on the server with the `SVGDynamicServlet` (server side), a user on the client will be able to visualize the contents of the manager on the client as an SVG document and to interact with this document. The user will be able to zoom, pan, display the tooltips on the JViews Graphics Objects, and change layer visibilities.

By extending the servlet, the JViews SVG generator, or the client-side SVG file, you can add your own features to the SVG thin client example.

SVG advanced personalization

Describes the default conversion of graphic objects to SVG and explains how to define the way user-defined graphic objects are converted to SVG..

In this section

Overview of conversion to SVG

The IBM® ILOG® JViews SVG generator uses a generic translator (`GenericGraphicTranslator`) by default to convert graphic objects, including those that you have defined, to SVG.. It is not mandatory to define the way your objects are translated to SVG, but you may want to for reasons such as custom interactions. To do this, you will need to write some additional code. This may also be necessary if you want to redefine the way a standard IBM® ILOG® JViews graphic object (`IlvGraphic`) is translated.

Customizing the conversion of a graphic object

Subclassing a graphic object

Assume your graphic object is a subclass of a rectangle that draws a blue Bezier curve into a rectangle from the left bottom point to the right top point. The code is as follows:

```
package mypackage;

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

public class MyGraphic extends IlvRectangle
{
  private IlvPoint[] pts = new IlvPoint[4];
  private float[] dash = {4, 2};

  public MyGraphic(IlvRect rect)
  {
     super(rect);
     for (int i = 0; i < 4; i++)
        pts[i] = new IlvPoint();
     computeBezier();
  }

  private computeBezier()
  {
    pts[0].x = drawrect.x;
    pts[0].y = drawrect.y + drawrect.height;
    pts[1].x = drawrect.x + drawrect.width / 4;
    pts[1].y = drawrect.y + drawrect.height / 4;
    pts[2].x = drawrect.x + 3*drawrect.width / 4;
    pts[2].y = drawrect.y + 3*drawrect.height / 4;
    pts[3].x = drawrect.x + drawrect.width;
    pts[3].y = drawrect.y;
  }

  public void draw(Graphics g, IlvTransformer t)
  {
     super.draw(g, t); // will draw the rectangle
     // Will draw a Bezier in blue
     g.setColor(Color.blue);
     IlvGraphicUtil.DrawBezier(g, pts, 4, 1,
                               IlvStroke.JOIN_MITER, IlvStroke.CAP_ROUND,
                               dash, t);
  }

  public void applyTransform(IlvTransform t)
  {
     super.applyTransform(t);
    computeBezier();
  }

  // ...
}
```

Creating a translator and setting the builder configurator

If you do not want your object to use the generic translation mechanism, you should implement the `SVGDocumentBuilderConfigurator.GraphicTranslator` interface to be able to translate `MyGraphic` instances to SVG `Element` instances.

The `translate` method of your class should build a new `Element` instance, and, after filling it with the information contained in the graphic object and applying a style, should return that instance. To fill the `Element`, you can use regular DOM methods (`Element.setAttribute()`) or the methods provided by the SVG DOM. Note, however, that only limited support of SVG DOM is implemented in IBM® ILOG® JViews, and not all the methods are accessible.

The following code shows how to create the translator.

```
import org.w3c.dom.*;
import org.w3c.svg.dom.*;

public class MyGraphicTranslator
       implements SVGDocumentBuilderConfigurator.GraphicTranslator
{
  // Method that translates the graphic to SVG
  public Element translate(IlvGraphic graphic, IlvTransformer t,
                           SVGDocumentBuilder builder)
  {
    SVGDocument doc = builder.getDocument();

    // The SVG 'group' element will contain
    // all drawings necessary to display MyGraphic.
    SVGGElement group = (SVGGElement)doc.createElementNS
                                    ("http://www.w3.org/2000/svg", "g");

    // First add to the group the element corresponding
    // to the parent class of MyRect (IlvRectangle).
    group.appendChild(builder.getConfigurator().
       getTranslator("ilog.views.graphic.IlvRectangle").
       translate(graphic, t));

    // Create the SVG element corresponding to
    // the drawing added at MyGraphic level.
    SVGPathElement path = (SVGPathElement)doc.createElementNS
                                   ("http://www.w3.org/2000/svg", "path")
;
    IlvRect rect = graphic.boundingBox(graphic, t);
    SVGList list = path.getPathSegList();
    // Go to the beginning point of the Bezier.
    list.appendItem(path.createPathSegMovetoAbs(rect.x,
                                            rect.y + rect.height));
    // Add the Bezier.
    list.appendItem(path.
      createPathSegCurvetoCubicAbs(rect.x + rect.width,
                                   rect.y,
                                   rect.x + rect.width / 4,
                                   rect.y + rect.height / 4,
                                   rect.x + 3*rect.width / 4,
                                   rect.y + 3*rect.height / 4));
```

```


    // Start to style the path.
    builder.startStyleElement(path, null);
    // Really apply the style (blue color...).
    builder.appendStyle("stroke", "blue");
    builder.appendStyle("stroke-width", "1");
    builder.appendStyle("stroke-join", "miter");
    builder.appendStyle("stroke-cap", "round");
    builder.appendStyle("stroke-dasharray", "4 2");
    // Finish to style the path.
    builder.endStylingElement();
    group.appendChild(path);

    return group;
  }
}
```

Customizing the SVG stream factory

You should create a well-configured builder configurator that refers to the new translator. If you are using an `SVGOutputStream` you can do the following:

```
SVGStreamFactory factory = new SVGStreamFactory();
factory.getBuilderConfigurator().putTranslator("mypackage.MyGraphic", new
 MyGraphicTranslator());
```

Do not forget to set this new stream factory on your manager instead of the regular one.

If you are in SVG thin-client context, you just have to call `putTranslator(java.lang.String, ilog.views.svg.SVGDocumentBuilderConfigurator.GraphicTranslator)` method on the `SVGBuilderConfigurator` instance you used for your Servlet.

Customizing the SVG DOM generated by the SVG thin client

In the SVG thin-client context, in addition to defining your own way to export `IlvGraphic` instances to SVG, you may want to modify the SVG DOM instance that will be sent to the client beforehand. To do this, you need to redefine the `generateSVGDocument` method of the `IlvSVGManagerServlet` class.

For example, if you need to put an SVG drop-shadow effect arrow on a particular object, the code is as follows:

```
protected Document generateSVGDocument(HttpServletRequest request,
                                       int width, int height,
                                       String[] requestedLayers)
   throws ServletException
  {
    Document document = super.generateSVGDocument(request,
                                                  width, height,
                                                  requestedLayers);
    // create the drop-shadow filter effect
    Element filter = document.createElementNS(null, "filter");
    filter.setAttribute("id", "drop");
    filter.setAttribute("filterUnits", "objectBoundingBox");
    filter.setAttribute("x", "-0.1");
    filter.setAttribute("y", "-0.1");
    filter.setAttribute("width", "1.2");
    filter.setAttribute("height", "1.2");
    Element blur = document.createElementNS(null, "feGaussianBlur");
    blur.setAttribute("in", "SourceAlpha");
    blur.setAttribute("stdDeviation", "2");
    blur.setAttribute("result", "balpha");
    Element offset = document.createElementNS(null, "feOffset");
    offset.setAttribute("in", "balpha");
    offset.setAttribute("dx", "4");
    offset.setAttribute("dy", "4");
    offset.setAttribute("result", "oba");
    Element merge = document.createElementNS(null, "feMerge");
    Element node = document.createElementNS(null, "feMergeNode");
    node.setAttribute("in", "oba");
    merge.appendChild(node);
    node = document.createElementNS(null, "feMergeNode");
    node.setAttribute("in", "SourceGraphic");
    merge.appendChild(node);
    filter.appendChild(blur);
    filter.appendChild(offset);
    filter.appendChild(merge);
    // add the drop-shadow filter effect to the document
    document.getDocumentElement().appendChild(filter);
    // set the effect on the "myGraphic" element if it is in one
    // of the generated layers:
    Element elt = document.getElementById("myGraphic");
    if (elt != null)
      elt.setAttribute("filter", "url(#drop)");
    // return the modified document
    return document;
  }
```

SVG features supported when reading an SVG file

To help you build SVG files that will be fully understood by the SVG reader of IBM® ILOG® JViews, use the following tables to see which SVG elements and CSS properties are supported/unsupported.

Supported SVG elements

Element name	Attributes not supported on this element
`a`	`xlink:role`, `xlink:acrole`, `xlink:actuate`
`circle`	
`clipPath`	`clipPathUnits`
`defs`	
`desc`	
`ellipse`	
`g`	
`image`	
`line`	
`linearGradient`	
`metadata`	
`path`	
`pattern`	`patternUnits`, `patternTransform`
`polygon`	
`polyline`	
`radialGradient`	
`rect`	
`stop`	
`style`	`!important` rules are not supported
`svg`	`zoomAndPan`
`switch`	
`symbol`	`refX`, `refY`, `viewBox`, `preserveAspectRatio`
`text`	`textLength, lengthAdjust`
`textPath`	`textLength, lengthAdjust, startOffset, method, spacing`
`title`	
`tspan`	multiple values `x, y, dx, dy` (single values supported), `rotate, textLength`
`use`	

Supported CSS properties

Property name	Remark
`clip-path`	URI local to the file only.

Property name	Remark
`color`	
color-interpolation	Supported on `linearGradient` and `radialGradient`.
`fill`	URI local to the file only, ICC colors are not supported.
`fill-opacity`	
`fill-rule`	
`font-family`	
`font-size`	Relative identifiers are not supported.
`font-stretch`	Relative identifiers are not supported.
`font-style`	
`font-weight`	Relative identifiers are not supported.
`opacity`	Supported on basic shapes, paths, and `g` (group) elements.
`stop-color`	ICC colors are not supported.
`stop-opacity`	
`stroke`	URI local to the file only, ICC colors are not supported.
`stroke-dasharray`	
`stroke-dashoffset`	
`stroke-linecap`	
`stroke-linejoin`	
`stroke-miterlimit`	
`stroke-opacity`	
`stroke-width`	
`text-anchor`	
`visibility`	

DHTML thin-client support in JViews Framework

Describes the support for thin-client applications in JViews Framework.

In this section

Generating a client-side image map
Describes how to generate an image map on the client side.

The IlvManagerServlet class
Describes the predefined servlet and how to use it.

The IlvManagerServletSupport class
Describes how to add thin-client support to a servlet.

Controlling tiling
Describes how to control tiling on the client side and the server side.

Overview of thin-client support

The IBM® ILOG® JViews class library can be used on the client side where you develop Java™ applets or applications. It can also be used on the server side. Some Web browser applications require that the client stay very light, with most of the functionality residing in the server. The thin-client support in IBM® ILOG® JViews Framework allows you to create such applications easily. You can use the power of the IBM® ILOG® JViews class library to build complex two-dimensional representations on the Web server and use the Dynamic HTML thin-client support of your Web browser to display and interact with the images created by the server.

IBM® ILOG® JViews thin-client Web architecture

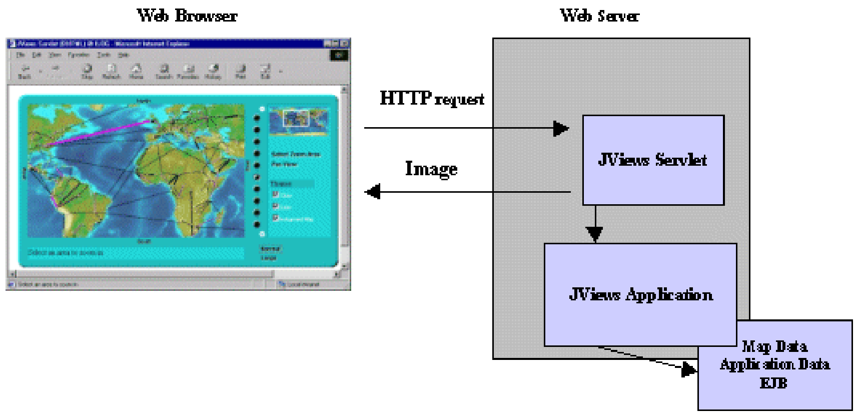
The IBM® ILOG® JViews thin-client support is based on the Java™ servlet technology. Servlets are Java programs that run on a web server. They act as a middle layer between HTTP requests coming from a Web browser or other HTTP clients such as applets or applications and the application or databases on the web server. The job of the servlet is to read and interpret HTTP requests coming from an HTTP client program and to generate a resulting document that in most cases is an HTML page.

For more information about servlet technology, you can visit the JavaSoft™ site *http:// java.sun.com/products/servlet*.

You will also find their information about the web servers supporting Java servlets.

For the predefined types of IBM® ILOG® JViews clients, the content created by the servlet is primarily a JPEG image. On the client side, user interactions with the image are managed by code in Dynamic HTML scripts.

Creating a web application with IBM® ILOG® JViews consists of using the IBM® ILOG® JViews library on the server side to create complex two-dimensional displays based on application data that resides on the server. A servlet will answer HTTP requests from a client and deliver images to this client, as illustrated in the following figure.



Client-Server Display Interaction

IBM® ILOG® JViews Framework thin-client support contains the following:

♦ An abstract servlet class that can generate JPEG images from an IBM® ILOG® JViews display.

♦ A set of Dynamic HTML scripts written in JavaScript™ that will be used on the client side to display and interact with the image created on the server side.

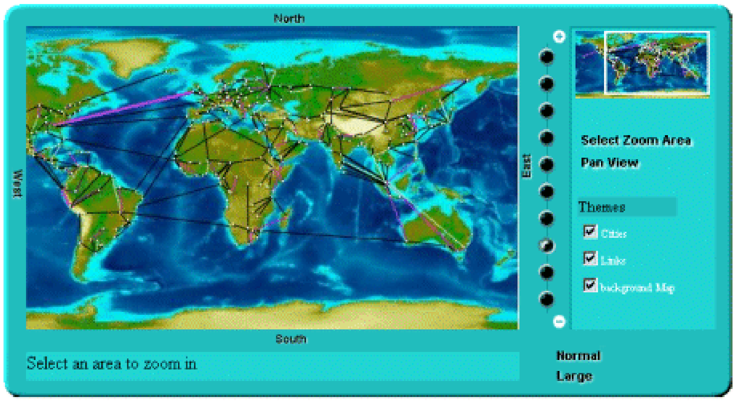
Creating an IBM® ILOG® JViews thin-client application consists of developing the server side and developing the client side.

Getting started with the IBM® ILOG® JViews thin client

The XML Grapher example shows how to build the server side and also how to create a Dynamic HTML client.

The XML Grapher example is available at ***<installdir>*** **`/jviews-framework8.6/samples/xmlgrapher`**.

This example allows you to display a network of interconnected cities on top of the map in a thin-client context.



The XML Grapher Example

The XML Grapher example is composed of the following pieces:

♦ An IBM® ILOG® JViews component that can read an XML file describing a set of interconnected cities and display them on top of a map as shown in the picture above.

  This component is located in the following files:

  <installdir> **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/XmlGrapher.java`**

  <installdir> **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/GrapherNode.java`**

♦ Some example XML files for the component, located in *<installdir>* / `jviews-framework86/samples/xmlgrapher/webpages/data`

♦ A servlet that can produce JPEG images from the component described above.

  The servlet is located in:

  <installdir> **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/servlet/XmlGrapherServlet.java`**

♦ A Dynamic HTML client composed of:

  - The HTML starting page: ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/webpages/dhtml/index.html`**

- The set of JavaScript™ Dynamic HTML components, located in: ***<installdir>*** **`/ jviews-framework86/lib/thinclient/javascript`**
- Some images required for the example, located in: ***<installdir>*** **`/ jviews-framework86/samples/xmlgrapher/webpages/dhtml/images`**

Installing and running the XML Grapher example

This sample is compatible with the browsers and browser versions listed in the Release notes under *Requirements for running thin-client applications*. The example contains a WAR (Web ARchive) file that allows you to install the example easily on any server that supports the Servlet API 2.1 or later.

For your convenience, the WAR file has already been installed for you on the Apache Tomcat™ Web server that is supplied with the IBM® ILOG® JViews installation. Tomcat is the official reference implementation of the Servlet and JSP™ specifications. If you are already using an up-to-date Web or application server, there is a good chance that it already has everything you need. You can check the latest list of servers that support servlets at: *http://java.sun.com/products/servlet/industry.html*.

To be able to run, this example requires a Web server and a Web browser that supports Dynamic HTML (for the DHTML client).

To run the example on the TOMCAT web server supplied with the IBM® ILOG® JViews installation:

1. Set the JAVA_HOME environment variable to point to your Java™ Platform, Standard Edition installation.
2. Go to the TOMCAT `bin` directory located in

   ```
   <installdir>/jviews-framework86/tools/apache-tomcat-6.0.14/bin
   ```

3. Depending on your system, run the `startup.bat` or `startup.sh` script to run the Apache Tomcat™ server.
4. To see the example, launch a Web browser and open the page:

   http://localhost:8080/xmlgrapher/index.html

Note: You must use `localhost` instead of the name of your machine. Otherwise, the sample applet may not be able to connect to the servlet.

The Web page gives you access to two different clients: a Dynamic HTML client and a thin Java client.

The IBM® ILOG® JViews servlets can run with the headless support that is built-in since Java SE 1.4, without an X server. For more information on this feature, refer to the Java SE *Release Notes*.

Developing the server

The server side of an IBM® ILOG® JViews thin-client application is composed of two main parts: the IBM® ILOG® JViews application itself, which can be any type of complex two-dimensional display built on top of the IBM® ILOG® JViews API, and a Servlet that produces JPEG images to the client.

The way the server side is built in the XML Grapher example helps in analyzing these parts.

The XML Grapher server

In the XML Grapher example, a graph of nodes and links is displayed on top of a map. This IBM® ILOG® JViews application is defined in the file `XmlGrapher.java`, located in ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/servlet/XmlGrapherServlet.java`**

Note: This part of the example contains only standard IBM® ILOG® JViews code and is therefore not explained in detail. You will only see how the class is used to create the example. The application on the server side really depends on the type of information you want to display anyway.

The XmlGrapher class

The `XmlGrapher` class is a simple subclass of the IBM® ILOG® JViews `IlvManagerView` class.

The main functionality of this small component is to read an XML file describing nodes and links and to create an IBM® ILOG® JViews grapher that represents those nodes and links on top of a map. This is done in the method:

```
public void setNetwork(URL url)
```

The XML file contains information on the map and the bitmap file of the map. It contains a list of nodes, including the position, or location, of each node and information on links. In the example, the position, or location, is described by using x-y coordinates. In a real mapping application, the IBM® ILOG® JViews Maps API allows you to use geographical projections.

The `setNetwork` method parses the XML file, creates the map, and places the nodes and the links on top of the map. It also applies an orthogonal link layout algorithm to lay out the links automatically.

You can look at an XML example file in *<install-dir>* `/jviews-framework86/samples/xmlgrapher/webpages/data`.

The servlet

Once the application is built, you need to create a servlet that produces images of the application to a client. IBM® ILOG® JViews Framework provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvManagerServlet`. This class can be found in the package `ilog.views.servlet`.

The servlet created for the XML Grapher example is very simple. To understand in depth how the servlet works, read *The IlvManagerServlet class*. The servlet for the XML Grapher example is located in the file: ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/src/xmlgrapher/servlet/XmlGrapherServlet.java`** .

```
import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;

import ilog.views.*;
import ilog.views.servlet.*;

import demo.xmlgrapher.*;

public class XmlGrapherServlet extends IlvManagerServlet
{
  private XmlGrapher xmlGrapher;

  /**
   * Initializes the servlet.
   */
  public void init(ServletConfig config) throws ServletException
  {
    super.init(config);
    xmlGrapher = new XmlGrapher();
    String xmlfile = config.getInitParameter("xmlfile");

    if (xmlfile == null) {
      xmlfile = config.getServletContext().getRealPath("/data/world.xml");
      xmlfile = "file:" + xmlfile;
    }
    try {
      xmlGrapher.setNetwork(new URL(xmlfile));
    } catch (MalformedURLException ex) {
    }
    setVerbose(true);
  }

  public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    return xmlGrapher;
  }

  protected float getMaxZoomLevel(HttpServletRequest request,
                                     IlvManagerView view)
  {
    return 30;
  }
}
```

The `import` statements:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

are required to use the Java Servlet API.

The `import` statements:

```
import ilog.views.*;
import ilog.views.servlet.*;
```

are required for using IBM® ILOG® JViews and the IBM® ILOG® JViews servlet support.

The `import` statement:

```
import demo.xmlgrapher.*;
```

is required for the XML Grapher class.

The `IlvManagerServlet`. class is an abstract Java™ class subclass of the `HTTPServlet` class from the Java servlet API. The `XmlGrapherServlet` inherits from the `IlvManagerServlet` class and defines only three methods.

The init method

This method initializes the servlet by creating an `XmlGrapher` object:

```
public void init(ServletConfig config) throws ServletException
{
    xmlGrapher = new XmlGrapher();
    ...
```

Then an XML file is read by the `XmlGrapher` object using the `setNetwork` method:

```
String xmlfile = config.getInitParameter("xmlfile");
if (xmlfile == null)
  xmlfile
       = config.getServletContext().
              getRealPath("/data/world.xml");

try {
  xmlGrapher.setNetwork(new URL("file:" + xmlfile));
} catch (MalformedURLException ex) {
}
```

The XML file can be specified in the configuration of the servlet. By default, the file `world.xml` is used.

The getManagerView method

The **getManagerView** method is the only abstract method of the `IlvManagerServlet` class and should return an `IlvManagerView` that will be used to generate the image. Here the `XmlGrapher` object is returned.

```
public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    return xmlGrapher;
  }
```

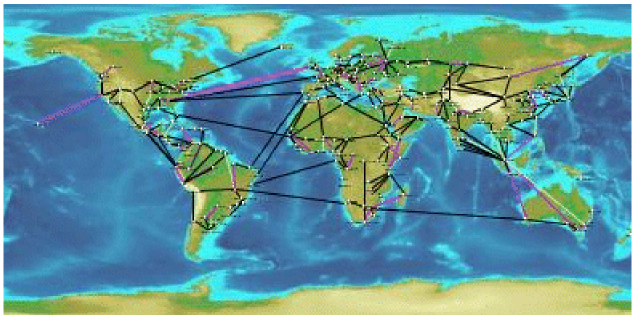
The getMaxZoomLevel Method

This method allows you to fix the user’s maximum zoom level on the client side. Here we overwrite the method to return a larger value.

As you have seen, creating the servlet is very simple. This servlet can now answer HTTP requests from a client by sending JPEG images. If you have installed the example, you can try the following HTTP request:

```
http://localhost:8080/xmlgrapher/
demo.xmlgrapher.servlet.XmlGrapherServlet?request=image
            &format=JPEG&bbox=0,0,512,512
            &width=400
            &height=200
            &layer=Cities,Links,background%20Map
```

This produces the following image:



Generated Bitmap Image

This request asks the servlet named `demo.xmlgrapher.servlet.XmlGrapherServlet` to produce an image of size 400 x 200 showing the area (0, 0, 512, 512) of the manager with the layers “Cities,” “Links,” and “Background Map” visible.

In most cases, you do not have to know the servlet parameters because the Dynamic HTML objects or the Java™ classes provided by IBM® ILOG® JViews for the client side will take care of the HTTP requests for you.

This example is a very simple servlet. This servlet uses the same `IlvManagerView` instance for all clients; this means that every client will see the same data. For more complex usage of the `IlvManagerServlet` classes, read *The IlvManagerServlet class*.

Developing the client

Describes the client side of a thin-client application and how to develop a dynamic HTML client by adding JavaScript™ components.

In this section

The IlvPanTool JavaScript component
Describes the `IlvPanTool` component.

The IlvMapInteractor and IlvMapRectInteractor JavaScript components
Describes the `IlvMapInteractor` and `IlvMapRectInteractor` components.

The Popup menu in JavaScript
Describes the JavaScript component for the popup menu.

Overview of client-side development

After creating the server (see *Developing the server*), you can create the client side. The IBM® ILOG® JViews thin-client support allows you to build a DHTML client easily. The static nature of HTML limits the interactivity of web pages. Dynamic HTML allows you to create more interactive and engaging web pages. It gives content providers new controls and allows them to manipulate the contents of HTML pages through scripting.

IBM® ILOG® JViews provides a set of Dynamic HTML components written in JavaScript™ that allows you to build your DHTML pages very easily. The JavaScript files are located in ***<installdir>*** **`/jviews-framework86/lib/thinclient/javascript`**.

Important: This sample is compatible with the browsers and browser versions listed in the Release notes under *Requirements for running thin-client applications*.

The Dynamic HTML client for the XML Grapher example includes most of the DHTML components. The full HTML file for the XML Grapher example is located in ***<installdir>*** **`/jviews-framework86/samples/xmlgrapher/index.html`**.

The full reference documentation of each component can be found in the *JavaScript Reference Manual* located in ***<installdir>*** **`/jviews-framework86/doc/html/en-US/refjsf/html/index.html`**.

The IlvView JavaScript component

The `IlvView` component (located in the `IlvView.js` file) is the main component. This component queries the servlet and displays the resulting image.

To use this component, you need to include the following JavaScript™ files: `IlvUtil.js` , `IlvView.js` , the files for the superclasses of `IlvView`: `IlvAbstractView.js`, `IlvResizableView.js`, and `IlvEmptyView.js`, and `IlvGlassView.js`.

Instead of including the individual `.js` files of each component, you can add the file `framework.js` which is located in *<installdir>* `/jviews-framework86/lib/thinclient/framework/framework.js`

This file is a concatenation of all the `.js` files required for doing DHML thin client in the Framework.

Here is a simple HTML page that creates an instance of `IlvView`:

HTML code

```
<html>
<head>
<META HTTP-EQUIV="Expires" CONTENT="Mon, 01 Jan 1990 00:00:01 GMT">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
</head>
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvView.js"></script>
<script TYPE="text/javascript">

function init() {
  view.init()
  return false
}

function handleResize() {
  if (document.layers)
   window.location.reload()
}
</script>
<body onload="init()" onunload=”IlvObject.callDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440
```

```
// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.toHTML()

</script>
</body>
</hmtl>
```

This example starts by importing some JavaScript files:

```

<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvView.js"></script>
```

In the body of the page, the example creates an `IlvView` object located in (40, 40) on the HTML page. The size is 440 x 270. This view displays images produced by the servlet `XmlGrapherServlet`. Note the `toHTML` method that creates the HTML necessary for the component.

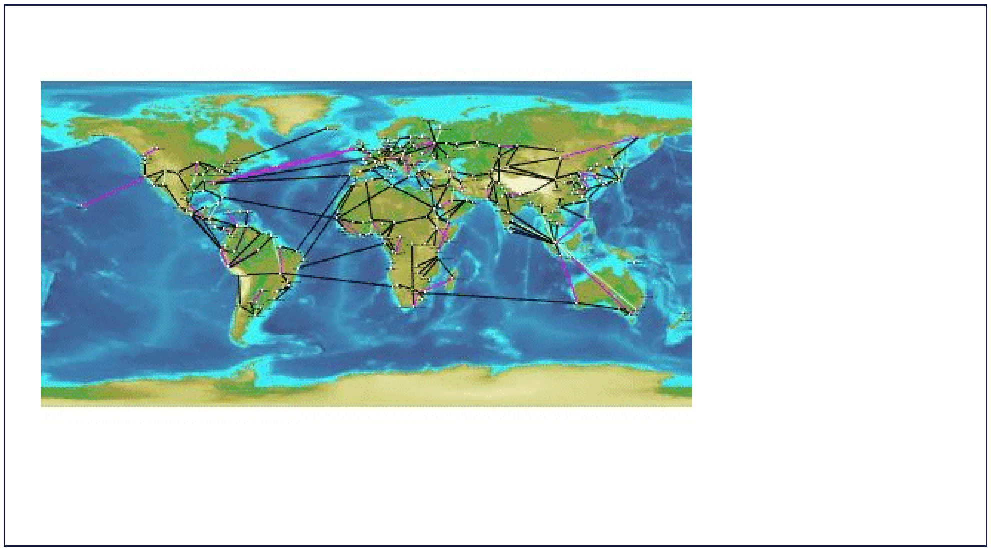
This example also defines two JavaScript functions:

♦ The `init` function, called on the `onload` event of the page, initializes the `IlvView` by calling its `init` method.

♦ The `handleResize` function, called on the `onresize` event of the page, will reload the page if the browser is Netscape Communicator 4 or higher. This is necessary for a correct resizing of Dynamic HTML content on Communicator.

Note: The global `IlvObject.callDispose()` function must be called in the `onunload` event of the HTML page. This function disposes of all resources acquired by the JViews DHTML components.

Once the image is loaded from the server, the page now looks like this:



Generated HTML Page

The IlvOverview JavaScript component

The `IlvOverview` component (located in the `IlvOverview.js`) file shows an overview of the manager. An `IlvOverview` is linked to an `IlvView` component. By default, the `IlvOverview` queries the server to obtain an image of the global area and displays it. Once the overview is visible, a rectangle corresponding to the area visible in the main view is drawn on top of the overview. You can move this rectangle to change the area visible in the main view.

Here is the body of the previous example with an `IlvOverview` component. Note that you cannot move the rectangle of the overview now because the complete area is visible in the main view. You will be able to do that later when the zooming functionality is added.

Note: The lines added are in **bold**.

```
<body onload="init()" onunload=”IlvObject.callDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL(’/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet’)

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor(’white’)

view.toHTML()
overview.toHTML()

</script>
```

Compared to the previous example, there is a new import statement for `IlvOverview.js`:

```
<script TYPE="text/javascript"  src="script/IlvOverview.js"></script>
```

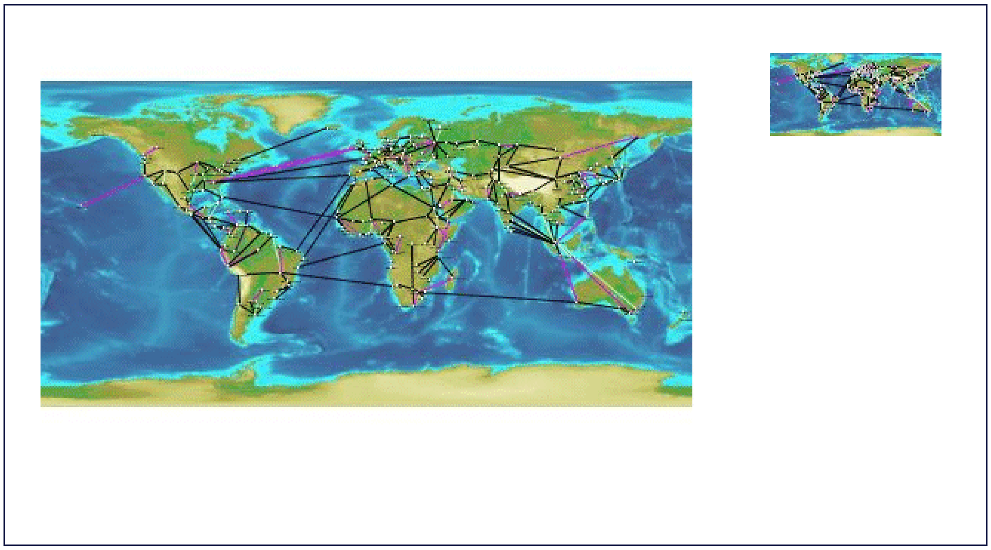
An `IlvOverview` object located in (x+w+50, y+4) with a size of 120 x 70 was created:

```
var overview = new IlvOverview(x+w+50, y+4, 120, 70, view)
```

The following line sets the color of the draggable rectangle:

```
overview.setColor('white')
```

The page looks now like this:



The IlvLegend JavaScript component

You can add an `IlvLegend` component to the page. The `IlvLegend` component shows a list of layers that are available on the server side, and allows you to turn the visibility of a layer on and off.

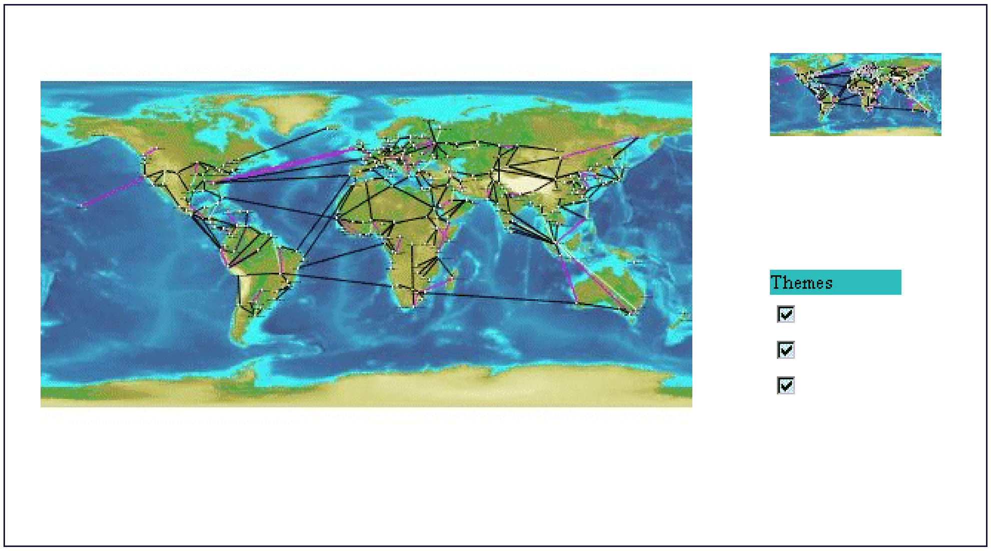
To use the `IlvLegend`, you must first include the `IlvLegend.js` file.

```
<script TYPE="text/javascript" src="IlvLegend.js"></script>
```

The body of the HTML file now looks like this:

```
<body onload="init()" onunload="IlvObject.callDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor('white')

// Legend
var legend = new IlvLegend(x+w+50, y+150 ,120, 115, view)
legend.setTitle('Themes')
legend.setTitleBackgroundColor('#21bdbd')
legend.setTextColor('white')
legend.setBackgroundColor('#21d6d6')
legend.setTitleFontSize(2);

view.toHTML()
overview.toHTML()
legend.toHTML()
</script>
</body>
```

You should see the following page:



The visibility of layers can now be turned on and off.

The IlvButton JavaScript component

The `IlvButton` component is a simple button that allows you to call some JavaScript™ code by clicking it. You can add some buttons to the page to zoom in and out.

In addition to buttons, you can add some Dynamic HTML panels to create a frame around the main view. A Dynamic HTML panel is an area of the page that can contain some HTML. Creating a panel is done using the class `IlvHTMLPanel`, defined in the `IlvUtil.js` file.

The body of the page is now:

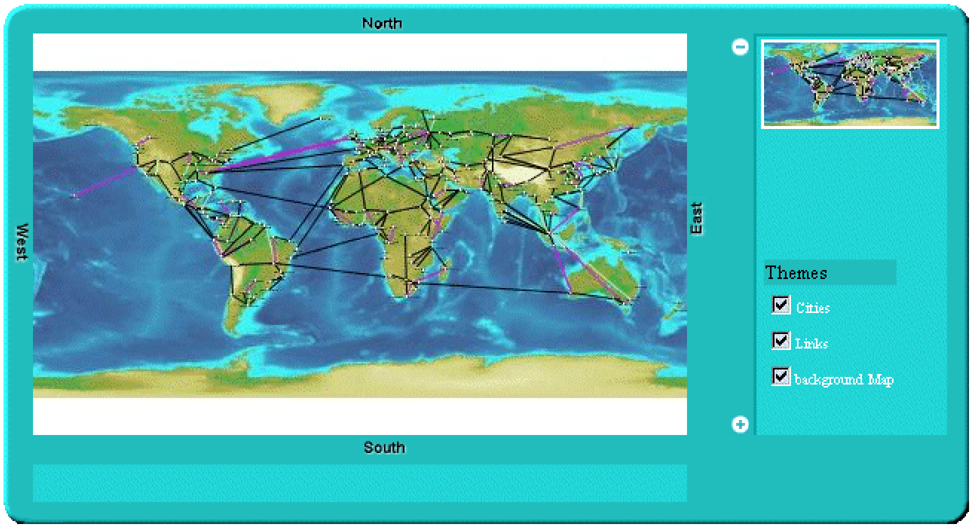
```
<body onload="init()" onunload="IlvObject.callDispose()"
      onresize="handleResize()" bgcolor="#ffffff" >
<script>

//position of the main view

var y = 40
var x = 40
var h = 270
var w = 440

// Creates a frame around the main view
var frameBackground = new IlvHTMLPanel('')
frameBackground.setBounds(x-20, y-20, w+210, h+80)
frameBackground.setVisible(true)
frameBackground.setBackgroundColor('#21bdbd')

var frameTopLeft = new IlvHTMLPanel('<IMG src="images/frame_topleft.gif">')
frameTopLeft.setBounds(x-20, y-20, 40, 40)
frameTopLeft.setVisible(true)

var frameBottomLeft =new IlvHTMLPanel('<IMG src="images/frame_bottomleft.gif">')
frameBottomLeft.setBounds(x-20, y+h+20, 40, 40)
frameBottomLeft.setVisible(true)

var frameTopRight = new IlvHTMLPanel('<IMG src="images/frame_topright.gif">')
frameTopRight.setBounds(x+w+150, y-20, 40, 40)
frameTopRight.setVisible(true)

var frameBottomRight = new IlvHTMLPanel('<IMG src="images/frame_bottomright.
gif">')
frameBottomRight.setBounds(x+w+150, y+h+20, 40, 40)
frameBottomRight.setVisible(true)

var frameTop = new IlvHTMLPanel('<IMG src="images/frame_top.gif">')
frameTop.setBounds(x+20, y-20, 570, 40)
frameTop.setVisible(true)

var frameBottom = new IlvHTMLPanel('<IMG src="images/frame_bottom.gif">')
frameBottom.setBounds(x+20, y+h+20, 570, 40)
frameBottom.setVisible(true)
```

```
var frameLeft = new IlvHTMLPanel('<IMG src="images/frame_left.gif">')
frameLeft.setBounds(x-20, y+20, 5, 270)
frameLeft.setVisible(true)
var frameRight = new IlvHTMLPanel('<IMG src="images/frame_right.gif">')
frameRight.setBounds(x+w+185, y+20, 5, 270)
frameRight.setVisible(true)

var border = new IlvHTMLPanel('')
border.setBounds(x+w+45, y, 130, h)
border.setVisible(true)
border.setBackgroundColor('#09a5a5')

var secondBorder = new IlvHTMLPanel('')
secondBorder.setBounds(x+w+47, y+2, 128, h-2)
secondBorder.setVisible(true)
secondBorder.setBackgroundColor('#21d6d6')

// message panel
var messagePanel = new IlvHTMLPanel('')
messagePanel.setBounds(x, y+h+20, w, 25)
messagePanel.setVisible(true)
messagePanel.setBackgroundColor('#21d6d6')
IlvButton.defaultMessagePanel = messagePanel;

// IBM® ILOG® logo
var logo = new IlvHTMLPanel('<IMG src="images/ilog.gif">')
logo.setBounds(x+w+95, y+h+10, 85, 40)
logo.setVisible(true)

IlvButton.defaultInfoPanel = messagePanel;

// Main view
var view = new IlvView(x, y, w, h)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.setMessagePanel(messagePanel)

// Overview window.
var overview=new IlvOverview(x+w+50, y+4, 120, 70,  view)
overview.setColor('white')
overview.setMessagePanel(messagePanel)

// Legend
var legend = new IlvLegend(x+w+50, y+150 ,120, 115, view)
legend.setTitle('Themes')
legend.setTitleBackgroundColor('#21bdbd')
legend.setTextColor('white')
legend.setBackgroundColor('#21d6d6')
legend.setTitleFontSize(2);
// Some buttons for navigation
var topbutton, bottombutton, rightbutton, leftbutton

topbutton = new IlvButton(x+w/2, y-15, 30, 13,'images/north.gif','view.panNorth
()')
```

```
topbutton.setRolloverImage('images/northh.gif')
topbutton.setToolTipText('pan north')
topbutton.setMessage('pan the map to the north')
```

```

bottombutton = new IlvButton(x+w/2, y+h, 33, 13,'images/south.gif','view.
panSouth()')
bottombutton.setRolloverImage('images/southh.gif')
bottombutton.setToolTipText('pan south')
bottombutton.setMessage('pan the map to the south')

leftbutton=new IlvButton(x-13, y+h/2-10, 13, 30,'images/west.gif','view.panWest
()')
leftbutton.setRolloverImage('images/westh.gif')
leftbutton.setToolTipText('pan west')
leftbutton.setMessage('pan the map to the west')

rightbutton=new IlvButton(x+w, y+h/2-25, 13, 28, 'images/east.gif', 'view.
panEast()')
rightbutton.setRolloverImage('images/easth.gif')
rightbutton.setToolTipText('pan east')
rightbutton.setMessage('pan the map to the east')

// Buttons to zoom in and out
var zoominbutton, zoomoutbutton

zoominbutton=new IlvButton(x+w+30, y+h-16,12, 12, 'images/zoom.gif', 'view.
zoomIn()')
zoominbutton.setRolloverImage('images/zoomh.gif')
zoominbutton.setMessage('click to zoom by 2')
zoominbutton.setToolTipText('Zoom In')

zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, 'images/unzoom.gif', 'view.
zoomOut()')
zoomoutbutton.setRolloverImage('images/unzoomh.gif')
zoomoutbutton.setMessage('click to zoom out by 2')
zoomoutbutton.setToolTipText('Zoom Out')

view.toHTML()
overview.toHTML()
legend.toHTML()
topbutton.toHTML()
bottombutton.toHTML()
leftbutton.toHTML()
rightbutton.toHTML()
zoomoutbutton.toHTML()
zoominbutton.toHTML()

</script>
</body>
</hmtl>
```

The page now looks like this:



A frame around the page was created by the following lines:

```
var frameBackground = new IlvHTMLPanel('')
frameBackground.setBounds(x-20, y-20, w+210, h+80)
frameBackground.setVisible(true)
frameBackground.setBackgroundColor('#21bdbd')

var frameTopLeft = new IlvHTMLPanel('<IMG src="images/frame_topleft.gif">')
frameTopLeft.setBounds(x-20, y-20, 40, 40)
frameTopLeft.setVisible(true)

var frameBottomLeft=new IlvHTMLPanel('<IMG src="images/frame_bottomleft.gif">')
frameBottomLeft.setBounds(x-20, y+h+20, 40, 40)
frameBottomLeft.setVisible(true)

var frameTopRight = new IlvHTMLPanel('<IMG src="images/frame_topright.gif">')
frameTopRight.setBounds(x+w+150, y-20, 40, 40)
frameTopRight.setVisible(true)

var frameBottomRight = new IlvHTMLPanel('<IMG src="images/frame_bottomright.
gif">')
frameBottomRight.setBounds(x+w+150, y+h+20, 40, 40)
frameBottomRight.setVisible(true)

var frameTop = new IlvHTMLPanel('<IMG src="images/frame_top.gif">')
frameTop.setBounds(x+20, y-20, 570, 40)
frameTop.setVisible(true)

var frameBottom = new IlvHTMLPanel('<IMG src="images/frame_bottom.gif">')
frameBottom.setBounds(x+20, y+h+20, 570, 40)
frameBottom.setVisible(true)
```

```
var frameLeft = new IlvHTMLPanel('<IMG src="images/frame_left.gif">')
frameLeft.setBounds(x-20, y+20, 5, 270)
frameLeft.setVisible(true)

var frameRight = new IlvHTMLPanel('<IMG src="images/frame_right.gif">')
frameRight.setBounds(x+w+185, y+20, 5, 270)
frameRight.setVisible(true)
```

This creates four DHTML panels for the corners, four additional panels for the sides, and a panel for the background. The corners and the sides of the frame are composed of simple GIF images.

Four buttons to pan south, north, east, and west have been added by the lines:

```
topbutton = new IlvButton(x+w/2, y-15, 30, 13,'images/north.gif','view.panNorth
()')
topbutton.setRolloverImage('images/northh.gif')
topbutton.setToolTipText('pan north')
topbutton.setMessage('pan the map to the north')

bottombutton = new IlvButton(x+w/2, y+h, 33, 13,'images/south.gif','view.
panSouth()')
bottombutton.setRolloverImage('images/southh.gif')
bottombutton.setToolTipText('pan south')
bottombutton.setMessage('pan the map to the south')

leftbutton=new IlvButton(x-13, y+h/2-10, 13, 30,'images/west.gif','view.panWest
()')
leftbutton.setRolloverImage('images/westh.gif')
leftbutton.setToolTipText('pan west')
leftbutton.setMessage('pan the map to the west')

rightbutton=new IlvButton(x+w, y+h/2-25, 13, 28, 'images/east.gif', 'view.
panEast()')
rightbutton.setRolloverImage('images/easth.gif')
rightbutton.setToolTipText('pan east')
rightbutton.setMessage('pan the map to the east')
```

A button is defined by its position and size, two images, the main image and the rollover image, and a piece of JavaScript to be executed when the button is clicked.

Note that in order to pan to the north, you use the `panNorth` method of `IlvView`.

Two additional buttons have been created to zoom in and out, by the lines:

```
var zoominbutton, zoomoutbutton

zoominbutton=new IlvButton(x+w+30, y+h-16,12, 12, 'images/zoom.gif', 'view.
zoomIn()')
zoominbutton.setRolloverImage('images/zoomh.gif')
zoominbutton.setMessage('click to zoom by 2')
zoominbutton.setToolTipText('Zoom In')
```

```
zoomoutbutton=new IlvButton(x+w+30, y, 12, 12, 'images/unzoom.gif', 'view.
zoomOut()')
zoomoutbutton.setRolloverImage('images/unzoomh.gif')
zoomoutbutton.setMessage('click to zoom out by 2')
zoomoutbutton.setToolTipText('Zoom Out')
```

Each button has a `message` property. The message will be automatically displayed in the status window of the browser when the mouse is over the button. The message can also be displayed in an additional panel. This is why the line:

```
IlvButton.defaultInfoPanel=messagePanel
```

tells you that messages of buttons will also be displayed in the DHTML message panel.

The IlvZoomTool JavaScript component

The `IlvZoomTool` component is a DHTML component that shows a set of buttons. Each button corresponds to a zoom level; clicking the button will zoom the view to this zoom level. The button corresponding to the current zoom level is visually different from others so that you can tell what the current zoom level is. The component can be vertical or horizontal, and the images of the buttons can be customized.

To add the component, add the following lines to the page:

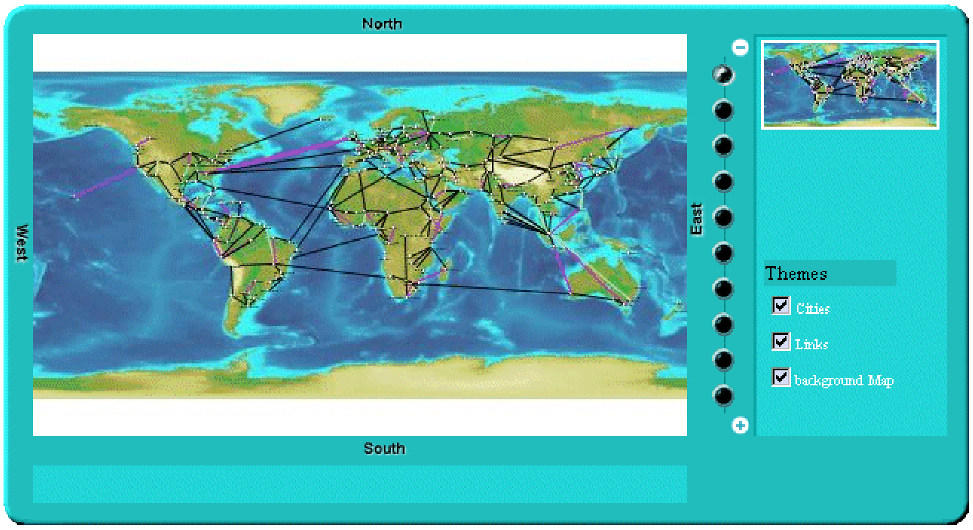
```
<script TYPE="text/javascript" src="script/IlvZoomTool.js"></script>
```

This line imports the script.

Note that this component uses the `IlvButton` class, so the `IlvButton.js` script must be included also.

```
var zoomtool = new IlvZoomTool(x+w+25, y+15, 25, h-30, 10 , view)
zoomtool.setOrientation('Vertical')
zoomtool.upImage = 'images/button.gif'
zoomtool.rolloverUpImage = 'images/buttonh.gif'
zoomtool.downImage = 'images/button.gif'
zoomtool.rolloverDownImage = 'images/buttonh.gif'
zoomtool.currentImage = 'images/center.gif'
zoomtool.rolloverCurrentImage = 'images/centerh.gif'

zommtool.toHTML()
```

The page now looks like this, with the vertical zoom tool on the right of the main view:



The IlvZoomInteractor JavaScript component

The `IlvZoomInteractor` allows direct interaction with the image; it allows the user to select an area on the image to zoom this area. Installing an interactor on the view is simple: you need only create the interactor and set it to the view:

```
var zoomInteractor = new IlvZoomInteractor()
view.setInteractor(zoomInteractor)
```

In the example, you add a button that will install the interactor. To do this, add the following lines to the page:

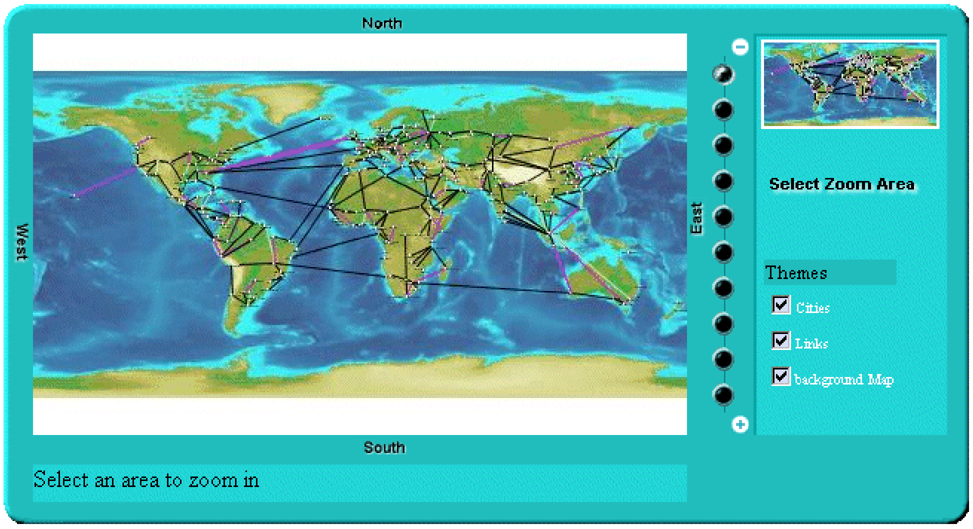
```
<script TYPE="text/javascript"
      src="script/IlvInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvDragRectangleInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvZoomInteractor.js"></script>
<script TYPE="text/javascript"
      src="script/IlvInteractorButton.js"></script>
```

To use the interactor, you have to import three JavaScript™ files: `IlvInteractor.js`, `IlvDragRectangleInteractor.js`, and `IlvZoomInteractor.js`. This is because the `IlvZoomInteractor` component is a subclass of the `IlvDragRectangleInteractor` component.

Then you add the following lines to the body of the page:

```
var zoomInteractor = new IlvZoomInteractor()
zoomInteractor.setLineWidth(1)
zoomInteractor.setColor('#00ffff')

...

var zoomrectbutton

zoomrectbutton=new IlvInteractorButton(x+w+50, y+90, 112, 24,
                                      'images/zoomrect.gif', zoomInteractor,
view)
zoomrectbutton.setRolloverImage('images/zoomrecth.gif')
zoomrectbutton.setMessage('click to set zoom mode')
zoomrectbutton.setToolTipText('Zoom Mode')

...

zoomrectbutton.toHTML()
```

This results in the following page:



You can now click the “Select Zoom Area” button to install the interactor and then select an area to zoom in.

IlvPanInteractor

The `IlvPanInteractor` component allows the user to click in the main view to pan the view. Just as for the `IlvZoomInteractor`, use the `setInteractor` method of `IlvView` to install the interactor. In the example, add another button that will install this interactor (see *The IlvZoomInteractor JavaScript component*). You will now be able to switch from the “Pan” mode and the “Zoom” mode.

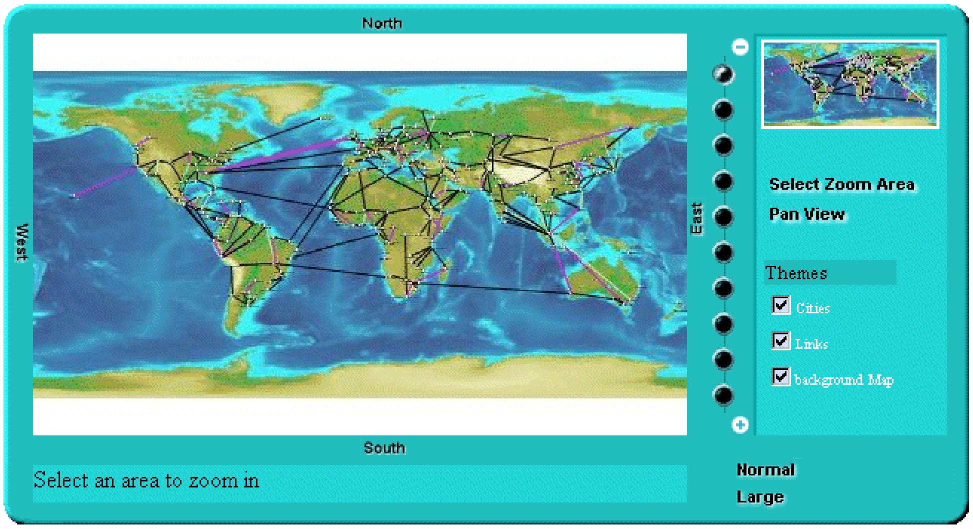
To be able to use the component, import the corresponding JavaScript™ file:

```
<script TYPE="text/javascript"
      src="script/IlvPanInteractor.js"></script>
```

Then add the following lines to the body of the page:

```
var panInteractor  = new IlvPanInteractor()
panbutton=new IlvInteractorButton(x+w+50, y+110, 63, 22, 'images/pan.gif',
                                  panInteractor, view)
panbutton.setRolloverImage('images/panh.gif')
panbutton.setMessage('click to set pan mode')
panbutton.setToolTipText('Pan Mode')
...

panbutton.toHTML()
```

The page now has one additional button labelled “Pan View”:



Select an area to zoom in

The example is now complete; it uses most of the DHTML components provided by IBM® ILOG® JViews.

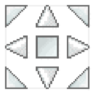
The IlvPanTool JavaScript component

The `IlvPanTool` component (located in the `IlvPanTool.js` file) is a component that allows panning of the view in all directions. You create the component in this way:

```
var pantool = new IlvPanTool(10, 10, view)
pantool.toHTML()
```

Note that this component uses the `IlvButton` class, so the `IlvButton.js` script must be included also.

This component looks like this:



The IlvMapInteractor and IlvMapRectInteractor JavaScript components

The `IlvMapInteractor` and `IlvMapRectInteractor` components are two additional interactors that can be used to perform an action on the server side when a point or an area of the image is selected by the client. These interactors and how to use them are described in detail in *Adding client/server interactions*.

The Popup menu in JavaScript

The popup menu component is attached to the main view. This popup menu in JavaScript™ is triggered by a right-click in the view.

To use the popup menu, you must first include the following scripts.

The popup menu component is `IlvGanttPopupMenu`.

```
<script TYPE="text/javascript" src="script/IlvAbstractPopupMenu.js"></script>
<script TYPE="text/javascript" src="script/gantt/IlvGanttPopupMenu.js">
</script>
```

The popup menu component is `IlvPopupMenu`.

```
<script TYPE="text/javascript" src="script/IlvAbstractPopupMenu.js"></script>
<script TYPE="text/javascript" src="script/framework/IlvPopupMenu.js"></script>
```

The popup menu can be contextual or static.

Static popup menu

The menu is static, that is, not conditioned by the context in which it is called, and is defined in the HTML file by using `IlvMenu` and `IlvMenuItem` instances. The menu is a pure client-side object and there is no roundtrip to the server to generate the menu.

Defining a static popup menu in the HTML file

```
//Creates the popup menu.
var popupmenu = new IlvGanttPopupmenu(true);

//Creates the menu model.
var root = new IlvMenu("root");
var item1 = new IlvMenuItem("item1", true, "alert('item1 clicked')");
var item2 = new IlvMenuItem("item2", true, "alert('item2 clicked')");
root.add(item1);
root.add(item2);

//Sets the menu model to the popup menu.
popupMenu.setMenu(root);

[...]

//Sets the popup menu to the view.
chartView.setPopupMenu(popupMenu);
```

Configuring servlet support for a popup menu

```
public class GanttChartServlet extends IlvGanttServlet
{
       [...]
  protected void configureServletSupport(IlvGanttServletSupport support) {
   [...]
                support.setPopupEnabled(true);
  }
       [...]
}
```

Defining a static popup menu in the HTML file at the JViews Framework level

```
//Creates the popup menu.
var popupMenu = new IlvPopupMenu();

//Creates the menu model.
var root = new IlvMenu("root");
var item1 = new IlvMenuItem("item1", true, "alert('item1 clicked')");
var item2 = new IlvMenuItem("item2", true, "alert('item2 clicked')");
root.add(item1);
root.add(item2);

//Sets the menu model to the popup menu.
popupMenu.setMenu(root);

[...]

//Sets the popup menu to the view.
view.setPopupMenu(popupMenu);
```

On the server side, you need to configure the servlet support to handle popup menu server-side actions.

Configuring servlet support for a popup menu at the JViews Framework level

```
public class XmlGrapherServlet extends IlvManagerServlet {
  public XmlGrapherServlet() {
    [...]
    getSupport().setPopupEnabled(true);
 }
 [...]
}
```

Contextual popup menu

The popup menu is dynamically generated by the server depending on:

♦ The `menuModelId` property of the current interactor set on the view.

♦ The object selected when the user triggered the popup menu.

On the client side, you need only declare the popup menu and set it on the view.

Declaring a contextual popup menu and setting it on the view, client side

```
var popupMenu = new IlvGanttPopupMenu(true);
```

```
//Sets the popup menu to the view
chartView.setPopupMenu(popupMenu);
```

On the server side, you need to configure the servlet support to handle popup menus and to set the factory that will generate the menu.

Configuring servlet support and setting the factory, server side

```
public class GanttChartServlet extends IlvGanttServlet {

        [...]
  protected void configureServletSupport(IlvGanttServletSupport support) {
    [...]
                support.setPopupEnabled(true);
     support.getPopupMenuSupport().setMenuFactory(new SimpleMenuFactory());
  }
       [...]
}
```

Declaring a contextual popup menu and setting it on the view, client side (JViews Framework level)

```
var popupMenu = new IlvPopupMenu();

//Sets the popup menu to the view
view.setPopupMenu(popupMenu);
```

On the server side, you need to configure the servlet support to handle popup menus and to set the factory that will generate the menu.

Configuring servlet support and setting the factory, server side (JViews Framework level)

```
public class XmlGrapherServlet extends IlvManagerServlet {

  public XmlGrapherServlet() {
    getSupport().setPopupEnabled(true);
    getSupport().getPopupMenuSupport().setMenuFactory
      (new XmlGrapherMenuFactory());
  }
  [...]
}
```

The factory must implement the `IlvMenuFactory` interface.

Styling the popup menu

You can style the popup menu by setting a CSS class name in the following properties:

♦ `itemStyleClass`: the base CSS class name applied to a menu item.

♦ `itemHighlightedStyleClass`: the style applied over the base style when the cursor is over the item.

♦ `itemDisabledStyleClass`: the style applied over the base style when the cursor is disabled.

The following example shows how to use CSS to style the popup menu.

```
  [...]

<style>
.PopupMenuItem {
  background: #21bdbd;
  color: black;
  font-family: sans-serif;
  font-size: 12px;
}

.PopupMenuItemHighlighted {
  background: #057879;
  font-style: italic;
  color: white;
}

.PopupMenuItemDisabled {
  background-color: #EEEEEE;
  font-style: italic;
  color: black;
}
</style>

  [...]

<script>

var popupMenu = new IlvGanttPopupMenu(true);
popupMenu.setItemStyleClass('PopupMenuItem');
popupMenu.setItemHighlightedStyleClass('PopupMenuItemHighlighted');
popupMenu.setItemDisabledStyleClass('PopupMenuItemDisabled');

</script>
```

At the JViews Framework level

```
  [...]

<style>
.PopupMenuItem {
  background: #21bdbd;
  color: black;
  font-family: sans-serif;
  font-size: 12px;
}

.PopupMenuItemHighlighted {
  background: #057879;
  font-style: italic;
  color: white;
}
```

```
.PopupMenuItemDisabled {
  background-color: #EEEEEE;
  font-style: italic;
  color: black;
}
</style>

  [...]

<script>

var popupMenu = new IlvPopupMenu();
popupMenu.setItemStyleClass('PopupMenuItem');
popupMenu.setItemHighlightedStyleClass('PopupMenuItemHighlighted');
popupMenu.setItemDisabledStyleClass('PopupMenuItemDisabled');

</script>
```

Adding client/server interactions

Overview of actions on the server and client sides

The IBM® ILOG® JViews thin-client support gives you a simplified way to define new actions that should take place on the server side. For example, suppose you want to allow the user to delete a graphic object that appears on the generated image. Part of this action—clicking the image to select the object—must be done on the client side. The destruction of the object must be done on the server side before a new image is generated. The notion of “server-side action” exists to perform such behavior. An action is defined by a name and a set of string parameters.

Actions on the client side

In a dynamic HTML client, you tell the server to perform an action using the `performAction` method of the `IlvView` JavaScript™ component.

Here is an example that asks the server side to execute the action “delete” with coordinate parameters, assuming that `view` is an `IlvView`:

```
var x = 100;
var y = 50;
var params = new Array();
params[0]=x;
params[1]=y;
view.performAction(“delete”, params);
In a thin-Java client the system is the same:
float x = 100f;
float y = 50f;
String[] params = new String[2];
params[0] = Float.toString(x);
params[1] = Float.toString(y);
view.performAction(“delete”, params);
```

The `performAction` method will ask the server for a new image. In the image request, additional parameters are added so that the server side can execute the action. Thus, the `performAction` call results in only one client/server round-trip.

Note that predefined interactors are provided to help you define new actions on the client side. They are explained in *Predefined interactors*.

Actions on the server side

On the server side, you need to detect that an action was requested and execute the action. This is done using the interface `ServerActionListener`.

To be able to listen and execute an action on the server side, you simply add an action listener to your servlet. In the `performAction` method of the listener, you check the action name and perform the action.

For the “delete” action, we would add the following lines of code in the `init` method of the servlet:

```
addServerActionListener(new ServerActionListener() {
   public void actionPerformed(ServerActionEvent e) throws ServletException
  {
    if (e.getActionName().equals("delete")) {
      IlvPoint p = e.getPointParameter(0);
        // find object under this point and delete it if there is one.
    }
  }
});
```

The `ServerActionEvent` object can give you all necessary information about the action, the name, and its parameters.

Predefined interactors

Two predefined interactors are provided to help you create new actions: `IlvMapInteractor` and `IlvMapRectInteractor`.

`IlvMapInteractor` allows the user to click in the map; it will ask the server to execute an action, with the coordinates of the clicked point passed as parameters. The second interactor is almost the same except that the user selects an area of the image instead of clicking on it.

Generating a client-side image map

If you are creating a Dynamic HTML client, the IBM® ILOG® JViews thin-client support allows you to create a client-side image map. Image maps are images with an attached map that points out hot spots, or clickable areas. In the IBM® ILOG® JViews thin-client support, a clickable area can be generated for each graphic object of the manager.

To create a client side image map:

- Define the image map on the server side
- Use the image map on the client side

Define the image map on the server side

The servlet provided by IBM® ILOG® JViews (`IlvManagerServlet`) is able to generate an image map for your IBM® ILOG® JViews application, but it is likely that you do not want to generate a clickable area for every graphic object. On the server side, you will then have to tell the manager servlet which IBM® ILOG® JViews layer and which graphic object are part of the image map generation. For both layer and graphic object, this is done by setting a property on them.

On a layer, assuming that the variable `manager` is an `IlvManager`, you will do:

```
manager.getManagerLayer(index).setProperty( IlvManagerServlet.
ImageMapAreaGeneratorProperty, Boolean.TRUE);
```

On a graphic object you can do almost the same thing, but the value of the property must be an instance of the class `IlvImageMapAreaGenerator`. This class is responsible for generating the AREA part of the image map.

Note that the same instance of `IlvImageMapAreaGenerator` can be used for all graphic objects.

By default, `IlvImageMapAreaGenerator` will generate a rectangular area with no HREF in it. You will have to subclass it to generate an HREF for your graphic object.

Here is an example that creates a custom `IlvImageMapAreaGenerator` and sets it on some objects:

```
IlvGraphic object1, object2;
....
IlvImageMapAreaGenerator generator = new IlvImageMapAreaGenerator() {
    public String generateHREF(IlvManagerView v, IlvGraphic obj) {
    String href;
    // place here code the
    // computes the URL depending on the graphic object
    return href;
  }

};
```

```
object1.setProperty(IlvManagerServlet.ImageMapAreaGeneratorProperty,
                      generator);
object2.setProperty(IlvManagerServlet.ImageMapAreaGeneratorProperty,
                      generator);
```

The HREF can be a URL to which the browser will jump when the area is clicked, but it can also be a call to a JavaScript™ method.

For example, in the XML Grapher example, you can define the generator like this:

```
IlvImageMapAreaGenerator generator = new IlvImageMapAreaGenerator() {

  public String generateALT(IlvManagerView v, IlvGraphic obj) {
     return ((GrapherNode)obj).getLabel();
  }

  public String generateHREF(IlvManagerView v, IlvGraphic obj) {
     return "javascript:doSomething('"+
             ((GrapherNode)obj).getLabel()+"')";
  }
};
```

In this example, the HREF generated is a call to the JavaScript method `doSomething`. You will have to define this method in the HTML page.

For more information about customizing an area, see the `IlvImageMapAreaGenerator` class in the *Java API Reference Manual*.

Use the image map on the client side

To tell the Dynamic HTML client to generate a client-side image map, you only need to set the `imageMap` property of the `IlvView` JavaScript™ component to `true`:

```
var view = new IlvView(40, 40, 300, 400);
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet');
view.setGenerateImageMap(true);
```

When this is done, the `IlvView` component will ask the servlet to generate the image map.

To make the image map visible, there are two possibilities. You can:

- Directly call the `showImageMap` method of `IlvView`:

  `view.showImageMap();`

- Use the `IlvImageMapInteractor` class. This class is a simple interactor that will show the image map when installed and hide it when de-installed.

The IlvManagerServlet class

Describes the predefined servlet and how to use it.

In this section

Overview of the predefined servlet

Developing the server side of a thin-client application consists of creating a servlet that can produce an image to the client. IBM® ILOG® JViews Framework provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvManagerServlet`. This class can be found in the package `ilog.views.servlet`.

The `IlvManagerServlet` class is an abstract Java™ subclass of the `HTTPServlet` class from the Java servlet API.

The servlet requests and parameters

The servlet can respond to three different types of HTTP requests, the “image” request, the “image map” request, and the “capabilities” request. The image request will return an image from the IBM® ILOG® JViews manager. The capabilities request will return information to the client, such as the layers available in the manager and the global area of the manager. This information allows the client to know the capabilities of the servlet in order to build the image request. When developing the client side of your application, you will use the DHTML scripts or the JavaBeans™ provided by IBM® ILOG® JViews; both will create the HTTP request for you, so you do not really need to write the HTTP request yourself.

The image request

The image request produces a JPEG image from the manager. The request has the following syntax, assuming that `myservlet` is the name of the servlet:

```
http://host/myservlet?request=image
   &bbox=x,y,width,height (area in the manager coordinate system)
   &width=width of the returned image
   &height=height of the returned image
   &layer=comma separated list of layers
   &format=JPEG
   &bgcolor=0xFFFFFF
```

Here is a list of parameters and their meanings.

Parameters of the IlvManagerServlet

Parameter Name	Parameter Value	Description
`request`	image	Asks the servlet to generate an image.
`bbox`	Float, Float, Float, Float	The area of the manager that will be displayed in the image. The first two values are the upper left

Parameter Name	Parameter Value	Description
		corner of the area. The last two values are the width and height of the area.
`width`	Integer	Width of the resulting image.
`height`	Integer	Height of the resulting image.
`format`	JPEG	The format of the resulting image.
`layer`	Comma-separated list of strings. For example: Cities, Roads	The layers of the `IlvManager` that will be visible.
`bgcolor`	`0xrrggbb` For example, `0xffffff` for white	The background color of the resulting image. This parameter is optional.
`action`	actionName(param1, param2)	Specifies an action to be executed on the server before the image is generated.

The following request will produce a JPEG image of size (250, 250) showing the area (0, 0, 1000, 1000) of the manager; only the layers named "Cities" and "Roads" will be visible:

```
http://host/myservlet?request=image
        &bbox=0,0,1000,1000
        &width=250
        &height=250
        &layer=Cities,Roads
        &format=JPEG
```

The capabilities request

The capabilities request produces information to the client. This request returns information on the manager.

The capabilities request has the following syntax:

```
http://host/myservlet?request=capabilities
        &format=(html|octet-stream)
      [ &onload= <a string> ]
```

The `request` parameter set to `capabilities` instead of `image` tells the servlet to return the capabilities information. The `format` parameter tells which format should be returned.

The result can be of two different formats, HTML or Octet stream.

HTML format

The HTML format is used when the client is a Dynamic HTML client. In this case, the result is a empty HTML page that contains some JavaScript™ code. The JavaScript code is executed on the client side, and some information variables are then available.

```
<html>
<head>
<script language="JavaScript">
var minx=0.0;
var miny=0.0;
var maxx=1024.0;
var maxy=512.0;
var themes=new Array();
var overviewthemes=new Array();
themes[0]="a layer name";
overviewthemes[0]=true;
themes[1]="another layer";
overviewthemes[1]=true;
themes[2]="a third layer";
overviewthemes[2]=true;
var maxZoom=6;
</script>
</head>
<body>
</body>
</html>
```

The variables `minx`, `miny`, `maxx`, `maxy` are defining the global area of the manager that can be queried. The `themes` variable is the list of layers available on the server side. The `overviewthemes` variable tells if a layer should be visible in the overview window. The `maxZoom` variable is the maximum level of zoom the application should perform.

The `onload` parameter allows you to specify a String that is used for the onload event of the generated HTML page. When an `onload` parameter is specified, the body tag of the HTML page is the following:

```
<body onLoad="+onload+">
```

Octet-stream format

The octet-stream format is used when the client is a Java™ applet. In this case, the result is a stream of octets. The data is produced using a `java.io.DataOutput` and can be read using a `java.io.DataInput`. It is organized as follows:

```
 Float: left coordinate of manager’s bounding box.
    Float: top coordinate of manager’s bounding box.
    Float: right coordinate of manager’s bounding box.
    Float: bottom coordinate of manager’s bounding box.
    Int: number of layers.

    for each layer:
       String (UTF format): name of the layer.
       Boolean: is the layer an overview layer.

    Float: Maximum zoom level
```

You see that this format gives the same type of information as the HTML format. Once again, you do not need to decode or read these formats. The client-side components provided by IBM® ILOG® JViews will do that for you.

The image map request

The image map request produces an image and a client-side image map. The parameters for this request are the same as for the image request except that the `request` parameter must have the value `imagemap`.

For example, the following code to the servlet:

```
http://host/myservlet?request=imagemap
        &width=400
        &height=200
        &bbox=0,0,500,500
        &format=JPEG
        &layer=Cities,Links,background%20Map
```

will produce something like:

```
<html>
<body>
<map name="imagemap">
<area shape="rect" coords="242,81,261,83" href="..." >
....
</map>
<img usemap="#imagemap" width="400" height="200"
  src="myservlet?request=image&layer=Cities,Links,background%20Map&width=400
  &format=JPEG&bbox=0,0,500,500&height=200" border=0>
</body>
</html>
```

The call generates an HTML document containing the client-side image map and an image. The contents of the image are then generated by another call to the servlet.

The graphic objects that are taken into account when generating the map can be specified as well as the shape of the clickable area and what appends when you click on it. All this is explained in *Generating a client-side image map*.

The image map request has two additional optional parameters:

♦ The `mapname` parameter allows you to specify the name of the map. The default name is `imagemap`.

♦ The `onload` parameter allows you to specify a String that is used for the onload event of the generated HTML page. When an `onload` parameter is specified, the body tag of the HTML page is the following:

```
<body onLoad="+onload+">
```

Multiple sessions

The XML Grapher is a very simple example that creates a single manager view for the servlet. This means that all calls to the servlet (that is, all clients) are looking at the same view. This is fine when the same data is used for all clients but in some applications—for example, when you want to allow the user to edit the graphic representation—you might want to have a view (and thus a manager) for each client. In this case, you might use the notion of *HTTP sessions*. You can then create a view and a manager and store them as parameters of the session.

Here is a slightly modified version of the XML Grapher servlet using sessions:

```
package demo.xmlgrapher.servlet;
import javax.servlet.*;
import javax.servlet.http.*;
import java.net.*;
import ilog.views.*;
import ilog.views.servlet.*;
import demo.xmlgrapher.*;

public class XmlGrapherServlet extends IlvManagerServlet
{
  String xmlfile;

  public void init(ServletConfig config)
       throws ServletException
  {

    xmlfile = config.getInitParameter("xmlfile");
    if (xmlfile == null)
      xmlfile = config.getServletContext().
                   getRealPath("/data/world.xml");
    setVerbose(true);
  }

  protected void prepareSession(HttpServletRequest request)
  {
    HttpSession session = request.getSession();
    if (session.isNew()) {

      XmlGrapher xmlGrapher = new XmlGrapher();
      try {
        xmlGrapher.setNetwork(new URL("file:" + xmlfile));
      } catch (MalformedURLException ex) {
      }
      session.putValue("IlvManagerView", xmlGrapher);
    }
  }

  public IlvManagerView getManagerView(HttpServletRequest request)
       throws ServletException
  {
    HttpSession session = request.getSession(false);
    if (session!= null)
      return (IlvManagerView)session.getValue("IlvManagerView");
    else
      throw new ServletException("session problem");
  }

  protected float getMaxZoomLevel(HttpServletRequest request,
                                  IlvManagerView view)
  {
    return 30;
  }
}
```

The `init` method does not create any `XmlGrapher` object any more. Instead, the `prepareSession` method (which has a default empty implementation) is overwritten to get the HTTP session. If this is a new session, an `XmlGrapher` object is created and stored as a parameter of the session. The `getManagerView` method returns the `XmlGrapher` object stored in the session.

Multithreading issues

The IlvManagerServlet class does not implement the SingleThreadModel interface from the Servlet API, so you can create servlets that use the multithread or single-thread model.

If your servlet implements the SingleThreadModel interface, then you do not have to deal with concurrent access to your servlet. The servlet will be thread safe. However, this interface does not prevent synchronization problems that result from servlets accessing shared resources such as static class variables or classes outside the scope of the servlet.

If your servlet does not implement the SingleThreadModel interface, then you might have to be concerned with concurrent access to the servlet. All basic operations done by the IlvManagerServlet on the IlvManagerView are already synchronized. This means that you will have to take care of concurrent access only if you are doing additional actions on the IlvManagerView. In this case you can define a locking object and use the getLock method of the IlvManagerServlet. Each request handling is implemented in the following way:

```
... reads the request parameters ...

synchronized(getLock(request)) {
  IlvManagerView view = getManagerView(request);

  ... handle the request ...
}
```

By default, the getLock method returns a new object each time. This means that the section is not synchronized.

The IlvManagerServletSupport class

The `IlvManagerServlet` class used in the XML Grapher example gives an easy way to create a servlet that supports the IBM® ILOG® JViews thin-client protocol. Using the `IlvManagerServlet` class is an easy way to create a servlet but has one main drawback. You cannot add the support for the IBM® ILOG® JViews thin-client protocol to an existing servlet since the `IlvManagerServlet` class derives from the `HttpServlet` class. The `IlvManagerServletSupport` class will allow you to do this. This class has the same API as the `IlvManagerServlet` but is not a servlet (that is, it does not derive from the `HttpServlet` class). You can thus create your own servlet and an instance of the `IlvManagerServlet` support class in this servlet to handle the requests coming from the IBM® ILOG® JViews client side.

Thin-client support in the XML Grapher example

In the XML Grapher example, the code of the servlet can be rewritten using the `IlvManagerServletSupport` class as follows:

```
package demo.xmlgrapher.servlet;

import javax.servlet.*;
import javax.servlet.http.*;

import java.net.*;
import java.io.*;
import ilog.views.*;
import ilog.views.servlet.*;

import demo.xmlgrapher.*;

public class XmlGrapherServlet extends HttpServlet
{
  IlvManagerServletSupport servletSupport ;

  class MySupport extends IlvManagerServletSupport {

    private XmlGrapher xmlGrapher;

    public MySupport(ServletConfig config) {
      super();
      xmlGrapher = new XmlGrapher();

      String xmlfile = config.getInitParameter("xmlfile");

      if (xmlfile == null)
        xmlfile = config.getServletContext().getRealPath("/data/world.xml");

      try {
        xmlGrapher.setNetwork(new URL("file:" + xmlfile));
      } catch (MalformedURLException ex) {
      }
      setVerbose(true);
    }

    public IlvManagerView getManagerView(HttpServletRequest request)
      throws ServletException {
        return xmlGrapher;
      }

    protected float getMaxZoomLevel(HttpServletRequest request,
                                    IlvManagerView view) {
      return 30;
    }
  }

  /**
   * Initializes the servlet.
   */
  public void init(ServletConfig config) throws ServletException {
    servletSupport = new MySupport(config);
  }
```

```


  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException {
    if (!servletSupport.handleRequest(request, response))
      throw new ServletException("unknow request type");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
       throws IOException, ServletException {
    doGet(request, response);
  }

}
```

This code creates a new servlet class, `XmlGrapherServlet`, that derives directly from the `HttpServlet` class. The `doGet` method passes the requests to an instance of the `IlvManagerServletSupport` class.

Specifying fixed zoom levels on the client side

Override the following method of the `IlvManagerServletSupport` class to specify the zoom levels that must be used on the client side:

```
public double[] getZoomLevels(HttpServletRequest request, IlvManagerView view)
```

In this case, the maximum zoom level is not used.

Controlling tiling

Describes how to control tiling on the client side and the server side.

In this section

Tiling

The static layers are represented by a grid of images of a fixed size. These fixed-size images are referred to as tiles. Dynamic layers are represented by a single image with a transparent background overlaying the view.

A static layer is not supposed to change during the application lifecycle and so can be generated once only. Typically, a static layer is the background of the view, such as a background map.

A dynamic layer contains objects, such as symbols, that can move and change their graphic representation.

Note: Dynamic layers must be placed on top of a static layer. Otherwise, they are not displayed.

The advantages of a tiled view are continuous panning and the capability of caching tiles. On the client side this avoids a roundtrip to the server and gives a better response time. On the server side it allows the server to receive the request, retrieve the image, and respond with the image without having to generate it. Not having to generate the image for the response is especially advantageous in complex applications.

Tile size

The size of the tile determines the number of tiles needed to cover the view.

The tile size must be carefully chosen because it can have a considerable and potentially critical impact on performance. The larger the number of tiles needed because of their size relative to the size of the view to be covered, the more simultaneous requests to be addressed to the image servlet. There will also be more graphic objects to manage on the client side.

If a server-side caching mechanism is implemented, such as pregenerated tiles, the size must be consistent with the configuration of the server-side caching mechanism. See `IlvTileManager` for more details about server-side caching mechanisms.

Cache mechanisms

Since tiles in static layers are not subject to change, they can be cached on the client side to be reused directly without the need for a server roundtrip.

You can consider several possible caching strategies on the server side:

- No caching: the server generates the images each time they are requested.
- Dynamic caching: the server can cache every generated tile, for example in the file system. This strategy allows you to have a quicker response for popular tiles and to limit the size of the cache.
- Pregeneration: a partial or complete set of tiles for specific zoom levels can be pregenerated and returned directly by the server without need of dynamic generation.

To manage the cache efficiently on the client and the server, the zoom levels must be fixed. If there is a free choice of what zoom level to apply, the probability of the client retrieving a cached tile is severely limited.

See *Specifying fixed zoom levels on the client side* for how to specify the zoom levels.

Developing client-side tiling

The API of the `IlvTileView` class is very similar to `IlvView`. To use the tiled view, import `IlvTiledView.js` instead of `IlvView.js`.

To instantiate an `IlvTiledView` object, proceed as with `IlvView`, but the class takes an additional argument that defines the tile size as shown in the following XML example.

```
<html>
<head>
<META HTTP-EQIV="Expires" CONTENT="Mon, 01 Jan 1990 00:00:01 GMT">
<META HTTP-EQIV="pRAGMA" CONTENT="No-cache">
</head>
<script TYPE="text/javascript" src="script/IlvUtil.js"></script>
<script TYPE="text/javascript" src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript" src="script/IlvImageView.js"></script>
<script TYPE="text/javascript" src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript" src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript" src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript" src="script/IlvTiledView.js"></script>
<script TYPE="text/javascript">
function init() {
  view.init()
  return false
}

function handleResize() {
  if (document.layers)
    window.location.reload()
}
</script>
<body onload="init()" onunload="IlvObject.callDispose()"
      onresize="handleResize()" bgcolor="#ffffff">
<script>

//position of the main view
var y = 40
var x = 40
var h = 270
var w = 440

//tile size
var t = 256

//Main view
var view = new IlvView(x,y,w,h,t)
view.setRequestURL('/xmlgrapher/demo.xmlgrapher.servlet.XmlGrapherServlet')
view.toHTML()
</script>

</body>
</html>
```

Developing server-side tiling

The tile manager stores and retrieves static and dynamic layers. See In *Server-side caching and the tile manager* for a description of the tile manager and Tiling for what is meant by static and dynamic layers in the context of tiling.

The list of dynamic layers is computed by the following method of the `IlvManagerServletSupport` class:

```
public IlvManagerLayer[] getDynamicLayers(HttpServletRequest request,
                 IlvManagerView view)
```

The default implementation of this method classifies the layers according to the value returned by the `getTripleBufferedLayerCount()` method. If the layer index is greater or equal to this value, the layer is dynamic. If not, it is a static layer. You can override this method to determine which are the dynamic layers in a different way.

Client-side caching

HTTP headers are sent with the tile image to control the caching of tiles on the client side.

There are two ways of specifying expiry data for tiles on the client side.

♦ Override the following method of `IlvManagerServletSupport`:

  ```
  public long getExpirationDate(HttpServletRequest request)
  ```

  This method returns the expiry date in milliseconds of tile lifespan in the client-side cache.

♦ Override the protected method:

  ```
  void setImageResponseCachePolicy(HttpServletRequest request,
  HttpServletResponse response);
  ```

  This method sends the HTTP headers to the client, so that the server instructs the client how to cache the tiles.

See RFC 2616 on HTTP/1.1 for a full description of HTTP headers.

You need to take the following cases into account:

1. The normal image request: you should prevent caching in this case.
2. The tile image request, which is identified by the `tile` request parameter: this type of request can be cached on the client.

Server-side caching and the tile manager

Use `IlvTileManager` to manage caching on the server side.

Static or dynamic layers can be used in conjunction with tiled views on the client side.

Static layers can be cached or pregenerated on the server. Cached tiles are part of layers that are not expected to change within the application lifecycle, as, for example, in a background map. Cached tiles can be retrieved through a tile manager.

Dynamic layers are likely to change between requests to the server, such as labeling or network display.

The tile manager, an instance of `IlvTileManager`, stores and retrieves tiles on the server side. `IlvManagerServlet` can take advantage of such a tile manager if one is installed on the servlet.

When an image request is received by the servlet, if a tile that matches the current request is managed by the tile manager, it will return this cached tile instead of generating a new image from `IlvManagerView`. If a tile is not yet managed by the tile manager, generate the image from `IlvManagerView` and ask the tile manager to manage it for future access.

When `IlvManagerServletSupport` responds to an image request, it uses the tile manager as follows:

```
if (useTileManager(request)) {
   IlvTileManager tm = getTileManager(request);
   if (tm != null) {
    Object key = getKey(request);
    BufferedImage image = tm.getImage(key);
    if (image == null) {
      image = doGenerateImageImpl( ... );
      tm.putImage(key, image);
     }
    return image;
   }
  }
  return doGenerateImageImpl( ... );
```

The tile manager is invoked by default if the request contains a parameter of the form `tile=true`. If the request contains such a parameter, `useTileManager(javax.servlet.http.HttpServletRequest)` will return `true`. You can override the `useTileManager` method to call the tile manager in other situations.

If a tile manager is installed, it will be retrieved and a key object will be constructed from the request to reference the tile. Then, an attempt is made to retrieve a tile from the tile manager. If the attempt is successful, the tile is returned as the response to the request.

If no tile is retrieved, an image will be constructed through the normal image generation process. This image is passed to the tile manager for use in future retrievals.

The tile manager is not installed by default in an `IlvManagerServletSupport` object. You need to subclass it to install a tile manager.

The method to override is `getTileManager(javax.servlet.http.HttpServletRequest)`. By default, this method returns null.

```
protected IlvTileManager getTileManager(HttpServletRequest request)
                throws ServletException {
   return null;
}
```

A default implementation of the tile manager is supplied. This implementation stores tiles on disk. You can use it to develop your own implementation of the `getTileManager` method.

```
protected IlvTileManager getTileManager(HttpServletRequest request)
                throws ServletException {
   ServletContext context = request.getSession().getServletContext();
   IlvTileManager tileManager =          (IlvTileManager)context.getAttribute
("tileManagerKey");
   if(tileManager == null) {
     tileManager = new IlvFileTileManager(getBase(), getMaxCacheSize(),
      getMinCacheSize());
     context.setAttribute("tileManagerKey", tileManager);
   }
   return tileManager;
}
```

In this implementation you need to provide:

♦ The base directory where the tiles are written.

♦ The maximum size allowed for the cache.

♦ The size to which the cache will be reduced by removing files when the maximum size is reached.

  When the maximum size is reached, the cache is considered to be full and files will be removed to reduce the size of the cache to the level indicated.

The tile manager is stored and retrieved from the `ServletContext`, so that the same tile manager is used for the same application. You can use a different strategy for storing and retrieving the tile manager.

You can also customize the reading and writing of tiles and the name of the file that is generated for each tile. This default implementation of the tile manager constructs a file name of the form `x_y_width_height.jpg`, where `x`, `y`, `width`, and `height` are the manager coordinates of the image request passed as the `bbox` attribute of the request.

This file is stored in and retrieved from the base directory provided when the `IlvFileTileManager` is constructed. This customization can be performed through the `IlvFileTileURLFactory`, which is responsible for building a URL from the key that identifies the tile. The default key is a `Rectangle2D.Double` object, which is created from the `bbox` parameter of the request.

Index