IBM ILOG JViews Maps V8.6

Programming with JViews Maps

Table of contents

Introducing the main classes

Introduces the main classes of the JViews Maps library.

In this section

Reader framework
Describes the classes for reading data in the reader framework.

Map layers and map styles
Describes map layers and map styles.

Map-specific manager properties
Describes the classes available for managing various aspects of maps, such as altitude, display, data sources, layers, threads, and labeling.

Readers and writers
Introduces you to the predefined reader/writer classes supplied with JViews Maps.

Raster image management
Describes the management of raster images including tile loading, subsampling, persistence, and storage.

Graphical User Interface beans and interactors
Describes the classes for JavaBeans™ and interactors.

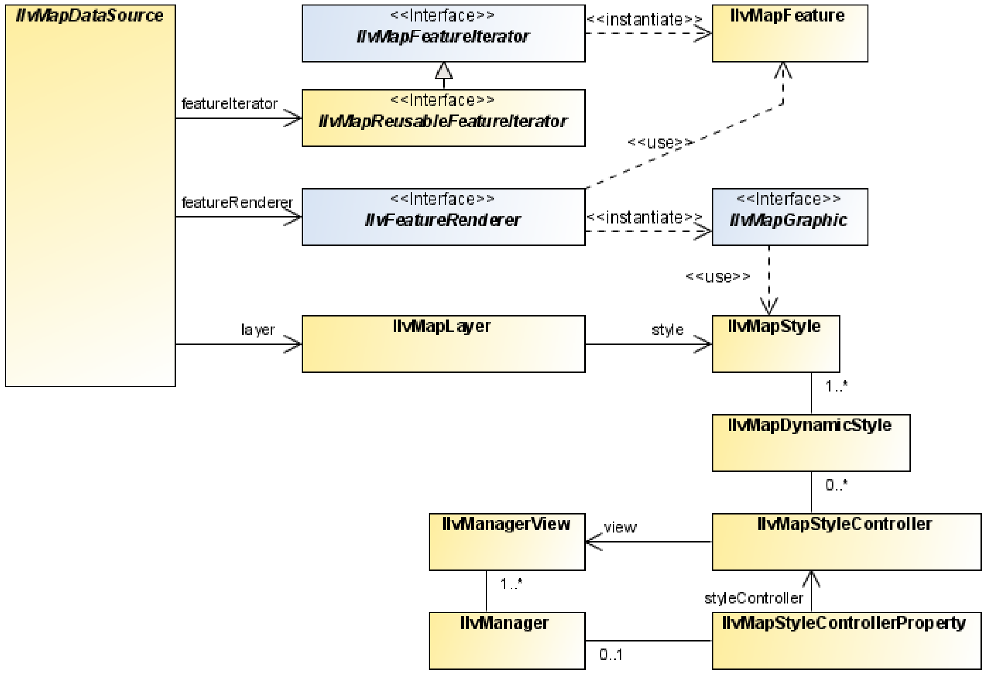
Geodetic computation and date line wrapping
Describes the class for geodetic calculations.

Utilities
Describes the class that provides utility methods.

Reader framework

Class diagram

The class diagram for the reader framework is shown in *Reader Framework UML Diagram*.



Reader Framework UML Diagram

JViews Maps provides a set of classes that you can use to read data from various cartographic data sources (files, databases, map servers, and so on), create map features, transform the features into JViews Maps graphic objects using renderers, and position them correctly onto an existing map.

The IlvMapFeature class

The `IlvMapFeature` class in the package `ilog.views.maps` is the base class for map features. This class allows you to read in data for cartographic display from source files. A map feature can be, for example, a segment of road, an aerial image, the summit of a hill, or a digital terrain model. For more information about this class, see *Handling map features*.

The IlvMapFeatureIterator interface

The `IlvMapFeatureIterator` interface in the package `ilog.views.maps` is the common interface for readers. All the classes that implement this interface can be used to read cartographic data, whatever the original format. JViews Maps provides a number of predefined readers, all of which, implement this interface. These readers are described in detail in *Readers and writers*.

The IlvFeatureRenderer interface

The `IlvFeatureRenderer` interface in the package `ilog.views.maps` is the common interface for renderers. All the classes that implement this interface can be used to translate an `IlvMapFeature` into a graphic object. JViews Maps provides a default class (`IlvDefaultFeatureRenderer`) implementing this interface and being able to render most of the `IlvMapFeature` returned by the JViews Maps predefined readers. A specific renderer can also be provided by a reader (method `IlvMapFeatureIterator.getDefaultRenderer()`).

The IlvMapStyle class

The `IlvMapStyle`class is a base class for the style used with `IlvMapGraphic` graphic objects. A single instance of `IlvMapStyle` can be shared by `IlvMapGraphic`s that can read their graphic (or other) attributes from the style.

The `IlvMapStyle` class should be used in conjunction with an `IlvMapLayer`. In this case, the `IlvMapLayer` applies the style to the objects within that layer. This can be used to change the appearance of a layer dynamically without reloading the map. Some attributes are layer specific, such as, layer visibility, and layer transparency

The IlvMapGraphic interface

Graphic objects requiring that they read their graphic attributes from a `IlvMapStyle`, must implement the `IlvMapGraphic` interface. Objects of this class are rendered with `IlvMapAreaRenderer`, `IlvMapPointRenderer`, `IlvMapCurveRenderer` and `IlvMapTextRenderer`. These renderers are used by the data sources provided by JViews Maps.

The IlvMapDynamicStyle class

The package `package-frame` contains classes used to change dynamically the style of a layer when the scale changes.

The `IlvMapDynamicStyle` associates an `IlvMapStyle` and a scale, and is used with a `IlvMapStyleController`. If the controller is installed on the view of the map, it listens for changes to the scale of the map and selects the appropriate style to apply to the appropriate layer. This allows a map to have different appearances at different scales.

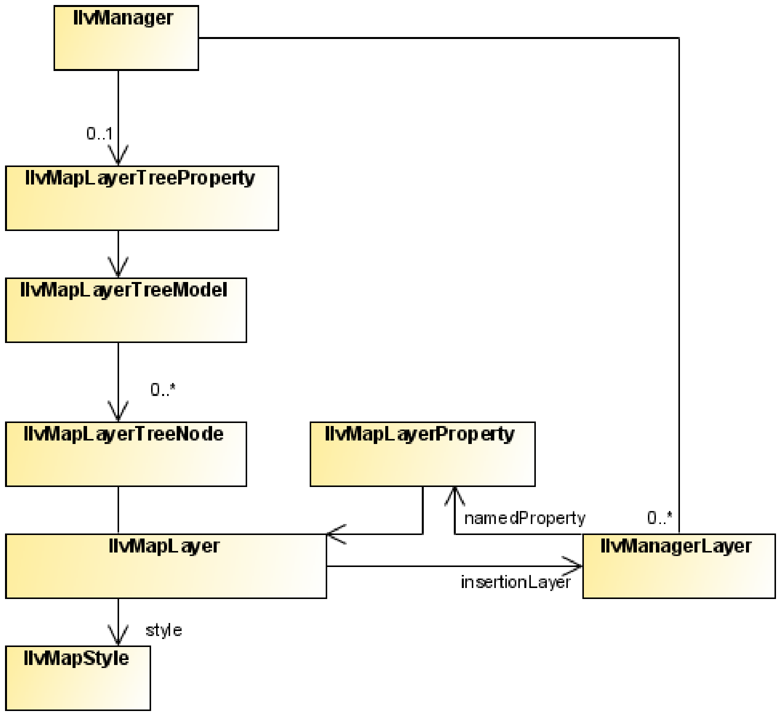
Map layers and map styles

Describes map layers and map styles.

In this section

Introduction to layers and styles

The class diagram for layers and styles is shown in *Layer and Style UML Diagram*.



Layer and Style UML Diagram

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **/jviews-maps86/samples/mapbuilder/ index.html**.

About map layers

The `IlvMapLayer` class represents a map layer, that is, a cartographic theme. It associates a style (`IlvMapStyle` or one of its subclasses) with an `IlvManagerLayer` containing graphic objects.

Map Layers are arranged in a tree structure, and stored in the manager using the `IlvMapLayerTreeProperty`, see *Map layer tree*.

Each map layer is attached to an `IlvMapLayerTreeNode`, which contains the necessary information on parent and child layers. You can use this node to step through layer hierarchy as follows:

```
String SVGpath = "C:/maps/map.svg";
IlvSVGDataSource source = new IlvSVGDataSource(SVGpath);
source.setManager(getView().getManager());
source.setDestinationBounds(lonMinRad,latMinRad,lonMaxRad,latMaxRad);
```

Note: Alternatively you can use a tailored transformation by calling IlvSVGDataSource.setInternalTransformation.

```
source.start();
IlvMapLayerTreeNode node = mapLayer.getNode();
for (int i = 0; i < node.getChildCount(); i++) {
  IlvMapLayerTreeNode child=(IlvMapLayerTreeNode) node.getChildAt(i);
  IlvMapLayer childLayer = (IlvMapLayer) node.getUserObject();
  ... do something with child layer
}
```

Most map layers are attached to a data source, which is responsible for populating the manager layer with the correct graphic objects when the map data is loaded (possibly on-demand-loading) and reprojection times. The exception to this rule is the composite layer. Composite layers are only used to group a set of sub-layers and manage their styles using attribute inheritance (See below).

About map styles

Every style in JViews Maps is a subclass of the base `IlvMapStyle` class.

This class provides access to a set of attributes, usually also accessible by a setter/getter pair, depending on each style subclass.

For example, you can change the view visibility setting using on of the following:

```
style.setAttribute(IlvMapStyle.VISIBLE_IN_VIEW,Boolean.TRUE);
```

```
style.setVisibleInView(true);
```

You can catch any change in the map style by writing and registering a listener on it. For example:

```
StyleListener listener = new StyleListener() {
   public void styleChanged(StyleEvent event) {
      if(IlvMapStyle.ALPHA.equals(event.getAttribute())) {
         //  ... do something when transparency changes
      }
   }
};
...
myStyle.addStyleListener(listener);
```

Style hierarchy

Styles form a hierarchy and attribute values can be inherited through this hierarchy. If a style attribute is inherited from a parent style, that parent attribute value is used when displaying objects using that style.

Although not enforced in the API, it is recommended that you make the style hierarchy the same as the map layer hierarchy. This can be done when the map layer is inserted in the layer tree model:

```
IlvMapLayerTreeModel ltm =
IlvMapLayerTreeProperty.GetMapLayerTreeModel(manager);
ltm.addChild(parentLayer, layer);
IlvMapStyle parentStyle = layer.getParent().getStyle();
IlvMapStyle childStyle = layer.getStyle();
childStyle.setParent(parentStyle);
```

Common styling properties

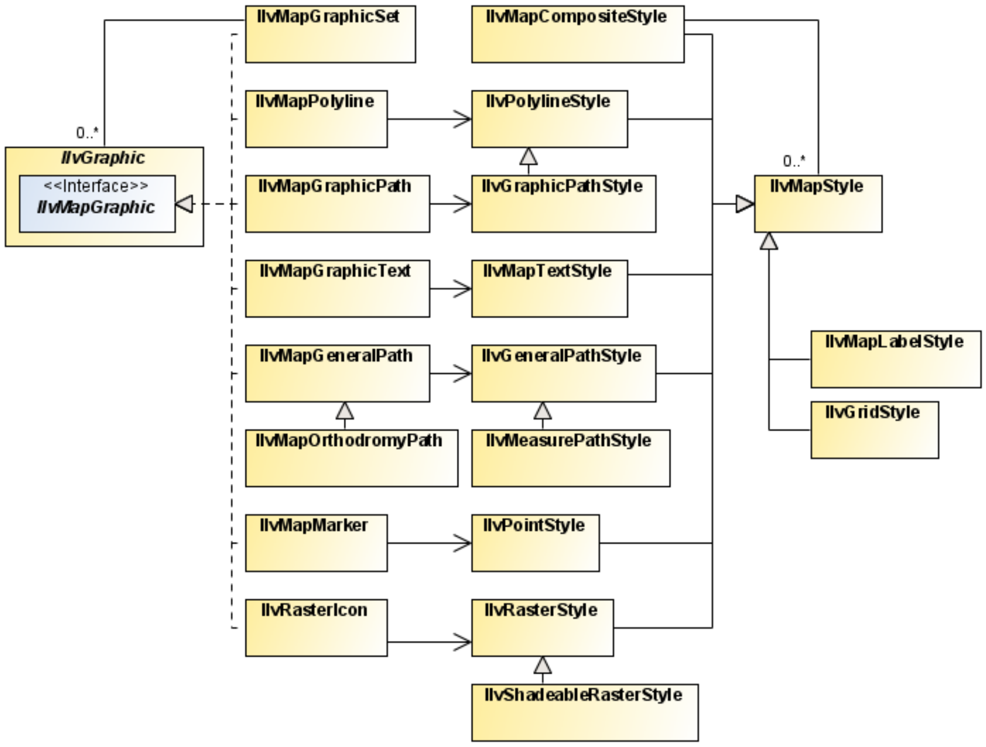
Whatever the type of layer, its style always has the following base properties.

Common Styling Properties

Property name	Contents
VISIBLE_IN_VIEW	Indicates whether the `IlvManagerLayer` is displayed on the map view.
VISIBLE_IN_OVERVIEW	Indicates whether the `IlvManagerLayer` is displayed on the map overview.
ATTRIBUTE_INFO	Contains the `IlvAttributeInfoProperty` used to describe all object properties. This is used to provide the list of possible property names displayed in the label attribute check box. This attribute cannot be changed by the user in the map layer tree panel.
LABEL_ATTRIBUTE	Contains either a null value, or the name of the property used when labeling this map layer (chosen usually in the list provided by the ATTRIBUTE_INFO attribute).
ALWAYS_ON_TOP	Indicates whether the attached map layer is placed on a normal or superimposing plane. This should be used only for overlay layers such as grids, labels, measures, and so on.
LEGEND_GROUP	A logical name, used and displayed to group map layers in the legend.
CATEGORY	An identifying name to group more than one layer on the same legend line.
ALPHA	The level of transparency of the manager layer.

Map layers for graphic objects

The class diagram for map layers for graphic objects is shown in *Layers for Graphic Objects UML Diagram*.



Layers for Graphic Objects UML Diagram

Most map layers are used to manage graphic objects created from map features imported from various map data files. The styling options differ according to the content of each file and the map features imported by the different readers. For example, you do not style a polygon the same way you do an altitude raster image.

The IlvGeneralPathStyle class

The `IlvGeneralPathStyle` class is used for `IlvMapGeneralPath` stylable graphic objects. Generally, most vectorial map renderers provide a `setUsingGeneralPath` method to allow a choice between using general paths, which give a better aspect and capabilities, or regular polygons, which provide better performance.

General Path Styling Properties

Property name	Content
FILL_PAINT	`Paint` object used to fill the shape.
STROKE_PAINT	`Paint` object used when stroking the path.
FILL_ON	`Boolean` indicating whether the inside of the object is filled.
STROKE_ON	`Boolean` indicating whether the shape of the object is stroked.
PAINT_ABSOLUTE	`Boolean` indicating whether the fill paint is adapted to the bounding rectangle of the object.
STROKE_WIDTH	Stroke width.
PAINT_ZOOMED	`Boolean` indicating whether the paint is zoomed according to the shape when the object is zoomed.
END_CAP	Type of decoration applied to the ends of unclosed subpaths (see `java.awt.BasicStroke`).
LINE_JOIN	Type of decoration applied at the intersection of two path segments (see `java.awt.BasicStroke`).
LINE_STYLE	Float table indicating how to make a dash pattern by alternating between opaque and transparent sections (see `java.awt.BasicStroke`).
STROKE	Object used for stroking the path. This attribute is dependant on all stroke attributes and is recomputed when one of them changes.This attribute is not visible in the Map Layer Tree Panel.

The IlvPolylineStyle class

The `IlvPolylineStyle` class is used for `IlvMapPolyline` stylable objects (non-filled polygons or polylines).

Polyline Styling Properties

Property name	Content
FOREGROUND	`Color` object used when drawing the polygon borders.
LINE_WIDTH	Polygon borders line width.
END_CAP	Type of decoration applied to the ends of unclosed subpaths (see `java.awt.BasicStroke`).
LINE_JOIN	Type of decoration applied at the intersection of two path segments (see `java.awt.BasicStroke`).
LINE_STYLE	Float table indicating how to make a dash pattern by alternating between opaque and transparent sections (see `java.awt.BasicStroke`).
DECORATION	Additional `IlvPathDecoration` object used for stroking the polygon.
DECORATION_ONLY	`Boolean` indicating whether only the decoration is displayed.
DECORATION_PAINT	Paint color of the decoration. By default (when this attribute is null), the decoration is colored using the FOREGROUND attribute.

The IlvGraphicPathStyle class

The `IlvGraphicPathStyle` class is a subclass of `IlvPolylineStyle` and so provides all the attributes in *The IlvPolylineStyle class*. It provides the style for `IlvMapGraphicPath` objects (filled polygon areas), by adding the following:

Graphic Path Styling Properties

Property name	Content
PAINT	`Paint` object used to fill the polygon area.
DO_FILL	`Boolean` indicating whether the inside of the polygon is filled.
DO_STROKE	`Boolean` indicating whether the borders of the polygon are stroked.

The IlvPointStyle class

The `IlvPointStyle` class controls the style of map point (`IlvMapPoint`) objects. It provides the following attributes:

Point Styling Properties

Property name	Content
FOREGROUND	`Color` object used for displaying the point.
MARKER_SIZE	The size of the marker.
MARKER_TYPE	The type of marker to use.

The IlvMapTextStyle class

The `IlvMapTextStyle` class controls the style of map stylable label objects (`IlvMapText`). It provides the following attributes:

Map Text Styling Properties

Property name	Content
ANTIALIASING	`Boolean` indicating whether text should be specifically anti-aliased (even if the view setting itself is different).
ATTACHMENT	The label attachment.
FILL_PAINT	`Paint` object used to fill the characters of the text.
FONT	The label `Font`.
FRAME_PAINT	`Paint` object used to draw the frame of the label.
INNER_MARGIN	Spacing between the frame of the label and the text.
INTERLINE	Spacing between two lines of text.
MAXIMUM_HEIGHT	The maximum height of the label.
MINIMUM_HEIGHT	The minimum height of the label.
STROKE_PAINT	`Paint` object used to draw the shape of the text character.
BACKGROUND_PAINT	`Paint` object used to fill the frame of the label.
ALIGNMENT	Alignment of the multi-line text.

The IlvRasterStyle class

The `IlvRasterStyle` class is used with `IlvRasterIcon` graphic objects, and provides attributes to control the appearance of the image:

Raster Styling Properties

Property name	Content
BRIGHTNESS	A `Double` percentage applied to the color model to make the entire image brighter or darker. 0% gives a black image.
CONTRAST	A `Double` percentage applied to the color model to increase the color difference. 0% gives a completely grey image.
SATURATION	A `Double` percentage applied to the color model of the image to change the color saturation. 0% gives a completely grey image.
COLOR_MODEL	The `ColorModel` object used to transform the pixel values of the image into RGBA colors.

Composite layers

The IlvMapCompositeStyle class

The `IlvMapCompositeStyle` class does not define any additional attributes. Its attribute list grows with its child list.

When adding a child style in a composite style, all the attributes of that child are added (if they did not exist) into the parent style. Those values can then be inherited for each child style – allowing the attributes for all children to be changed in one single place.

You can change the inheritance setting of each attribute by calling the `setInherited(java.lang.String, boolean)` method.

Label layer

Label layers are created automatically on demand by the `IlvMapLabeler` property installed on the manager when the LABEL_ATTRIBUTE style attribute is changed on a registered layer.

The IlvMapLabelStyle class

The `IlvMapLabelStyle` class is used by the `IlvMapAreaLabel`, `IlvMapLineLabel` and `IlvMapPointLabel` classes, and controls the labeling parameters:

Map Label Styling Properties

Property name	Content
FOREGROUND	The `Color` used to display labels.
LABEL_FONT	The `Font` of the labels.
LABEL_STROKE	The `Stroke` object used to display label outlines (not visible to users).
OUTLINE_COLOR	The `Color` of the label outline.
DRAW_OUTLINE	`Boolean` indicating whether the outline is displayed.
LABELLING_SMALL_AREAS	`Boolean` indicating whether areas too small to enclose their label are labeled.
FOLLOW_PATH	`Boolean` indicating whether labels on polygons follow those polygons or are placed horizontally.

Grid layers

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>* /jviews-maps86/samples/mapbuilder/ index.html**.

The IlvAbstractBaseGrid class

The `IlvAbstractBaseGrid` base class has subclasses `IlvMGRSGrid` and `IlvLatLonGrid`. It is a manager layer that, instead of drawing graphic objects (`IlvGraphic` instances) added to it, displays a grid adapted to the current zoom level of the map.

This base grid also implements the `IlvManagerViewDecoration` interface, which allows you to display the grid on the screen, but prevents it from being displayed on printed material.

The IlvMGRSGrid class

The `IlvMGRSGrid` class displays a set of auto-adaptive MGRS standard grids and labels on top of a geographic view. This grid displays a list of Grid zones (either UPS or UTM zones), and their sub grids (100000m, 10000m, 1000m). Each grid or sub-grid is labeled using the standardized MGRS name for the current area.

You can create and add an MGRS grid on any geo-referenced manager view (that is, any view that supports the `IlvCoordinateSystemProperty`). For example, to add an MGRS grid containing all MGRS zones as a decoration of the view:

```
IlvMGRSGrid grid = new IlvMGRSGrid();
IlvMGRSGridZone.addAllZones(grid);
view.addViewDecoration(grid);
```

The IlvLatLonGrid class

The `IlvLatLonGrid` class displays a set of auto-adaptive grids and labels along latitude or longitude lines on top of a geographic view. If the grid is auto-adaptive, the step between each successive lat/lon line is dependant on the scale of the current view.

You can create and add an MGRS grid on any geo-referenced manager view (that is, any view that supports the `IlvCoordinateSystemProperty`). For example, to create a lat/lon grid layer:

```
view.getManager().addLayer(new IlvLatLonGrid()-1);
```

By default, the `IlvLatLonGrid` creates only the points at the corners of each grid square. If you are using a coordinate system that transforms the map in a non-linear way (such as Orthographic projection, UTM projection, and so on), you can increase the number of intermediate points on each grid square in order to show a smoother version of the grid:

```
lgrid.setSmoothness(4);
```

Writing specific grids

The easiest way to implement your own grid system is to start with an empty MGRS grid and then add your own zones to it. For examples of this, see the Map Builder `GridManager` class.

Integrating grids into map layers

As the grids are implemented as manager layers, you only need to connect these to an `IlvMapLayer`.

This map layer must be created, styled and integrated in the map layer tree model:

```
IlvMapLayerTreeModel ltm =
IlvMapLayerTreeProperty.GetMapLayerTreeModel(view.getManager());
 IlvMapLayer mapInsertionLayer = new IlvMapLayer();
 mapInsertionLayer.setStyle(new IlvGridStyle());
 ltm.addChild(null, mapInsertionLayer);
```

Then you can integrate the grid manager layer into it:

```
mapInsertionLayer.insert(grid);
```

You can also use an `IlvDelayedDecoration` manager layer to encapsulate the grid. Whenever the user moves or zooms the grid rapidly, a simplified version of the grid is used, which displays more quickly. When the user stops moving or zooming the grid, the full grid is displayed:

```
IlvDelayedDecoration delayedGrid = new IlvDelayedDecoration(200);
delayedGrid.setDecoration(grid);
   mapInsertionLayer.insert(delayedGrid);
```

The IlvGridStyle class

The `IlvGridStyle` class is used when displaying grids. It defines the following attributes:

Grid Styling Properties

Property name	Content
GRID_COLOR	The `Color` used to display grid lines.
TEXT_FONT	The `Font` of grid labels.
TEXT_COLOR	The `Color` used to display grid labels.
OUTLINE_COLOR	The `Color` of the outline of the grid labels.

Map scales and layer styles

As a result of the use of the `IlvMapStyleControllerProperty` property of the manager (and its underlying `IlvMapStyleController`), layer styles are dependant on the current scale of the map. Every style attribute of a map layer can be different at different scales.

One of the most widely used attribute changes is the VISIBLE_IN_VIEW attribute, which is used to show and hide different map layers at particular scales. Using the same mechanism, your users can also change colors, line thickness, decorations and so on. Whenever the scale changes, the scale controller can change the current map layer styles, if required. This change of style usually triggers a view repaint, because the attributes of the objects have changed.

Some objects also listen to these style changes in order to re-render themselves. For example, if the color model of a raster style changes, all the images that depend on the style need to be recreated.

You can add dynamic styles with API calls such as:

```
IlvMapStyleController themeControl=
  IlvMapStyleControllerProperty.GetMapStyleController(view.getManager());
themeControl.addTheme(0.001,mapLayer,"new style");
themeControl.getStyle(mapLayer,0.001).setVisibleInView(true);
```

When you add dynamic styles, be careful to ensure that you still manage style inheritance. When adding a new map layer into a parent layer, you can do this as follows:

```
IlvMapDynamicStyle []t=themeControl.getThemes(mapLayer);
for (int i = 0; i < t.length; i++) {
   t[i].getStyle().setParent(parentLayer.getStyle());
}
```

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Map attribute filters

A IlvMapAttributeFilter object is used to compute the value of a styling attribute from a value found in an IlvGraphic instance. Typically, this value is retrieved from the IlvFeatureAttributeProperty named property attached to a IlvGraphic object. To install such a filter, you set it to an IlvMapStyle object.

The following example shows a custom filter class that defines the foreground of map graphic objects according to their "VALUE" feature attribute.

```
/**
 * Computes the foreground from a graphic's IlvFeatureAttribute
*/
class ColorAttributeFilter implements IlvMapAttributeFilter {
  /**
   * Method that returns the new color computed from the "VALUE"
   * feature attribute. If DEFAULT_VALUE is returned, the style value on which

   * the filter is installed will not be affected.
   */
  public Object get(IlvGraphic g, String attributeName) {
    if(IlvPolylineStyle.FOREGROUND.equals(attributeName)) {
     IlvAttributeProperty p = (IlvAttributeProperty)
       g.getNamedProperty(IlvAttributeProperty.NAME);
     if(p == null)
       return DEFAULT_VALUE;
     Object o = p.getValue("VALUE");
     Object ret = convertObjectToColor(o);
     if(ret == null)
       return DEFAULT_VALUE;
     return ret;
   }
}
```

The following code example shows how to install a custom filter class in a map layer style.

```
IlvMapAttributeFilter filter = new ColorAttributeFilter();
IlvMapLayer layer = getLayer();
IlvMapStyle style = layer.getStyle();
style.setAttributeFilter(filter);
```

Once the filter is installed, each request to retrieve an attribute value is passed to the get method of the filter.

Map-specific manager properties

Describes the classes available for managing various aspects of maps, such as altitude, display, data sources, layers, threads, and labeling.

In this section

Persistent symbol model

Describes the property used to make symbols persistent.

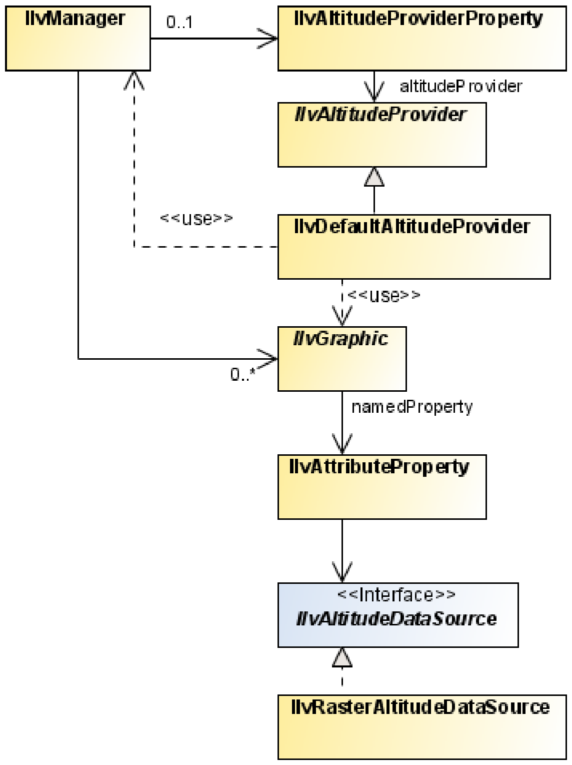
Altitude management

Describes the altitude management classes and the use of this property.

In this section

Altitude management classes

The class diagram for altitude management is shown in *Altitude Management UML Diagram*.



Altitude Management UML Diagram

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Using the altitude provider property

Terrain analysis computations are based on the `IlvAltitudeProviderProperty` of the manager and its underlying `IlvAltitudeProvider`, which is responsible for providing altitudes for each point on a map.

The `IlvJMouseCoordinateViewer` Bean uses altitude property information to provide altitude information whenever the mouse is over an altitude providing map object. Many other features of JViews Maps also use this information.

To access the altitude provider:

1. You can access the altitude provider by calling:

   ```
   IlvAltitudeProvider provider =
   IlvAltitudeProviderProperty.GetAltitudeProvider(manager);
   ```

   If you do not set a specific provider to this property, this method creates or returns an instance of `IlvDefaultAltitudeProvider`.

   This provider supports GTOPO30 and DTED format data sources.

2. To integrate another Digital Elevation Model, which provides the graphic objects with an `IlvAltitudeDataSource` property, see **To retrieve the altitude attached to a graphic object** below.

If you have read an image containing elevation data with an `IlvRasterAbstractReader`, for an example, see *Writing a raster reader for DEM data*; you can reuse its altitude information to provide altitude data by using an `IlvRasterAltitudeDataSource` instance.

To attach an altitude data source to a graphic object:

1. You first need to decide on the structure of the attribute property, for example, for a property containing only altitude data:

   ```
   IlvAttributeInfoProperty info = new IlvAttributeInfoProperty(
   new String[] { "myAltitudeDataSourcePropertyName" },
   new Class[] { IlvRasterAltitudeDataSource.class },
   new boolean[] { true });
   ```

2. You can then reuse this structural information with different altitude data sources, and set it as the graphic object property:

   ```
   IlvFeatureAttribute value[] = { new
   IlvRasterAltitudeDataSource(rasterImageReader, imageIndex) };
   graphic.setNamedProperty(new IlvFeatureAttributeProperty(info, value);
   ```

If you use the default altitude management described in *Using the altitude provider property*, you can also retrieve the altitude attached to an object as described.

To retrieve the altitude attached to a graphic object:

1. Get the attribute properties of the graphic object:

```
IlvAttributeProperty property = (IlvAttributeProperty)
graphic.getNamedProperty(IlvAttributeProperty.NAME);
```

2. You should find the altitude data source in that property:

```
IlvAltitudeDataSource ads =
(IlvAltitudeDataSource)property.getValue
("myAltitudeDataSourcePropertyName")
;
```

3. As the data source object provides only altitude information for a specific latitude/longitude pair, you may need to transform the coordinates into latitude and longitude. Here is an example that converts a mouse location into such a pair:

```
// transform the mouse point into manager coordinates
IlvPoint pt=new IlvPoint(mouseLocation.x,mouseLocation.y);
view.getTransformer().inverse(pt);
IlvProjectionUtil.invertY(pt);
try {
  // compute the coordinate transformation from manager coordinates to
lat/
lon
  IlvCoordinateSystem cs =
IlvCoordinateSystemProperty.GetCoordinateSystem(view.getManager());
  IlvCoordinateTransformation ct =
IlvCoordinateTransformation.CreateTransformation(cs,

IlvGeographicCoordinateSystem.KERNEL);
  // transform the point into lat/lon
  IlvCoordinate c = new IlvCoordinate(pt.x, pt.y);
  ct.transform(c, c);
  // retrieve the altitude
} catch (IlvCoordinateTransformationException e) {
}
```

4. You can then obtain the altitude. You should check if its value is a valid `double`, because the default data source and default provider of the manager return a `Double.NaN` value when there is no altitude information available.

```
    double alt = ads.getAltitude(c.x, c.y, 0);
    if(!Double.isNaN(alt)){
       return alt;
    }
```

If the pixel values stored in the `IlvRasterMappedBuffer` are altitudes, you can use the `IlvRasterAltitudeDataSource` class directly as the altitude provider.

Use the `IlvRasterAltitudeDataSource` class in the `IlvFeatureAttributeProperty` of every image that the reader creates.

To set altitudes use:

♦
```
public IlvFeatureAttributeProperty getProperties(int imageIndex)
      {
      IlvAttributeInfoProperty info = new IlvAttributeInfoProperty(
      new String[] { " myAltitudeDataSourcePropertyName" },
      new Class[] { IlvRasterAltitudeDataSource.class },
      new boolean[] { true });
        IlvFeatureAttribute values[] = new IlvFeatureAttribute[] {
       new IlvRasterAltitudeDataSource(this,imageIndex)
       };
      return new IlvFeatureAttributeProperty(info, values);
      }
```

See also To attach an altitude data source to a graphic object.

You need to provide JViews Maps with a way of knowing where the resulting image is placed. This is done through two methods that return the transformation and coordinate system used in the raster property boundaries.

To manage coordinates:

♦ Provide two methods as shown in the following example, which assumes that the bounds are given in degrees.

```
private static IlvCoordinateTransformation INTERNAL =
IlvCoordinateTransformation.CreateTransformation
   (IlvGeographicCoordinateSystem.KERNEL,
      IlvGeographicCoordinateSystem.WGS84);
public IlvCoordinateSystem getCoordinateSystem() {
   return INTERNAL.getTargetCS();
   }
public IlvMathTransform getInternalTransformation(int imageIndex) {
   return INTERNAL.getTransform();
}
```

Writing a raster reader for DEM data

To read a new Digital Elevation Model (DEM format), you should write a subclass of `IlvRasterAbstractReader` with an `addMap` method. If you want to take advantage of map load and save features, you have to manage the serialization of the reader, that is, provide methods that perform the required functions.

To create the reader:

1. Load a list of raster DEM files as shown in the following example.

```
public class MyDEMReader extends IlvRasterAbstractReader {
    // list of files to be read.
    private ArrayList filenameList = new ArrayList();
    /** default constructor */
    public MyDEMReader() {
    }
```

2. Write an `addMap` method to compute the `IlvRasterProperties` and `IlvRasterMappedBuffer` attached to the file name, see *Raster image management*, and then add this information to the list managed by the reader. As this step is heavily dependent upon format, only a summary is provided here.

```
public void addMap(final String filename) throws IOException {
     IlvRasterProperties loadingRaster = read/compute raster properties ...
     IlvRasterMappedBuffer source= read/compute raster pixel values...
    loadingRaster.setBaseName(filename);
    // to retrieve the file name when serializing data.
    addRaster(loadingRaster, source);
      }
```

To use the map load and save features:

1. Serialize all the necessary information to rebuild the images - in this example, only the filenames.

```
    public void write(IlvOutputStream stream) throws IOException {
        super.write(stream);
        int imageCount = getImageCount();
        for (int i = 0; i < imageCount; i++) {
            IlvRasterProperties props=getRasterProperties(i);
            stream.write("filename"+i,props.getBaseName());
        }
    }
```

2. Rebuild the reader from serialized data. Because the image data may have been saved in an IMG file associated with the map, you should only read the filenames of the raster DEM not the files themselves:

```
 public MyDEMReader(IlvInputStream stream) throws IlvReadFileException {

      super(stream);
      try {
             for(int count=0;true;count++) {
              String filename = stream.readString("filename"+count);
              filenameList.add(filename);
          }
      } catch (IlvReadFileException e1) {
          // No more filenames to read
      }
    }
```

3. If the complete map data is saved, the raw image data is reconnected by standard mechanisms. However, to reload files when the user has only saved the description of the map and not its data, write a reload method:

```
    public void reload(IlvThreadMonitoringData monitorInfo) {
        super.reload(monitorInfo);
        // clear all images
        dispose();
        // save the known filenames in a temporary array - the addMap 
would
else add them again.
       String[] filenames = (String[])filenameList.toArray(new String[0]
);
        // clear the file name list
        filenameList.clear();
        for (int i = 0; i < filenames.length; i++) {
            try {
                // load each file
                addMap(filenames[i]);
                if (monitorInfo != null) {
                    // update the thread monitoring information, if 
necessary
                    int percent = Math.round(i/(float)filenames.length
                    * 100);
                    monitorInfo.updateProgress(percent);
                }
            } catch (IOException e) {
                new IlvExceptionMessage(e,null);
            }
        }
    }
```

The display preferences property

Map display preferences are accessible through the `IlvDisplayPreferencesProperty` of the manager and its underlying `IlvDisplayPreferences`, which is responsible for indicating user preferences when displaying the map.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The display preferences are used to share:

- ♦ The preferred unit and format to use for altitudes.
- ♦ The preferred unit and format to use for distances.
- ♦ The preferred coordinate formatter to use to show earth coordinates.
- ♦ An indication of whether geodetic computation is activated or not.

Display preferences access

You can access the display preferences by calling:

```
IlvDisplayPreferences pref =
IlvDisplayPreferencesProperty.GetDisplayPreferences(manager);
```

This method creates or returns the last instance set of `IlvDisplayPreferences`.

Display preferences uses

The `IlvJMouseCoordinateViewer` Bean uses preferences to format coordinates and altitude information whenever the mouse moves over the map.

The `IlvJAutomaticScaleBar` also listens to this property in order to adapt the map distance unit to the preferences of the application.

When creating measurements (or an `IlvMapOrthodromyPath`), a specific `IlvDistanceAttribute` property is attached to contain the measurement length. This property displays itself taking into account the preferred distance unit found in the preferences.

The `IlvCoordinatePanelFactory` needs to retrieve a coordinate formatter to know how to display the coordinates selected on the view, which can also edited by the user. You should usually do this by retrieving the coordinate formatter of the current preferences:

```
IlvDisplayPreferences prefs =
IlvDisplayPreferencesProperty.GetDisplayPreferences(manager);
IlvCoordinateFormatter formatter= prefs.getCoordinateFormatter();
JPanel coordPicker=new
IlvCoordinatePanelFactory.CoordPointInputPanel(view,formatter);
```

Adding a listener to changes in display preferences

You can listen to changes in this property (such as those triggered by the Display Preference Editor), by adding a named property listener on the manager:

```
manager.addNamedPropertyListener(new NamedPropertyListener() {
  public void propertyChanged(NamedPropertyEvent event) {
    if(event.getNewValue() instanceof IlvDisplayPreferencesProperty){
      IlvDisplayPreferencesProperty 
prop=(IlvDisplayPreferencesProperty)event.getNewValue();
      IlvDisplayPreferences preferences=prop.getDisplayPreferences();
      // manage the new preferences
      ...
    }
  }
});
```

Geodetic computation

All data sources should use the display preferences properties to adapt their rendering to the activation of geodetic computation, which is an important user choice.

When geodetic computation is activated, every polygon-like map feature is rendered, through the use of an `IlvGeodeticPathComputation`, into a series of orthodromies (see orthodromy measure). Whenever a segment of the polygon has two extremities on different sides of the screen, the polygon is cut into many areas, separated on the screen but representing the same map feature.

The data source property

Describes the data source classes and use of this property.

In this section

Data source tree

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

IlvMapDataSourceProperty

The `IlvMapDataSourceProperty` class is a named property used to attach a `IlvMapDataSourceModel` to an `IlvManager`.

`IlvManager`This data source model controls all data sources that you import into the manager. It is also persistent data that is saved when you save the map.

Accessing the data source model

You can access the map data source model by calling:

```
IlvMapDataSourceModel dsm =
IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
```

Adding a listener

As the `IlvMapDataSourceProperty` is a named property of the manager, you can add a listener that is called whenever the property as a whole is changed, for example:

```
manager.addNamedPropertyListener(new NamedPropertyListener() {
  public void propertyChanged(NamedPropertyEvent event) {
    if (event.getPropertyName().equals(IlvMapDataSourceProperty.NAME)) {
      IlvMapDataSourceProperty p = (IlvMapDataSourceProperty)
event.getNewValue();
      if (event.getType() == NamedPropertyEvent.PROPERTY_SET) {
  ...do something
      }
    }
  }
});
```

Controlling data sources

You can allow users to edit and control individual data sources by adding an `IlvDataSourcePanel` Bean to your application. See *Using the GUI beans*.

For more information about the data source tree, see *Creating data source objects*, *Using data sources*, and *Developing a new data source*.

Note: Although the Data Source Tree contains a Tree structure, JViews Maps uses only a single level tree, resulting in a simpler data source list. Future versions, or user applications, may use the full tree structure for advanced features.

Reloading all data sources

If you have created an instance of `IlvSDMEngine` (even empty) in your manager.

To reload the data, you can simply write:

```
engine.loadData();
```

In case you do not want, or do not need, a symbol management utility, you can use the following code to reload your data:

To reload without using an IlvSDMEngine

1. Retrieve the data source model `root`:

   ```
   DefaultMutableTreeNode root = (DefaultMutableTreeNode) dsm.getRoot();
   ```

2. Retrieve the map layers attached to each of the data sources:

   ```
   int count = root.getChildCount();
   for (int i = 0; i < count; i++) {
    DefaultMutableTreeNode node = (DefaultMutableTreeNode) root.getChildAt
   (i);
    IlvMapDataSource source = (IlvMapDataSource) node.getUserObject();
    IlvMapLayer mlayer = source.getInsertionLayer();
   ```

3. Restart each data source ensuring that the tile manager updates the visible part of the view for data sources containing tiled layers:

   ```
    source.reset();
    source.start();
    IlvManagerLayer layer = mlayer.getManagerLayer();
    if (layer instanceof IlvTiledLayer) {
      ((IlvTiledLayer) layer).getTileController().updateView(getView());
    }
   ```

Map layer tree

The `IlvMapLayerTreeProperty` class stores an `IlvMapLayerTreeModel`.

This layer tree model controls the order and style inheritance of `IlvMapLayer` you add into your manager. This model is also persistent data that is saved when you save the map.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Accessing the map layer tree model

You can access the layer tree model by calling:

```
IlvMapLayerTreeModel ltm =
IlvMapLayerTreeProperty.GetMapLayerTreeModel(manager);
```

Controlling the map layer tree

You can allow users to edit and control individual layer styles by adding an `IlvLayerTree` Bean in your application, see *Using the GUI beans*.

Usually, applications only need to add layers with the `addChild(ilog.views.maps.beans.IlvMapLayer, ilog.views.maps.beans.IlvMapLayer)` method, or remove them with the `removeChild(ilog.views.maps.beans.IlvMapLayer)` method of the layer model.

For more information about the map layer tree, see *Creating data source objects*, *Using data sources*, and *Developing a new data source*

Adding a listener for layer organization changes

The map tree model is a subclass of `DefaultTreeModel`, so you can add a `TreeModelListener` to your application to trap any change in the layer organization.

Adding a listener for tree structure changes

As the `IlvMapLayerTreeProperty` is also a named property of the manager, you can add a listener that is called whenever the entire tree structure is changed, for example:

```
manager.addNamedPropertyListener(new NamedPropertyListener() {
  public void propertyChanged(NamedPropertyEvent event) {
    if (event.getPropertyName().equals(IlvMapLayerTreeProperty.NAME)) {
      IlvMapLayerTreeProperty p = (IlvMapLayerTreeProperty)
event.getNewValue();
      if (event.getType() == NamedPropertyEvent.PROPERTY_SET) {
  ...do something
      }
    }
  }
});
```

Adding code to all layers

If your application needs to apply a piece of code to all layers, whatever their depth in the tree, you should call the `getEnumeration()` method of the tree model, such as:

```
       Enumeration e = ltm.getEnumeration();
       while(e.hasMoreElements()) {
         Object o = e.nextElement();
         if(o instanceof IlvMapLayerTreeNode) {
           IlvMapLayer layer =
(IlvMapLayer)((IlvMapLayerTreeNode)o).getUserObject();
            ... act on the layer
       }
}
```

For example, the method `clearAllObjects()` in `IlvMapLayerTreeModel` calls the `removeAllObjects` for each layer found.

Thread monitoring

The `IlvThreadedActivityMonitorProperty` class is a named property used to attach a `IlvThreadedActivityMonitor` to an `IlvManagerIlvManager`.

This threaded monitor is a centralized object with which threads should register their activities. The activities, that is, tasks, must also notify this controller of their progress.

All activity listeners registered with this controller are notified in turn of any activity progress. The `IlvThreadedActivityMonitorPanel` is one such important listener. This Bean provides the user with feedback on activities that are currently running, see *Using the GUI beans*.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Accessing the threaded activity monitor

You can access the threaded activity monitor by calling:

```
IlvThreadedActivityMonitor mon =
IlvThreadedActivityMonitorProperty.GetThreadedActivityMonitor(manager);
```

Providing threaded activity information

It is the responsibility of each thread to register, update and un-register their activities to provide the user with correct information on the background tasks that are running:

```
mon.updateActivityProgress(activityID,10,"doing something long");
... do something long that takes 10% of total time...
mon.updateActivityProgress(activityID,20,"doing another thing");
...
```

Setting the activity progress to 100% or un-registering the activity removes it from the list currently managed by the monitor.

Registering objects as listeners

You can also register your own objects as listeners to changes in the activity monitor with lines of code such as:

```
mon.addActivityListener(new IlvThreadedActivityMonitor.ActivityListener() {
  public void activityChanged(ActivityEvent e) {
if(e.getEventType() ==
IlvThreadedActivityMonitor.ActivityEvent.ACTIVITY_REMOVED) {
 ... an activity just ended, do something.
  }
});
```

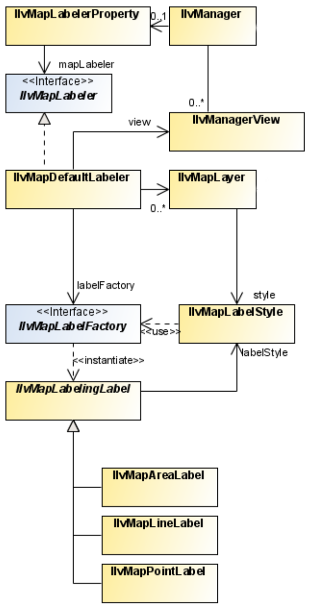
Map labeling

Describes the map labelling classes and the use of this property.

In this section

Map labeling classes

The class diagram for map labeling is shown in *Map Labeling UML Diagram*.



Map Labeling UML Diagram

JViews Maps has a dynamic map labeling mechanism. When activated for a specified map layer, all graphic objects of this layer are labeled in a separate thread. This ensures maximum responsiveness of the GUI while performing the background layout. The labeling process is done every time the user changes the view by zooming, scrolling, and so on.

Using the IlvMapLabeler interface

The entry point for the dynamic map labeling mechanism is the `IlvMapLabelerIlvMap` interface. A class implementing this interface is responsible for managing labels for a given `IlvManagerIlv`. The `IlvMapLabelerPropertyIlvMapLabeler` class is a named property used to attach an `IlvMapLabeler` to an `IlvManager`. This map labeler handles all the data sources imported into the manager. The model data is made persistent and saved when you save the map.

You can access the map labeler by calling:

```
IlvMapLabeler labeler = IlvMapLabelerProperty.GetMapLabeler(manager);
```

If a specific labeler is not set for the property, this method creates or returns an instance of `IlvMapDefaultLabelerIlvMapDefault`. This default labeler class automatically creates and configures a map layer for labels and listens for changes to the layer structure attached to the view. This allows the labeling to be updated when layer order changes. It also contains an internal `IlvMapLabelFactoryIlvMapLabel` to create the appropriate labels for graphic objects, according to the LABEL_ATTRIBUTE field of the `IlvMapLayerIlvMap`. For more information about `IlvMapLabelFactory`, see *The IlvMapLabelFactory Interface*.

To add labels to a given map layer on your map:

1. Define the layer to be labeled:

   ```
   IlvMapLayer layerToLabel;
   ```

2. Create a default `IlvMapDefaultLabeler` and set it on the manager:

   ```
   IlvMapLabeler labeler =
     IlvMapLabelerProperty.GetMapLabeler(manager);
   ```

3. Since this labeler has to interact with the view, you must indicate in which view it is to display the labels:

4. Set the label attribute for the map layer to specify which attribute of the graphic objects should be displayed as a label (check the available attributes in the file format you read in):

5. Register the map layer to label with the labeler:

6. Finally, notify the labeling thread to compute labels for all the labeled layers:

The IlvMapLabelManager class

The `IlvMapDefaultLabelerIlvMapDefault` class holds references to labeled layers (the layers to be labeled) and label layers (the layers that display the labels). This class also holds a reference to an `IlvMapLabelManagerIlvMapLabel`that acts as a controller for label rendering operations. The `IlvMapLabelManager` class monitors changes in the view (scrolling, zooming...), creates labels for all the visible graphic objects in the view that require labeling, and lays them out and draws them. This is all done in a separate thread so that user interaction is not blocked. Basically, this class contains the JViews Maps rendering engine.

The IlvMapLabelFactory Interface

To create labels for graphic objects, the `IlvMapLabelManagerIlvMapLabel` class relies on a label factory implementing the `IlvMapLabelFactoryIlvMapLabel` interface. Any object implementing this interface is responsible for returning the appropriate `IlvMapLabelingLabelIlvMapLabeling` instances for specified `IlvGraphicIlv` objects. This is done by implementing the method:

```
labeler.setView(view);
layerToLabel.getStyle().setLabelAttribute("NAME");
labeler.addLayer(layerToLabel);
labeler.performLabeling();
public IlvMapLabelingLabel[] getGisLabel(IlvGraphic comp);
```

The default label factory performs the following tasks:

♦ It extracts an attribute from the specified `IlvGraphicIlv` object (according to the label attribute in the `IlvMapStyle` of the `IlvMapLayerIlvMap` being labeled), and takes its `String` representation as the text of the label.

♦ Then it creates the appropriate instance of the `IlvMapLabelingLabelIlvMapLabeling` class, according to the kind of graphic object being labeled:

- `IlvMapPointLabel` for objects with an anchor point.
- `IlvMapLineLabel` for polyline objects (for example, label along the line).
- `IlvMapAreaLabel` for closed areas with a label placed within visible parts of the area.

The `IlvMapLabelManagerIlvMapLabel` uses this default implementation of the factory, but you can provide your own implementation if you want to have fine control over what is labeled, and how. To replace the default label factory with your own label factory:

```
IlvMapLabelFactory myLabelFactory = new myLabelFactory(); //Your own
implementation.

//Get the map labeler.
IlvMapLabeler labeler =
   IlvMapLabelerProperty.GetMapLabeler(manager);

//Set the factory, if the labeler is an IlvMapDefaultLabeler instance.
if(labeler instanceof IlvMapDefaultLabeler) {
   IlvMapDefaultLabeler dflt = (IlvMapDefaultLabeler)labeler;
   dflt.setLabelFactory(myLabelFactory);
 }
```

Controlling renderer parameters

When you implement your own `IlvMapLabelFactory`, you can specify a set of layout and rendering parameters for each instance of `IlvMapLabelingLabel` that you create in the constructor.

For more information about these parameters, see the `IlvMapPointLabel`, `IlvMapLineLabel`, and `IlvMapAreaLabel` classes.

Areas of interest

The `IlvAreasOfInterestProperty` class is a named property used to store areas of interest in an `IlvManager`.

For more information on this property see *Area of Interest panel*.

Coordinate system

The IlvCoordinateSystemProperty class is a named property used to store the geographic or coordinate system in which the map should be displayed. It can be different from the individual data source coordinate system in which case JViews Maps will automatically perform a reprojection of the data.

See the relevant data source documentation in the Readers and Writers section, for more information.

See *Coordinate System Editor* for examples of use.

Persistent symbol model

The `IlvPersistentSDMModelProperty` class is a named property used to make the symbols added through the `IlvSymbologyTreeView` API persistent. It is not compatible with the JViews Diagrammer designer way of symbol persistence through an xml description in a separate file.

See *Making the model persistent*.

Readers and writers

Introduces you to the predefined reader/writer classes supplied with JViews Maps.

In this section

The image file reader
Describes a generic image file reader.

The Oracle spatial reader and writer
Describes the Oracle® reader and writer classes, using data sources, how to use tiling, multithreading and exporting to an SDO database.

The GeoTIFF reader
Describes the classes that allow you to read GEO TIFF files.

The TIGER/Line reader
Describes the TIGER/Line® reader for maps related to US census data.

The DXF reader
Explains how to read DXF files.

The KML reader and writer
Describes the KML reader and writer and exporting KML files.

The DEM/GTOPO30 reader
Explains how to read GTOPO30 files.

The Web Map Server reader
Explains how to read images from a Web Map Server (WMS).

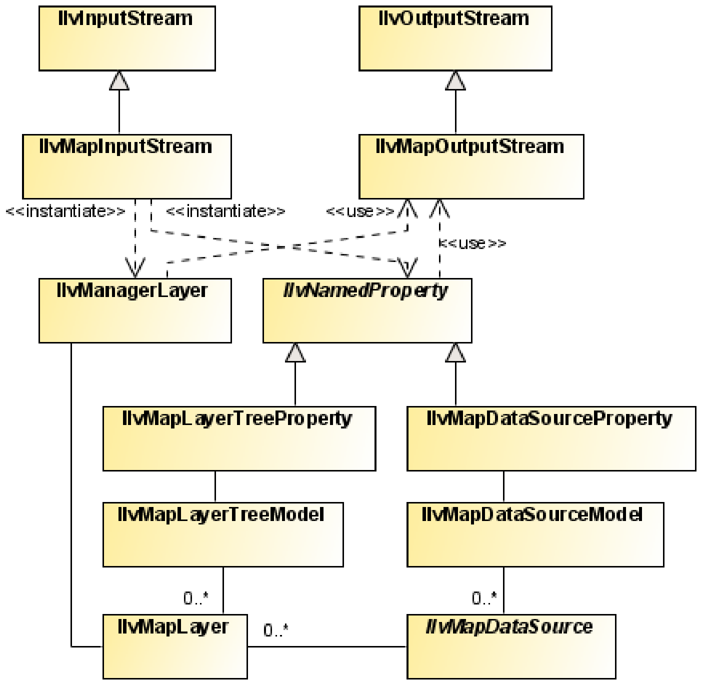
The SVG reader
Explains how to read SVG files.

Overview of readers and writers

The predefined reader/writer classes supplied with JViews Maps are defined in subpackages of the `ilog.views.maps.format` package.

The pivot format reader and writer

The class diagram for the pivot format reader and writer is shown in *Pivot Format Reader and Writer UML Diagram*.



Pivot Format Reader and Writer UML Diagram

Saved files

Saving a map can produce two files:

♦ A mandatory `.ivl` file, containing all information about the map and all map objects (unless only the theme is saved, see *Saving and reloading a map*).

♦ An `.img` file, containing the map raster data in a proprietary format for fast access when reloading. This file is not produced if the map does not contain an image.

The .ivl file

The `.ivl` format is the standard IBM® ILOG® JViews file format used to save an `IlvManager` object, see Saving and reading in *The Essential JViews Framework*. The file basically contains:

♦ The `IlvManagerLayer`s of the manager, which themselves contain all of the graphic objects.

♦ All additional `IlvNamedProperty` objects attached to the `IlvManagerLayer`s.

For example, in JViews Maps 8.1, the entire map model, and all related information, is stored in the `IlvManager` as named properties, see *Map-specific manager properties*, and saved in the `.ivl` file.

The .img File

To deliver the map reload performance currently achieved by JViews Maps, raster data for image objects is not saved in the `.ivl` file because this would slow down the `.ivl` file read time. Instead, all raster data is saved in a separate `.img` file. When reloading the map, this file can be mapped directly in memory (if allowed by the operating system) for fast access and virtually no parsing time.

Note that if the map does not contain an image, no `.img` file is generated with the `.ivl` file.

Finally, the `.ivl` and `.img` file extensions are defined by the Map Builder, but you are free to use the API to give any filename you want to the map you are saving. It is up to you to be consistent between the saving and loading processes.

Saving and reloading a map

In JViews Maps 8.1, maps can be saved in the proprietary `.ivl` file format. This is basically the regular IBM® ILOG® JViews format slightly modified to improve the loading performance of maps containing large amounts of raster data.

To save a map:

1. Create an `IlvMapOutputStream` object from a filename. This class extends the `IlvOutputStream` (see Input/output operations in *The Essential JViews Framework*) and provides the option of not saving the layers (and hence the graphic objects) contained in an `IlvManager`. You can choose to write the file in binary format, which is more compact, or in ASCII format, which is more readable.

   ```
   String mapFilename = "myMap.ivl";
   boolean binary = false;
   IlvMapOutputStream mapOutput = new IlvMapOutputStream(mapFilename, binary)
   ;
   ```

   Note that when using the above `IlvMapOutputStream` constructor, the name of the image file (see *The .img File*) is inferred from the specified map filename, by adding the `.img` extension or by replacing the `ivl` extension by `.img`. You can also create the `IlvMapOutputStream` with a different constructor, specifying a `java.io.OutputStream` to save the map contents to, and a different filename for the `.img` file. In this case, you must use the appropriate matching constructor of `IlvMapInputStream` when loading the map, see the reloading a map procedure below for more information.

2. Optionally:

   Save only the map theme. When reloading a theme file only, all `IlvMapDataSource` instances of the model are started again so that they can read data from their original files and reconstruct the map. This is useful if the original data sources (ESRI Shape, DTED® and so on) are available when reloading the `.ivl` file (when working on the same machine for example, or on the same network if data formats are stored on a networked resource). The file produced is small, typically only a few tens of KB:

   ```
   mapOutput.setWritingObjects(false);
   ```

   Conversely, if you plan on distributing the map without the original data sources, or if you want to achieve higher reload performance, you should consider saving all the data in the file:

   ```
   mapOutput.setWritingObjects(true);
   ```

3. Write the manager to the `IlvMapOutputStream` and clean up the remaining resources:

   ```
   try {
     mapOutput.write(manager);
   } catch (IOException ex) {
     // handle I/O exception here....
   }
   ```

To reload a map:

1. First, it is best is to clear the current manager, see *Clearing map data*.

2. Create an `IlvMapInputStream` and use it to read the map file. This takes care of reconnecting read data with the corresponding read map model (data sources, map layers and so on).

```
String mapFilename = "myMap.ivl";
IlvMapInputStream mapInputStream = new IlvMapInputStream (mapFilename);
try {
   mapInputStream.read(manager);
}  catch (Exception ex) {
   // handle reading exceptions here
}
```

   Note that when using the above `IlvMapInputStream` constructor, the `.img` file that goes with the `.ivl` file (passed to `IlvMapInputStream`) must be in the same directory as the `.ivl` file or the stream will not be able to find it, leading to missing images on the map.

   If you choose to specify a different name for the `.img` file when saving the map (see *Saving and reloading a map*), you must also specify the filename to the `IlvMapInputStream` using the appropriate constructor as follows:

```
String ivlFilename = "myMap.ivl";
String imgFilename = "myImgFilename.img";

//Create the input stream for the .ivl file.
FileInputStream fis = new FileInputStream(ivlFilename);

//Create the IlvMapInputstream and specify the image file name.
IlvMapInputStream mapInputStream = new IlvMapInputStream(fis, imgFilename)
;

//Read the map as described previously.
   try {
     mapInputStream.read(manager);
     } catch (Exception ex) {
// Handle reading exceptions here.
}
```

Map Export API

The map export API makes it easier for the user to export a part of a map to different output formats.

Class for exporting selected map features

The main class is the `IlvMapExportManager`. This class is responsible for exporting selected map features of a map through specified export plugins known as `IlvMapExportManager.IlvMapExporter` implementations.

Exporters

`IlvMapExportManager.IlvMapExporter` is an interface that defines the expected methods of any object capable of writing map features in sequence.

Two kind of exporters can be set on the `IlvMapExportManager`:

♦ A vectorial exporter, which handles all vectorial map features when exporting a map.

♦ A raster exporter, which handles map features containing raster data.

They are set by calling the methods `setVectorialExporter(ilog.views.maps.export.IlvMapExportManager.IlvMapExporter)` and `setRasterExporter(ilog.views.maps.export.IlvMapExportManager.IlvMapExporter)` on an `IlvMapExportManager` instance.

A region of export can also be set on the `IlvMapExportManager` so that map features outside this region are not exported.

Once the two exporters of a `IlvMapExportManager` object are properly set and configured, an array of `IlvMapLayer` instances of the map may be exported with the method `exportMapLayers(ilog.views.maps.beans.IlvMapLayer[])`.

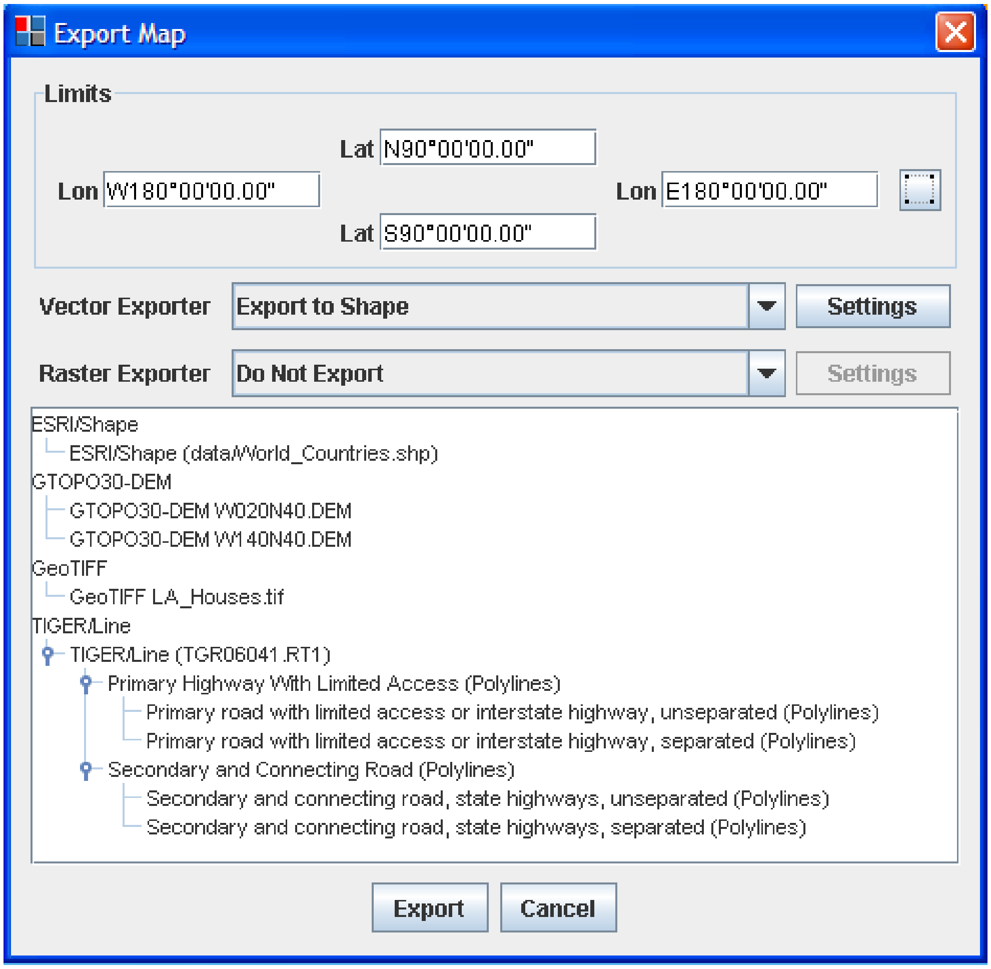
The IlvMapExportDialog class

The `IlvMapExportDialog` is a user-friendly dialog box class that lets the user choose:

♦ The exporters to use (from a list of registered exporters)

♦ A list of map layers to export

♦ A region of interest for the export



ESRI/Shape
- ESRI/Shape (data/World_Countries.shp)

GTOPO30-DEM
- GTOPO30-DEM W020N40.DEM
- GTOPO30-DEM W140N40.DEM

GeoTIFF
- GeoTIFF LA_Houses.tif

TIGER/Line
- TIGER/Line (TGR06041.RT1)
  - Primary Highway With Limited Access (Polylines)
    - Primary road with limited access or interstate highway, unseparated (Polylines)
    - Primary road with limited access or interstate highway, separated (Polylines)
  - Secondary and Connecting Road (Polylines)
    - Secondary and connecting road, state highways, unseparated (Polylines)
    - Secondary and connecting road, state highways, separated (Polylines)

Export Map Dialog Box

Vectorial and raster exporters are registered with this dialog box through the methods `registerVectorExporter(ilog.views.maps.export.IlvMapExportManager.IlvMapExporter)` and `registerRasterExporter(ilog.views.maps.export.IlvMapExportManager.IlvMapExporter)`, and similarly are removed by calling `unregisterVectorExporter(ilog.views.maps.export.IlvMapExportManager.IlvMapExporter)` and `unregisterRasterExporter(ilog.views.maps.export.IlvMapExportManager.IlvMapExporter)` methods.

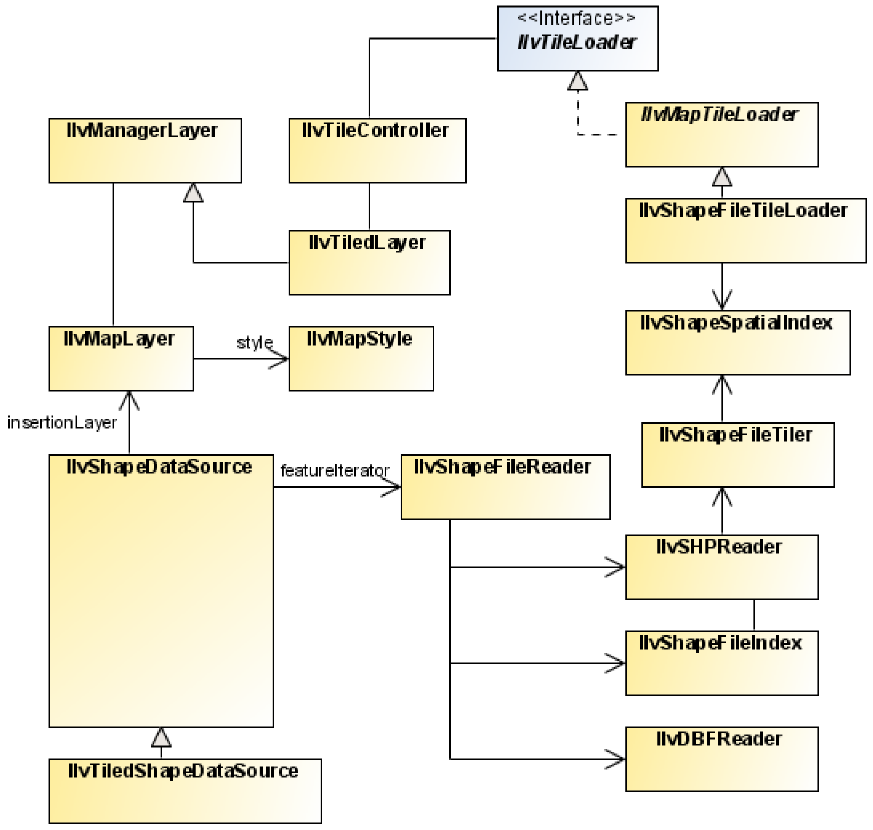
The shapefile reader and writer

Describes the shapefile reader and writer classes and data sources.

In this section

The shapefile reader and writer classes

The class diagram for the shapefile reader and writer is shown in *Shapefile Reader and Writer UML Diagram*.



Shapefile Reader and Writer UML Diagram

These classes are based on the Shapefile Specifications listed in the ESRI (Environmental Systems Research Institute) document: `ESRI Shapefile Technical Specifications - An ESRI White Paper - July 1998`.

The Shapefile format is the exchange format for vector maps of the ESRI. This format supports polygons, arcs, lines, and points. Each Shapefile contains one single theme, meaning that all the objects in the file are of the same type (either line, point, polygon, or another type of object). In the Shapefile format, a theme is essentially described with four different files:

♦ A shapefile (`.shp`) contains the geometry of the objects.

♦ A Dbase file (`.dbf`) contains the attributes of the objects.

- An index file (`.shx)` contains the index and sizes of the objects of the `.shp` file.
- A spatial index file (`.idx`) contains tiling information. This file is JViews Maps package specific, and is used to performload-on-demand on Shapefiles.

This format does not contain information concerning the coordinate system used to reference the position of the graphic objects. Objects in Shapefiles are often positioned within a geographic coordinate system (`IlvGeographicCoordinateSystem`), but this is far from being the rule.

The Shapefile format comprises the following classes:

- *The shape data source*
- *Classes for reading the shape format*
- *Classes for writing the shape format*
- *Shapefile load-on-demand*

The complete source code for an ESRI shapefile demonstration can be found at ***<installdir>*** **`/jviews-maps86/samples/shape/index.html`**

The shape data source

The `IlvShapeDataSource` class is a specialized data source that reads ESRI shapefiles.

Reading Shapefiles

The easiest way of reading shapefiles is to use the dedicated data source:

```
// Create the data source
IlvShapeDataSource source = new IlvShapeDataSource(fileName);
// affect the manager
source.setManager(manager);
// start the data source.
try {
  source.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

In this example, the data source is simply invoked and started. You have to use the `setManager(ilog.views.IlvManager)` method to specify the manager into which the graphic objects are inserted. Graphic objects are instances of `IlvMapPolyline`, `IlvMapGraphicPath` or `IlvMapPoint`, depending on the feature contained in the shapefile. Additionally, the data source creates an `IlvMapLayer`. You can specify a `IlvMapStyle` to change the rendering of the graphic objects in this layer.

Filtering Shapefiles

If the shape file you read has an associated dbf file, you can filter a subset of the file content through the `setfilter` method. For example, the following code will only load the records which have a "NAME" property with a value "usa".

```
source.setFilter(new IlvSplitEqualsFilter("NAME", "usa", true);
```

Modifying graphic object rendering

The following example shows how to modify the rendering of the graphic objects produced by this data source by changing the style of the `IlvMapLayer` of the data source.

```
// create the data source
IlvShapeDataSource source = new IlvShapeDataSource(fileName);
source.setManager(manager);
// Assuming that the geometry of the shape file are areas.
IlvGraphicPathStyle style = new IlvGraphicPathStyle();
source.getInsertionLayer().setStyle(style);
style.setPaint(Color.blue);
try {
  source.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

If you are not sure of the type of objects contained in the shape file, you can use the `setAttribute` method on the layer style, for example:

```
    IlvMapLayer layer = source.getInsertionLayer();
    layer.getStyle().setAttribute(IlvPolylineStyle.FOREGROUND,Color.black);
      layer.getStyle().setAttribute(IlvPolylineStyle.BACKGROUND,new
    Color(1,1,0.8f));
```

Classes for reading the shape format

The ilog.views.maps.format.shapefile package includes the following classes:

- IlvShapeFileReader

  This class implements the IlvMapFeatureIterator interface and allows you to read .shp, .dbf and .shx files. Since Shapefiles provide no information on the projection system used, this reader is not georeferenced. This reader uses the two specialized readers described below. Its getNextFeature()getNext method merges the information generated by these specialized readers into a single map feature.

- IlvSHPReader

  This class implements the IlvMapFeatureIterator interface. This reader only reads .shp files.

- IlvDBFReader: reads .dbf files.
- IlvShapeFileIndex: reads .shx files.
- IlvShapeSpatialIndex: reads maps spatial index .idx files.

The IlvSHPReader class

The geometries stored in Shapefiles are not necessarily 2-D objects. Each point that makes up a shape object can be associated with measurements, or with measurements and an elevation.

Measurements are stored in an attribute of type IlvAttributeArray, which itself is stored in the map feature attribute of index 0.

The following are the shape types that are associated with measurements:

- POINTZ
- POLYLINEZ
- POLYGONZ
- MULTIPOINTZ
- POINTM
- POLYLINEM
- POLYGONM
- MULTIPOINTM

Elevations are stored in an attribute of type IlvAttributeArray, which itself is stored in the map feature attribute of index 1.

The following are the shape types that are associated with measurements and elevations:

- POINTZ

♦ POLYLINEZ

♦ POLYGONZ

♦ MULTIPOINTZ

Since the JViews Maps package does not have a predefined geometry to represent shape objects of type MULTIPATCH, which are essentially used for 3-D rendering, these are ignored. It is possible, however, to modify this behavior by subtyping the class IlvShapeSHPReader. Since shape objects are read in protected methods, modifying the reader to include new geometries requires minimal effort.

The IlvDBFReader class

This reader is used exclusively for reading a file of the .dbf format. It can be used to iterate over a file as follows:

```
try {
      IlvDBFReader reader = new IlvDBFReader("myFile.dbf");
      IlvFeatureAttributeProperty attributes = reader.getNextRecord();
      while (attributes != null) {
        // Process attributes.
        ...
       attributes = reader.getNextRecord();
      }
   } catch (Exception e) {
      e.printStackTrace();
   }
```

If the reader has been created from a file and not from a URL, you can access map feature attributes directly by specifying their record number:

```
reader.readRecord(index);
```

The IlvShapeFileReader class

This reader reads the .shp file storing geometries and the .dbf file storing attributes simultaneously, and merges the information into a single IlvMapFeature object.

It can be instantiated in one of three ways:

♦ By specifying the name of the .dbf and .shp files.

♦ By specifying the URL of these two files.

♦ By specifying an IlvDBFReader and IlvSHPReader object directly.

  This is useful, for example, when using a derived IlvSHPReader object.

Classes for writing the shape format

The `ilog.views.maps.format.shapefile` package contains the following classes for writing Shapefiles.

The class `IlvSHPWriter` is used to generate the geometry and index parts of a Shapefile (`.shp` and `.shx` files), while the `IlvDBFWriter` is used to write the attribute file (`.dbf` extension).

Like with feature iterator, writing map features to a Shapefile consists of writing repeatedly map features using the methods `writeFeature(ilog.views.maps.IlvMapFeature)` and `writeAttributes(ilog.views.maps.IlvFeatureAttributeProperty)`, then call the `close()` method of writers to flush the data and write the headers.

The IlvSHPWriter class

The `IlvSHPWriter` class manages the writing of geometries to a Shapefile, with the creation of the Shapefile index that allows direct access to Shapefile records.

The following example shows how to use this class to write the contents of a feature iterator to a file `foo.shp`, creating the index file at the same time.

```
try {
   IlvSHPWriter shpwriter = new IlvSHPWriter("foo.shp",
                                             "foo.shx");
  // Loop on features.
  IlvMapFeature feature = iterator.getNextFeature();
  while (feature != null) {
    shpwriter.writeFeature(feature);
    feature = iterator.getNextFeature();
  }
  shpwriter.close();
} catch (IOException e) {
  // Error processing.
  e.printStackTrace();
}
```

Note: The Shapefile format defines a header that can be completed only once all data is written. For this reason, it is mandatory to call the `close()` method of the shape writer once all data is written, so that the header is updated.

The IlvDBFWriter and IlvDBFAttributeInfo classes

The `IlvDBFWriter` and `IlvDBFAttributeInfo` classes manage the writing of DBase III+ files (`.dbf` files). These files contain records corresponding to attributes of geometries contained within a Shapefile.

As `.dbf` files need records with fixed-size fields, it is important to choose a field size that is large enough to contain all data of a field, as well as a size small enough not to waste data.

The goal of the class `IlvDBFAttributeInfo` is to complement `IlvAttributeInfoProperty` for the definition of record fields.

The following example shows how to write the contents of an iterator to a set of `.shp`, `.shx` and `.dbf` files:

```
try {
  // Create the SHP writer.
  IlvSHPWriter shpwriter = new IlvSHPWriter("foo.shp", "foo.shx");

  // Read the first feature.
  IlvMapFeature feature = iterator.getNextFeature();

  // Create the DBF Writer.
  IlvDBFAttributeInfo info =
      new IlvDBFAttributeInfo(feature.getAttributeInfo());

  IlvDBFWriter dbfwriter = new IlvDBFWriter(info,
                                            "foo.dbf");

  // Loop on features.
  while (feature != null) {
    shpwriter.writeFeature(feature);
    dbfwriter.writeAttributes(feature.getAttributes());
    feature = iterator.getNextFeature();
  }
  shpwriter.close();
  dbfwriter.close();
} catch (IOException e) {
  // Error processing.
  e.printStackTrace();
}
```

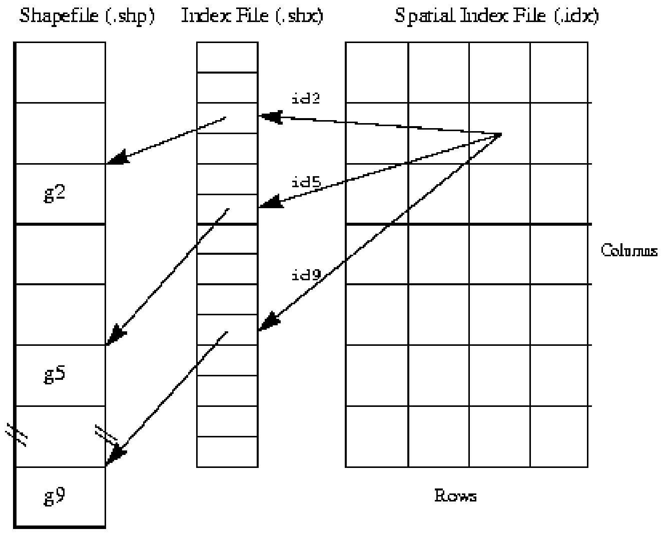
Once again, the writers must be closed to write the headers correctly.

Shapefile load-on-demand

The Maps package provides classes to perform load-on-demand on Shapefiles. This is achieved by the use of specific spatial index files. These files, usually having an `.idx` extension, store relations between tiles and object identifiers that belong to these tiles. A class and a tool example are provided to generate these spatial index files. A generic tile loader is also provided to minimize the amount of code needed to implement the load-on-demand mechanism using Shapefiles.

The load-on-demand mechanism involves two classes in addition to the shape reader and the `dbf` reader: the `IlvShapeFileIndex` class and the `IlvShapeSpatialIndex` class. A utility class is also provided to generate the spatial index for a given Shapefile: the `IlvShapeFileTiler` class.

The mechanism used to store and retrieve objects by tiles is illustrated in the following diagram:



The Spatial Index file holds objects identifiers for each tile. Objects identifiers are their ordinal place in the IndexFile. Geometries are retrieved in the Shapefile using the IndexFile. In the following example, the tile [2, 1] (tiles indices begin at 0) contains identifiers 2, 5 and 9 referring to the geometries g2 g5 and g9.

The classes used to perform load-on-demand on the Shapefiles are the following:

- *The IlvShapeFileIndex Class*
- *The IlvShapeSpatialIndex Class*
- *The IlvShapeFileTiler Class*
- *The IlvShapeFileTileLoader Class*

The IlvShapeFileIndex Class

This class allows you to directly access geometries in a Shapefile. The spatial index and the Shapefile must correspond to the same theme:

```
// Open the index file.
IlvShapeFileIndex index = new IlvShapeFileIndex("example.shx");
// Open the corresponding Shapefile.
IlvSHPReader shape = new IlvSHPReader("example.shp");
// Retrieve the feature for each index.
int count = index.getRecordCount();
for(int i = 0; i < count; i++)
    IlvMapFeature f = shape.getFeatureAt(i);
```

The IlvShapeSpatialIndex Class

This class stores tile information: tile size and count, and identifiers of the objects belonging to each tile. To retrieve objects from a tile specified by its row and column, use the `getIdArray(int, int)` method:

```
// Open the spatial index file.
IlvShapeSpatialIndex spatialIndex =
    new IlvShapeSpatialIndex("example.shx");
// Loop on all columns and rows.
for(int c = 0; c < spatialindex.getColumnCount(); c++) {
    for(int r = 0; r < spatialindex.getRowCount(); r++) {
        // Retrieve the IDs of objects belonging to the tile at row ‘r’ and
        // column ‘c’.
        int[] ids = spatialindex.getIdArray(c, r);
        // Loop on these IDs and get the corresponding map feature.
        for(int i =0; i < ids.length; i++) {
            IlvMapFeature f = shape.getFeatureAt(i);
        }
    }
}
```

The IlvShapeFileTiler Class

This class is used to generate tiling information from a given Shapefile. To use this class you have to provide the Shapefile to tile, the SpatialIndexFile to write to, and either the tile size or the number of rows and columns.

```
IlvShapeFileTiler.CreateShapeSpatialIndex("example.shp",
                                          "example.idx",
                                                      5., 10.);
```

The above code extract produces a `SpatialIndexFile` named `example.idx` with a tile size of width 5 and height 10.

```
IlvShapeFileTiler.CreateShapeSpatialIndex("example.shp",
                                          "example.idx",
                                          20, 30);
```

The above code extract produces a `SpatialIndexFile` of 600 tiles, 20 columns and 30 rows.

The IlvTiledShapeDataSource

The `IlvTiledShapeDataSource` is the tiled version of the `IlvShapeDataSource`. This means that it takes advantage of the tiling and load-on-demand mechanism. To read a tiled shapefile with such a data source, you can use the following code:

```
IlvTiledShapeDataSource source = new IlvTiledShapeDataSource(fileName);
source.setManager(getManager());
try {
  source.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

The IlvShapeFileTileLoader Class

This class implements load-on-demand for tiled Shapefiles. When associated with an `IlvTiledLayer`, this class automatically handles tile loading if the Shapefile file name, the IndexFile file name, and the SpatialIndexFile file name are provided. An optional Dbase file name can also be provided to load object attributes.

```
IlvShapeFileTileLoader tileLoader =
    new IlvShapeFileTileLoader("example.shp",
                              "example.dbf", // Or null if attributes loading

                                             // is not wanted.
                               "example.shx",
                               "example.idx");
IlvTiledLayer tiledLayer = new IlvTiledLayer(new IlvRect(), null,
   IlvTileController.FREE);
tiledLayer.setTileLoader(tileLoader);
```

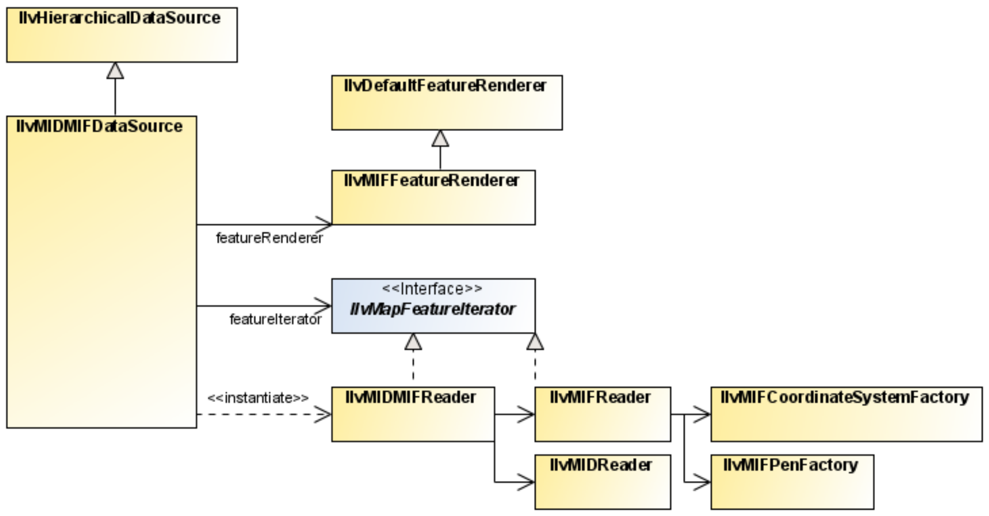
The MID/MIF reader and writer

Describes the MID/MIF reading and rendering classes and data sources.

In this section

The MID/MIF classes

The class diagram for the mid/mif reader is shown in *MID/MIF Reader UML Diagram*.



MID/MIF Reader UML Diagram

This section describes the `ilog.views.maps.format.midmif` package, which allows you to read files provided in the MapInfo Interchange File (MIF) format.

A MapInfo Interchange Format consists of a MIF file containing the geometries and rendering attributes, and a MID file containing the attributes attached to each geometry.

The MIF file is an ASCII text file that describes the geometry data associated with rendering attributes such as pen and brush styles, font, and color that should be used to transform geometries into graphic objects. The reader classes provided with the JViews Maps package allow you to use either the information contained in the MIF file or your own rendering attributes.

The MID file contains attributes, which are nongraphical additional information associated with the geometries contained in the MIF file.

The MID/MIF Reader contains the following classes:

♦ *The MIDMIF data source*

♦ *Classes for reading the MapInfo Interchange File format*

♦ *Classes for rendering the MapInfo Interchange File format*

♦ *Coordinate System Support*

The MIDMIF data source

The `IlvMIDMIFDataSource` class is a specialized data source that reads MIDMIF files. It wraps all MIDMIF file rendering processes.

```
IlvMIDMIFDataSource source = new IlvMIDMIFDataSource(filename);
source.setManager(manager);
source.start();
```

The `IlvMIFDMIFDataSource` produces a `IlvMapLayer` for objects having the same geometries and similar graphic attributes. If, however, a MIDMIF file contains lines whose color attributes are different, for example, red and blue, two layers are created, one for the blue lines and one for the red lines.

Classes for reading the MapInfo Interchange File format

The JViews Maps package provides all the necessary classes to read data provided in the MapInfo Interchange File (MIF) format. In addition to these classes, the JViews Maps package also provides the `IlvGraphic` classes needed to render specific MIF objects.

The classes used to read MapInfo Interchange files are the following:

♦ *The IlvMIDMIFReader class*

♦ *The IlvMIFReader class*

♦ *The IlvMIDReader class*

The IlvMIDMIFReader class

This class reads both MIF file and MID file by providing a MIF file name and a MID file name. If the MID file name is set to `null`, no attribute is loaded. If the MID file name corresponds to a valid MID file, the attributes are read from the file and attached to the `IlvMapFeature` returned by the reader.

```
IlvMIDMIFReader reader = new IlvMIDMIFReader("example.mif", "example.mid");
IlvMapFeature feature = reader.getNextFeature();
while(feature != null)
     feature = reader.getNextFeature();
```

The IlvMIFReader class

This is the class that reads a single MIF file, providing a MIF file name or a reader to the constructor. You can then iterate to obtain the `IlvMapFeatures` of the file, but you can also obtain information about the MIF file you are reading, such as:

♦ The `IlvAttributeInfoProperty`, which defines the name and the classes of the attributes that can be found in the associated MID file.

♦ The default feature renderer suitable to render the geometries contained in this file.

♦ The character encoding used in this file.

♦ The coordinate system in which the geometries of the file are expressed.

Note that the coordinate system returned by the reader can be `null` even if the MapInfo specifications consider the geometries to be in the Geographic coordinate system if none is specified in the MIF file.

```
IlvMIFReader reader =
    new IlvMIFReader("example.mif");
IlvAttributeInfoProperty info = reader.getAttributeInfo();
if(info != null) {
    System.out.println("Attributes info : ");
    for(int i = 0; i < info.getAttributesCount(); i++) {
        System.out.println("Name " + info.getAttributeName(i));
        System.out.println("Class " + info.getAttributeClass(i));
    }
}
IlvCoordinateSystem cs = reader.getCoordinateSystem();
if(cs != null)
    System.out.println("The coordinate system is " + cs.getName());
else
    System.out.println("Assuming a geographic coordinate system");
String encoding = reader.getCharset();
System.out.println("Char set is " + encoding);
```

The IlvMIDReader class

This class reads a MID file. You must have previously opened the corresponding MIF file, in order to build the second argument of the MID reader (`IlvAttributeInfoProperty`) that is provided in the MIF file.

```
IlvMIFReader reader =
    new IlvMIFReader("example.mif");
IlvAttributeInfoProperty info = reader.getAttributeInfo();
IlvMIDReader mid = new IlvMIDReader("example.mid", info);
IlvFeatureAttributeProperty prop = mid.getNextRecord();
while(prop != null)
    prop = mid.getNextRecord();
```

Classes for rendering the MapInfo Interchange File format

Since the MIF file can contain rendering information, the JViews Maps package provides classes to use this rendering information directly. This includes a ready-to-use `IlvFeatureRenderer` suitable to render geometries found in the MIF file and a factory to create `IlvMapLineRenderingStyle`.

In addition to these rendering classes, the JViews Maps package also provides a set of specific `IlvGraphic` objects to display the MID/MIF geometries. These `IlvGraphic` objects are needed by the MID/MIF specifications but can also be used outside a MID/MIF context.

The IlvMIFFeatureRenderer Class

This class is a subclass of the `IlvDefaultFeatureRenderer` that dispatches on the appropriate renderer with the rendering styles found in the MIF file. This is the `IlvFeatureRenderer` returned by the `getDefaultFeatureRenderer()` method of the `IlvMIFReader` class. The rendering styles found in the MIF file are attached to the `IlvMapFeature` as properties by the reader and interpreted in the renderer.

The IlvMIFPenFactory Class

This class is a factory to create an `IlvMapLineRenderingStyle` by specifying a MID/MIF pattern identifier.

The IlvMIFCoordinateSystemFactory Class

This class is a factory to translate a MID/MIF coordinate system into a Maps coordinate system.

Specialized Graphics

The `IlvGraphic` objects needed to render MID/MIF geometries include the following classes:

- ♦ `IlvDecoratedPath`: A general path supporting decorations. A decoration is a drawing that follows the path of the graphic. These decorations can be clipped against the clip region of the `java.awt.Graphics` to provide high performance drawing.
- ♦ `IlvMapLabel`: A label able to display multiline text. This label supports interline spacing.
- ♦ `IlvFontMarker`A graphic object able to display a character of a given font as the marker point. This object is usually used with a symbol font that provides cartographic symbology.

Coordinate System Support

Along with objects and attribute information, MIF files also contain information on the coordinate system used to store the graphic objects of a defined file.

The possible coordinate systems in MID/MIF are the following:

- Earth: a coordinate system where coordinates are expressed within a projected coordinate system. See *List of Supported MID/MIF Projections* for the list of supported projections.
- NonEarth: a coordinate system where coordinates are expressed in a specified unit. When such a file is encountered, the coordinate system is set to `null`, and the `getUnit()` method returns the unit defined in the file.
- Layout: a coordinate system corresponding to coordinates on a sheet of paper. When such a file is encountered, the coordinate system is set to `null`, and the `getUnit()` method returns the unit defined in the file.
- Table: a coordinate system corresponding to an open table in MapInfo. This coordinate system is not supported in the JViews Maps package.
- Window: a coordinate system corresponding to an open window in MapInfo. This coordinate system is not supported in the JViews Maps package.

The JViews Maps package fully supports projected coordinates defined in a MID/MIF file. The supported projections are listed in *List of Supported MID/MIF Projections*:

List of Supported MID/MIF Projections

MID/MIF Projection Number	Name	Corresponding Projection
9	Albers Equal-Area Conic	IlvAlbersEqualAreaProjection
5	Azimuthal Equidistant	IlvAzimuthalEquidistantProjection
2	Cylindrical Equal Area	IlvCylindricalEqualAreaProjection
14	Eckert IV	IlvEckert4Projection
15	Eckert VI	IlvEckert6Projection
17	Gall	IlvMercatorProjection
7	Hotine Oblique Mercator	IlvObliqueMercatorProjection
4	Lambert Azimuthal Equal Area	IlvLambertAzimuthalEqualAreaProjection
3	Lambert Conformal Conic	IlvLambertConformalConicProjection
19	Lambert Conformal Conic (modified for Belgium 1972)	IlvLambertConformalConicProjection
1	Longitude/Latitude	Not a projection. An `IlvGeographicCoordinateSystem` is used.
10	Mercator	IlvMercatorProjection
11	Miller Cylindrical	IlvMillerCylindrical Projection
13	Mollweide	IlvMollweideProjection
27	Polyconic	IlvPolyconicProjection
12	Robinson	IlvRobinsonProjection
16	Sinusoidal	IlvSinusoidalProjection
20	Stereographic	IlvStereographicProjection
8	Transverse Mercator	IlvTransverseMercatorProjection

JViews Maps supports all ellipsoids and nearly all datums from MapInfo (see the MID/MIF file format specification for the full list of ellipsoids and datums).

Note: The MIF format specification allows user datum definition, with either 3 or 7 parameters. As the JViews Maps package does not support 7-parameter datum yet, definitions of 7- parameter datum is decoded as 3-parameter datum, skipping the rotations and scale part of datum definitions.

The DTED file reader

You can use the DTED® file reader to read Digital Terrain Elevation Data (DTED format) files. The DTED read format is a map format for representing terrain elevations published by the U.S. National Imagery and Mapping Agency (NIMA). The DTED files contain digital terrain models as rasters. A raster is a georeferenced grid containing a value in each of its cells. In the case of DTED, and a digital terrain model in general, the value indicates the average elevation in the cell. However, this value can indicate any other attribute: surface temperature, surface pressure, nitrogen rating of the soil, and so on.

The DTED reader provided in this package is based on the specification document MIL-PRF-89020A of 19 April 1996. DTED files are available with various precision levels, called DTED0, DTED1, and DTED2. A DTED file contains a digital terrain model raster that covers a zone of one degree by one degree. The cell size of the raster depends on the DTED level:

♦ DTED0 provides raw data (approximately 30 to 40 KB for a file).

♦ DTED1 provides data that is more detailed.

♦ DTED2 is the most precise level. It has surface cells that are nine times smaller than those of DTED1. At this degree of precision, a DTED file is enormous (several megabytes).

The complete source code for a DTED demonstration can be found at

<installdir> **`/jviews-maps86/samples/dted/index.html`**.

The DTED raster reader

The following code shows how to use an `IlvRasterDTEDReader` to load a single DTED file using the `IlvMapDataSource`:

```
IlvRasterDTEDReader r = new IlvRasterDTEDReader();
try {
  r.addMap(filename);
} catch (IOException e1) {
  e1.printStackTrace();
}
IlvMapDataSource source =
IlvRasterDataSourceFactory.buildImageDataSource(manager, r, null);
try {
  source.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

The following code shows how to read a set of DTED files using an `IlvTiledRasterDataSource`. This produces an `IlvMapLayer` containing an `IlvTiledLayer` holding the tiles.

```
IlvRasterDTEDReader r = new IlvRasterDTEDReader();
try {
  r.addMap(filename);
} catch (IOException e) {
  e.printStackTrace();
}
IlvMapDataSource source =
IlvRasterDataSourceFactory.buildTiledImageDataSource(manager, r, true, true,
null);
try {
  source.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

Classes for reading the DTED format

The main class for reading DTED formats is `IlvDTEDReader`. This class implements the `IlvMapFeatureIterator` interface and returns only one `IlvMapFeature` object, which is the raster corresponding to the digital terrain model (DTM) stored in the file. The geometry of this map feature is of type `IlvMapRaster`. The map feature has no attribute. The projection of the reader is the source projection of the DTED data, that is, the geographic projection.

The `IlvDTEDLayer` class defines load-on-demand for the DTED format. Load-on-demand is implemented on a DTED-level basis from the corresponding file name or URL. In other words, the size of a tile in a JViews Maps tiled layer corresponds to the size of a DTED tile. This specific implementation of load-on-demand works exclusively with maps drawn with the geographic projection. Specifying a URL allows you to access one of the NIMA CDs directly from their web site. For an example, see the DTED demonstration at the following location:

<installdir> **`/jviews-maps86/samples/dted/index.html`**.

The image file reader

The formats handled by the generic image file reader, which are GIF, PNG and JPEG, are supported by the Java™ Platform, Standard Edition. An image coded in one of these formats does not contain any geo-referencing information, so this information has to be known before loading this kind of image. This is also true for TIFF images that do not contain GeoTIFF tags. Below you can find a description of the `IlvRasterBasicImageReader`, the `IlvImageReader` class and `IlvImageTileLoader` class.

The IlvRasterBasicImageReader class

The `IlvRasterBasicImageReader` class is a GIF, JPG, PNG or TIFF reader that creates reprojectable, stylable, and pixel-on-demand images.

Creating an image reader

You need first to create an image reader, and add the image file to be read:

```
IlvRasterBasicImageReader imageReader = new IlvRasterBasicImageReader();
imageReader.addMap(gifFile);
```

As these images have no longitude or latitude information, you need to geo-reference this image. The easiest way to do this is to set the image bounds in longitude and latitude. This obviously works only for images that have a spatial coverage along lines of longitude and latitude. For advanced management of more complex images, the Map Builder provides an image transformation model that manages reprojection and image interpolation (`ImageControlModel` class).

In the example below, you set the image to cover the entire earth:

```
imageReader.setImageBounds(0,-Math.PI,Math.PI/2,Math.PI,-Math.PI/2);
```

Creating a data source

Once you have created the reader, you need to create a data source, which should be integrated into the manager properties:

```
IlvMapDataSource imageDataSource =
IlvRasterDataSourceFactory.buildTiledImageDataSource(manager,imageReader,true,t
rue,null);
IlvMapDataSourceModel dataSourceModel =
IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(imageDataSource);
```

Reading the data

You can then start reading your data:

```
dataSourceModel.start();
```

Starting the data source creates the necessary tiled layers, tile managers and `IlvRasterIcon` instances to manage the pixel-on-demand feature and the progressive display of the new geo-referenced image.

You can also create a subsampled preview of an image, using code such as:

```
int iconWidth=imageReader.getRasterProperties(0).getNumColumns();
int iconHeight=imageReader.getRasterProperties(0).getNumLines();
Image image =
Toolkit.getDefaultToolkit().createImage(iReader.getTileLoader(0).getScaledImage
Producer(subsampling, new IlvRect(0, 0, iconWidth, iconHeight)));
```

The IlvImageReader class

If you do not need image the reprojection, styling or pixel-on-demand features, but simply want to insert and geo-reference a single GIF, JPG or PNG image, you can use the `IlvImageReader` class.

Note: If you use this class, the images created are not compatible with the data source and map layer management.

This class implements the `IlvMapFeatureIterator` interface. It returns only one `IlvMapFeature` object, which is the image stored in the file.

The geometry of this map feature is of type `IlvMapImage`. The map feature has no attributes. To use this reader you have to provide a file name and the coordinates of this image.

```
// The image is known to be at 77 degrees 30 seconds east
// and 10 degrees north for the upper left corner.
// Lower right corner is at 82 degrees 30 seconds east
// and 5 degrees north.
IlvCoordinate ul = new IlvCoordinate(77.5, 10);
IlvCoordinate lr = new IlvCoordinate(82.5, 5);
IlvImageReader reader = new IlvImageReader("image.jpg", ul, lr);
IlvMapFeature feature = reader.getNextFeature();
IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
// Image is known to be in the geographic coordinate system.
IlvCoordinateTransformation tr
       = new IlvCoordinateTransformation(IlvGeographicCoordinateSystem.WGS84,

                                         IlvGeographicCoordinateSystem.WGS84,

                                         new IlvMapAffineTransform());
IlvGraphic g = renderer.makeGraphic(feature, tr);
manager.addObject(g, false);
```

The IlvImageTileLoader class

If you do not need image reprojection, styling or pixel-on-demand features, but have access to a set of images, you can use the `IlvImageTileLoader` class.

Note: If you use this class, the images created are not compatible with the data source and map layer management. This information is provided for compatibility with older versions of JViews.

This class is used to read a set of images that are part of a larger image. This tile loader allows an application to load only the images that are visible at a given time, each image corresponding to a tile. Each file must be named so that it is possible to construct its file name knowing the row index and column index of the corresponding tile. To use this tile loader, you must provide the information needed to reconstruct the file name for a given tile: a pattern that matches the file naming scheme and two formatting strings.

```
IlvImageTileLoader loader = new IlvImageTileLoader(String pattern,
                                                   String rowFormatString,
                                                   String colFormatString);
```

The `pattern` argument must contain one ‘%r’ and one ‘%c’ conversion specifier. The %r conversion specifier is used to convert the row index of the tile, and the %c conversion specifier is used to convert the column index of the tile. These conversion parameters are replaced accordingly with the `rowFormatString` and the `colFormatString` parameters. This format string is used to construct two `java.text.DecimalFormat` strings.

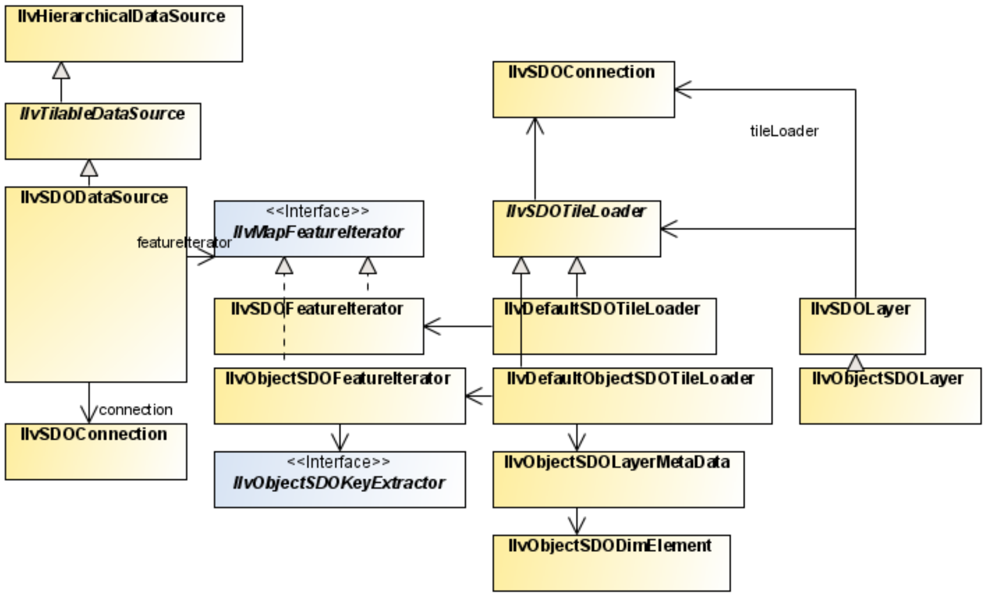
The Oracle spatial reader and writer

Describes the Oracle® reader and writer classes, using data sources, how to use tiling, multithreading and exporting to an SDO database.

In this section

The Oracle spatial reader and writer classes

The class diagram for the Oracle® spatial reader and writer is shown in *Oracle Spatial Reader and Writer UML Diagram*.



Oracle Spatial Reader and Writer UML Diagram

Oracle SDO, or Oracle Spatial, is the spatial extension of Oracle in its version 7.3. This extension has been renamed to Spatial Cartridge in the version 8.0 and has been renamed again to Oracle Spatial in the version 8i.

The classes are based on the relational implementation of Oracle Spatial available since Oracle 7.3. It contains classes that facilitate the use of map data (stored in an Oracle database) in an IBM® ILOG® JViews application. Oracle Spatial allows you to store georeferenced objects in an Oracle database and to perform spatial queries, such as getting the list of objects that intersect a specific polygon.

Oracle has written two implementations of Oracle Spatial:

- ♦ An implementation based on relational tables, available since Oracle 7.3.
- ♦ An implementation based on the Object Relational model, available since Oracle 8i, which also contains the relational implementation of Oracle Spatial.

The Oracle reader of JViews Maps supports reading and writing data in the relational and the object relational model of Oracle Spatial through two different packages:

- ♦ The package `ilog.views.maps.format.oracle` contains classes to read and write Oracle Spatial Relational model.
- ♦ The package `ilog.views.maps.format.oracle.objectmodel` contains classes to read and write Oracle Spatial Object Relational model.

The Oracle Spatial Reader and Writer comprises the following classes:

- *Using the Oracle SDO data source*
- *Relational model classes*
- *Object relational model classes*
- *Oracle SDO export*

The complete source code for an Oracle Reader demonstration can be found at ***<installdir>*** **`/jviews-maps86/samples/oracle/index.html`**.

Using the Oracle SDO data source

The `IlvSDODataSource` class is an `IlvMapDataSource` for reading georeferenced objects from an Oracle® SDO (formerly Oracle Spatial) database. It relies on the Oracle SDO API that was introduced in former versions of the JViews Maps product, but wraps up all operations (connecting to database, reading features, rendering them to graphic objects) in a more convenient way.

To read a map stored in an Oracle SDO database:

1. Create a connection to the Oracle database:

   ```
   String url = "jdbc:oracle:thin:@hostMachine:1529:mySID";
   String userName="login";
   String password="pass";
   IlvSDOConnection connection = new IlvSDOConnection(url,userName,password)
   ;
   connection.createConnection();
   ```

2. Create a data source with this connection as a parameter:

   ```
   // Assume we want to read from an object model Oracle database (not
   relational)
   boolean isObjectModel = true;

   // we want to fetch layer "MY_LAYER_GEOMETRY"
   IlvSDODataSource SDODataSource  =
   new IlvSDODataSource(connection, isObjectModel , "MY_LAYER_GEOMETRY");
   ```

   To get a list of layers, see *Getting a list of layers*.

3. Set parameters on this data source to use load-on-demand:

   ```
   boolean useTiling = true;
   int rowCount = 5;
   int columnCount = 5;
   SDODataSource.setTilingParameters(useTiling, rowCount, columnCount);
   ```

4. Connect this data source to the manager of the view:

   ```
   SDODataSource.setManager(getView().getManager());
   ```

5. Insert the data source into the data source tree. You first need to retrieve the data source model from the property of the manager:

   ```
   IlvMapDataSourceModel dataSourceModel =
   IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
   dataSourceModel.insert(SDODataSource);
   ```

6. Finally, start the Oracle SDO data source:

   ```
   SDODataSource.start();
   ```

Using tiling and multithreading

To benefit from IBM® ILOG® JViews Maps advanced image tiling capabilities and multithreading, use `IlvTiledRasterDataSource` in the place of `IlvSDODataSource`

Note: This assumes that you want to load an SDO layer containing raster objects, SDO_GEORASTER.

To use a tiled data source:

1. Instantiate a `IlvRasterSDOReader`IlvRasterSDOReader object.

   This class extends `IlvRasterAbstractReader`IlvRasterAbstractReader and holds a list of raster elements. It fetches the following elements from the Oracle® DB

```
IlvSDOConnection SDOConnection; // the connection to your Oracle DB
String layerName;  // the name of the oracle layer containing SDO_GEORASTER
... // Initialize your connection.
IlvRasterSDOReader sdoReader = new
IlvRasterSDOReader(SDOConnection,layerName);
```

2. Create an `IlvTiledRasterDataSource`IlvTiledRasterDataSource backed by the reader.

```
IlvTiledRasterDataSource tiledSource =
IlvRasterDataSourceFactory.buildTiledImageDataSource(
view.getManager(), sdoReader, true,true, null);
```

3. Configure the raster datasource, and call the `start()` method to start data production.

```
tiledSource.setManager(view.getManager());
tiledSource.start();
```

Getting a list of layers

To get a list of layers available in the Oracle® database, you can do the following:

♦ For a relational model:

♦ For an object model:

```
String[] layersList =
IlvSDOUtil.GetAllLayers(connection.getConnection(),connection.getUser());
layersList = IlvObjectSDOUtil.GetAllLayers(connection.getConnection(),
connection.getUser(), true);
```

Relational model classes

Classes for reading data from an Oracle spatial relational model database

The reader classes for Oracle® SDO relational model (package `ilog.views.maps.format.oracle`) are:

♦ `IlvSDOFeatureIterator` for converting Oracle Spatial layer data into `IlvMapFeature` objects.

♦ `IlvSDOLayer` for implementing load-on-demand for Oracle Spatial data.

♦ `IlvSDOTileLoader` abstract class for defining Oracle queries for the `IlvSDOLayer`. An optimized subclass, `IlvDefaultSDOTileLoader` is used by `IlvSDOLayer`.

The IlvSDOFeatureIterator class

The `IlvSDOFeatureIterator` class reads data from the result of an SQL query to a relational Oracle Spatial layer and converts them into `IlvMapFeature` objects. JViews Maps applications can handle Oracle Spatial data using this class in a transparent manner.

The following example of Java™ code performs a query, loading data from an Oracle Spatial layer named `ROADS_SDOGEOM`. It includes classes from the `java.sql` package:

```
String query = "SELECT * FROM ROADS_SDOGEOM ORDER BY 1, 2, 4 ";
Statement statement = getConnection().createStatement();
ResultSet resultSet = statement.executeQuery(query);
IlvSDOFeatureIterator iterator = new IlvSDOFeatureIterator(resultSet);
```

The `connection` variable is a `java.sql.Connection` object.

The query orders the result using the following three criteria, which must be given in the order indicated:

1. GID (Geometric ID)
2. ESEQ (Element Sequence)
3. SEQ (Row Sequence)

Note: This ordering is necessary for the `IlvSDOFeatureIterator` to work correctly.

The `ResultSet` of any query to an Oracle Spatial layer can be used to initialize an `IlvSDOFeatureIterator`, but all the SDO columns must be in the `resultSet` (columns defining the GID, ESEQ, ETYPE, SEQ, and the coordinates).

The features returned by this iterator have no attributes. However, the GID of the Oracle Spatial geometry is used as the identifier of each feature and this identifier can be used to retrieve additional attributes from the database. See the method `getId()`.

The IlvSDOLayer class

This class implements load-on-demand for a relational Oracle Spatial data source. The default implementation takes an Oracle Spatial layer for which a spatial indexation has been performed and reads its content with a tiling equivalent to the Oracle Spatial tiling.

The following example creates an `IlvSDOLayer` on an Oracle Spatial layer named `ROADS`:

```
IlvSDOConnection connection = new IlvSDOConnection(url, userName, password);
IlvSDOLayer layer = new IlvSDOLayer(connection, "ROADS");
manager.addLayer(layer,-1);
```

The IlvSDOTileLoader class

This class offers additional possibilities when retrieving data from an Oracle Spatial database. These possibilities are meant as a supplement to the default behavior of `IlvSDOLayer`. For example, you may want to add filters to a layer or to have a tiling definition that is different than the Oracle tiling.

The example in the file ***<installdir>*** **`/jviews-maps86/samples/oracle/index.html`**shows how to implement a subclass `IlvSDOTileLoader` that uses spatial queries to retrieve data for a JViews Maps tile.

The IlvDefaultSDOTileLoader class

This class is a subclass of `IlvSDOTileLoader` and is used by the `IlvSDOLayer`. It has some optimizations. For example, the `setTileGroupingCount(short)` method allows you to set the number of tiles that are grouped in one unique query to the database. In fact, each tile corresponds to a Spatial Query. If you have an average of `n` tiles to load each time you want to load on demand, you should use `setTileGroupingCount(n)`, where all the `n` queries are grouped into one unique query that is sent to the database.

Note: If you want to handle special operations on each `IlvMapFeature` retrieved in Load-On-Demand with the `IlvSDOLayer` layer, you have to subclass the `IlvDefaultSDOTileLoader` in order to override the `getFeatureIterator()` method. In this method, you have to return an instance of a subclass of `IlvSDOFeatureIterator` where you have overridden the `getNextFeature()` method (inside which you can perform your specific operations on each `IlvMapFeature` returned by the layer). Finally, you have to set your subclass of `IlvDefaultSDOTileLoader` as the tile loader of the layer.

Class for writing data to an Oracle spatial relational model database

The `IlvSDOWriter` class allows you to write map features into a relational Oracle Spatial database.

The IlvSDOWriter class

The `IlvSDOWriter` class can write any `IlvMapFeatureIterator` whose features have a geometry supported by the relational model of Oracle Spatial and write them to the database as in the following example:

```
IlvSDOWriter writer =
   new IlvSDOWriter(connection.getConnection()
                   "MyLayer",
                   16,
                   new IlvCoordinate(-360d, 90d),
                   new IlvCoordinate(360d, -90d));
 // Creating a source feature iterator.
 IlvShapeFileReader reader = new IlvShapeFileReader(...);
 // Dumping its content to the Oracle layer.
 writer.writeFeatureIterator(reader);
```

The `write()` method of the `IlvSDOWriter` does not write the attributes of the features. If you want to write the attributes of the features, you can subtype the `writeFeature()` method of the `IlvSDOWriter`, after calling `super.writeFeature(feature)`.

The geometries supported by the Oracle Spatial writer are:

- `IlvMapPoint`
- `IlvMapLineString`
- `IlvMapPolygon`
- `IlvMapMultiPoint`
- `IlvMapMultiCurve` for geometries composed of multiple line strings.
- `IlvMapMultiArea` for geometries composed of multiple polygons.

Object relational model classes

Since version 8.1.7, Oracle® Spatial allows spatial data to be georeferenced. Geometries can be georeferenced by associating a spatial reference ID (SRID) to each geometry. Coordinate systems associated to these SRIDs are defined in the table MDSYS.CS_SRS.

JViews Maps can import these reference systems using their OpenGIS WKT (Well Known Text) specifications, found in the MDSYS.CS_SRS table. If your data is not georeferenced, there is no need to change anything. The JViews Maps classes in the `objectmodel` package handle both georeferenced and nongeoreferenced data.

For more information, see the Oracle Spatial documentation (Coordinate Systems section) and *Handling spatial reference systems*.

Classes for reading data from an Oracle Spatial object relational model database

The reader classes for Oracle Spatial object relational model included in the package `ilog.views.maps.format.oracle.objectmodel` are:

- `IlvObjectSDOFeatureIterator` for converting Oracle Spatial layer data into `IlvMapFeature` Objects.
- `IlvObjectSDOLayerMetaData` and `IlvObjectSDODimElement` for representing metadata information for a given Spatial Layer.
- `IlvObjectSDOKeyExtractor` and `IlvDefaultObjectSDOKeyExtractor` for an optimized use of the load-on-demand.
- `IlvObjectSDOLayer` for implementing the load-on-demand for object Oracle Spatial data.
- `IlvDefaultObjectSDOTileLoader`, an optimized subclass of `IlvSDOTileLoader` is used by `IlvObjectSDOLayer`.

The IlvObjectSDOFeatureIterator class

This class reads data from the result of an SQL query to a relational Oracle Spatial layer and converts the data into `IlvMapFeature` objects. The JViews Maps package applications can handle Oracle Spatial data using this class in a transparent way. The following example performs a query, loading data from an Oracle Spatial layer named ROADS:

```
IlvSDOConnection connection = new IlvSDOConnection(url,
                                                   userName,
                                                   password);
connection.createConnection();
IlvObjectSDOFeatureIterator iterator =
   new IlvObjectSDOFeatureIterator(connection.getConnection(),
                                   "select * from ROADS",
                                    // The name of the geometry column.
                                    "GEOMETRY",
                            // No key ID.
                                    null,
                                    // Name of the x-ordinates column.
```

```
                                        "X",
                                        // Name of the y-ordinates column.
                                        "Y");
```

The result set of any query to an Oracle Spatial layer can be used to initialize an `IlvObjectSDOFeatureIterator`, but the column containing the geometry must be in the result set.

The features returned by the iterator can have attributes and coordinate systems attached.

- *Attributes*: they can be retrieved by means of the method `getAttributes()`.

  Any column of the layer that can be interpreted as a String, a float number, or an integer number is translated into attributes and set in the returned map feature. Moreover, if you instantiate the feature iterator with an ID name, the value of these features can be used to retrieve additional attributes (if any) from the database. See the `getId()` method.

  This ID is used in the library essentially for load-on-demand optimization. If you give an ID name when instantiating the iterator, a large geometry that covers more than one tile is loaded just once.

  If you ignore the ID name, the `load()` method of each covered tile fully loads the large geometry.

- *Coordinate Systems*: they can be retrieved by means of the `getCoordinateSystem()` method. This coordinate system is the interpretation of the Well Known Text contained in the MDSYS.CS_SRS table corresponding to the SDO_SRID attached to the geometry read by the iterator. The interpretation is performed by the `ilog.views.maps.srs.wkt` package.

  This behavior can be bypassed by using `setCoordinateSystem(ilog.views.maps.srs.coordsys.IlvCoordinateSystem)`. For example, if you call this method, the coordinate system passed as a parameter is assigned to all the map features returned by the iterator.

  The Oracle SRID of the geometry that has been currently read by the iterator can be obtained by the `getCurrentSRID()`.

The IlvObjectSDOLayerMetaData and IlvObjectSDODimElement classes

These two classes correspond to two data structures defined in Oracle Spatial.

The `IlvObjectSDOLayerMetaData` class corresponds to the data contained in each row of the `(XXX_)SDO_GEOM_METADATA` view. This view contains information, called *metadata*, about Spatial layers. Each Spatial user has the following views available in the schema associated with that user (in Oracle 8.1.6+):

- `USER_SDO_GEOM_METADATA`

  Contains metadata information for all spatial tables owned by the user (schema). This is the only view that the user must keep up-to-date. For instance, the `close()` method of `IlvObjectSDOWriter` updates this view.

- `ALL_SDO_GEOM_METADATA`

  Contains metadata information for all spatial tables on which the user has SELECT permission.

♦ DBA_SDO_GEOM_METADATA

  Contains metadata information for all spatial tables on which the user has SELECT permission (if the user has the DBA role).

Each metadata view has the following definition:

```
 TABLE_NAME    VARCHAR2(32),
 COLUMN_NAME   VARCHAR2(32),
 DIMINFO       MDSYS.SDO_DIM_ARRAY,
 SRID          NUMBER
```

In addition, the ALL_SDO_GEOM_METADATA and DBA_SDO_GEOM_METADATA views have an OWNER column identifying the schema that owns the table specified in TABLE_NAME.

You do not need to build metadata by hand: the IlvObjectSDOUtil method can help you retrieve the metadata directly from the database. For instance, if a Spatial layer is called ROADS, then you can retrieve its metadata as follows:

```
IlvObjectSDOLayerMetaData metadata =
 IlvObjectSDOUtil.GetLayerMetaData(connection.getConnection()
                                   "ROADS",
// You can pass the geometry column name as null: the first
// metadata row that matches the layer name “ROADS” is then
// returned.
                                   null,
// You can pass the metadata view name as null: the default
// is "(USER_)SDO_GEOM_METADATA".
                                   null,
// You can pass the owner name as null: the name of the
// current user is then taken as the owner name.
                                   null);
```

The IlvObjectSDOLayerMetaData class is then composed of the following elements that match the (XXX_)SDO_GEOM_METADATA view: the table name (the name of the Spatial layer), the geometry column name (called COLUMN_NAME in the view, the name of the column of type MDSYS.SDO_GEOMETRY), the owner name, the optional SRID if the Spatial layer is georeferenced, and an array of IlvObjectSDODimElement (called DIMINFO in the view).

The IlvObjectSDODimElement corresponds to the MDSYS.SDO_DIM_ELEMENT data type. The MDSYS.SDO_DIM_ELEMENT data type is defined in Oracle as:

```
Create Type SDO_DIM_ELEMENT as OBJECT (
   SDO_DIMNAME  VARCHAR2(64),
   SDO_LB  NUMBER,
   SDO_UB  NUMBER,
   SDO_TOLERANCE  NUMBER);
```

The DIM element data type describes, for each Spatial Layer, its extent for each dimension. For instance, for typical two-dimensional geometries, the DIM element array of the metadata table has two entries. The first entry describes the first dimension (x, longitude), the second one describes the second dimension (y, latitude).

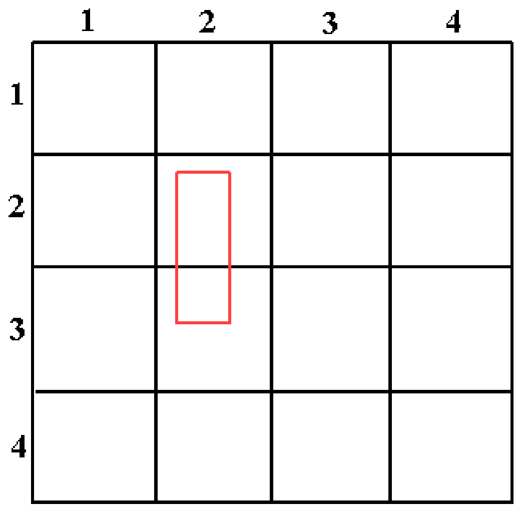
Note: In the `DIM` element array of the metadata table, the elements are supposed to be ordered. For example, the first element of the array is considered as the first dimension of the layer, and so on.

The IlvObjectSDOKeyExtractor and IlvDefaultObjectSDOKeyExtractor classes

The `IlvObjectSDOKeyExtractor` class is an interface associated with feature iterators and tile loaders. Its main purpose is to associate an object with the map feature read by the iterator or the tile loader.

This object is supposed to be unique, so that the next time the feature iterator meets the object, it is not process the geometry in the corresponding row, considering that this geometry has already been loaded.

The goal is to avoid multiple loading of the same geometry. This interface extracts a key from a `ResultSet` using the `extractKey(java.sql.ResultSet)` method. The object returned can be extracted from one Table column or multiple columns as soon as it constitutes a unique object.



Example of Multiple Loading in Load-On-demand

The red rectangle belongs to the tile (2, 2) and (2, 3). When the `load` method of the tile (2, 2) is called, it loads the rectangle. When the `load` method of the tile (2, 3) is called, the rectangle is loaded again as soon as it intersects with this tile too, and the tile loader has no way to know that this geometry has already been loaded. You can avoid this by using the `IlvObjectSDOKeyExtractor` class.

The `IlvDefaultObjectSDOKeyExtractor` is a default class that implements the key extraction behavior from *one* Table column. For example, given the previous schema let us suppose that the Spatial Layer is the following:

```
GEOMETRY  MDSYS.SDO_GEOMETRY,
ID        NUMBER
```

Note: The ID column does not need to be described in Oracle as a key, when it contains unique values.

The IlvDefaultObjectSDOKeyExtractor used with this layer can be easily obtained in the following way:

```
IlvDefaultObjectSDOKeyExtractor extractor =
                  new IlvDefaultObjectSDOKeyExtractor("ID");
```

Then, this extractor can be associated with an IlvObjectSDOLayer, or an IlvDefaultObjectSDOTileLoader.

```
IlvObjectSDOLayer layer =
   new IlvObjectSDOLayer(connection,
                         IlvObjectSDOUtil.GetLayerMetaData(...),
                         1000,
                         1000,
                         "X",
                         "Y",
                         extractor, // used here
                         null);
```

The IlvObjectSDOLayer class

This class implements load-on-demand for an object Oracle Spatial data source. The default implementation takes an Oracle Spatial layer for which a spatial indexation has been performed and reads its content.

The following example creates an IlvObjectSDOLayer on an Oracle Spatial layer named ROADS:

```
IlvSDOConnection connection = new IlvSDOConnection(url,
                                                   userName,
                                                   password);
connection.createConnection();
IlvObjectSDOLayer layer =
     new IlvObjectSDOLayer(connection,
                           // The name of the SDO layer.
                           "ROADS",
                           // Width of a tile in the database
                           // coordinate system.
                           1500,
                           // Height of a tile in the database
                           // coordinate system.
                           1500,
                                  // Special handling of ID.
                           null //Is not required.
```

```
                                );
manager.addLayer(layer,-1);
```

The IlvDefaultObjectSDOTileLoader class

This class is a subclass of `IlvSDOTileLoader` and is used by the `IlvObjectSDOLayer`. It has some optimizations.

For example, the method `setTileGroupingCount(short)` allows you to set the number of tiles that are grouped in one unique query to the database. In fact, each tile corresponds to a Spatial Query, and if you have an average of `n` tiles to load each time you want to load on demand, you should use `setTileGroupingCount(n)`, where all the n queries are grouped into one unique query that is sent to the database once.

Note: If you want to handle some special operations on each `IlvMapFeature` retrieved in load-on-demand with the `IlvObjectSDOLayer` layer, you have to subclass the `IlvDefaultObjectSDOTileLoader` in order to override the `getFeatureIterator` method. In this method, you have to return an instance of a subclass of `IlvObjectSDOFeatureIterator` where you have overridden the `getNextFeature` method (inside which you can perform your specific operations on each `IlvMapFeature` returned by the layer). Finally, you have to set your subclass of `IlvDefaultObjectSDOTileLoader` as the tile loader of the layer.

Another interesting method of this class is the `setRequestParameters(java.lang.String, java.lang.String, java.lang.String, java.lang.String, java.lang.String, int)` method.

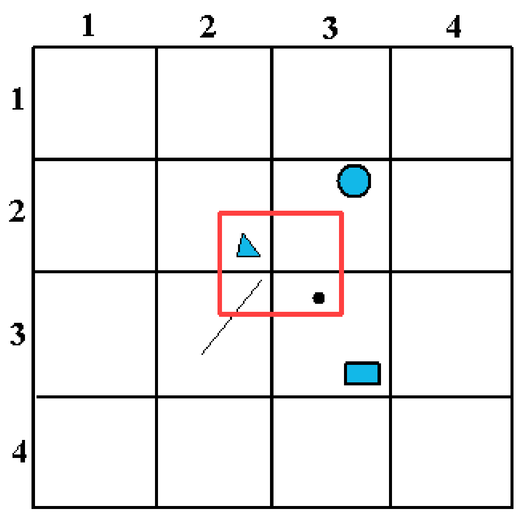
This method allows you, for instance, to set the spatial operator used to query the layer. The default operator is SDO_FILTER.

Tiles shows a Spatial layer using a fixed tiling of level 2. The red rectangle is the area queried by the tile loader. If the SDO_FILTER operator is used (default case), all the geometries belonging to the Oracle Spatial Tiles intersecting with the red rectangle fit the request. In the case of *Tiles*, all the geometries belonging to the tiles (2,2), (2,3), (3,2), and (3,3), for example the line, the point, the triangle, the circle, and the rectangle are retrieved.

You may not want to retrieve the geometries that do not explicitly intersect with the red rectangle (for example, the circle and the rectangle geometries here). In this case, you have two choices.

- ♦ The first choice is to keep the SDO_FILTER operator and to use the `setClippingRequest (boolean)` method in order to let the tile loader perform a bounding box clipping check.
- ♦ The second choice is to use another spatial operator in Oracle which is SDO_RELATE. This operator is to be used with the following parameters: "`querytype=window mask=anyinteract`". This way, the `setClippingRequest()` is not needed anymore so that all the retrieved geometries are the ones that intersect with the red rectangle, for example the point, the triangle, and the line in *Tiles*.

Finally, note that the SDO_RELATE Spatial operator is slower than the SDO_FILTER operator.



Tiles

Class for writing data to an Oracle Spatial object relational model database

This section presents the `IlvObjectSDOWriter` class, which allows you to write map features into an Object Oracle Spatial database.

The class `IlvObjectSDOWriter` can write any `IlvMapFeature` or any `IlvMapFeatureIterator` whose features have a geometry supported by Oracle Spatial 8i (vectorial geometries) and write them to the database as in the following example:

```
IlvSDOConnection connection = new IlvSDOConnection(url,
                                                   userName,
                                                   password);
connection.createConnection();
IlvObjectSDOWriter writer =
    new IlvObjectSDOWriter(connection.getConnection(),
                           "MyLayer",       // Layer name.
                           "GEOMETRY",      // Geometry column.
                           "X",             // X ordinate name.
                           "Y",             // Y ordinate name.
                           true             // Create table.
);
IlvShapeFileReader reader = new IlvShapeFileReader("foo.shp",null);
int saved_objects_count = writer.writeFeatureIterator(reader,
                                                false,    // No attributes
                                                null     );// No SRID
```

```
writer.close(0.0, // Tolerance
             null );// No SRID
```

Note: In the case of the Oracle Spatial Object Model, some auxiliary tables, like the `(USER_) SDO_GEOM_METADATA` view, need to be updated. It is very important to call the method `IlvObjectSDOWriter.close()` once the data has been written through the `write ()` method, so that the database is kept up-to-date.

The `writeFeature(ilog.views.maps.IlvMapFeature, boolean, java.lang.Long)` method of the `IlvObjectSDOWriter` can also write the attributes of the feature.

The method `writeFeature(ilog.views.maps.IlvMapFeature, boolean, java.lang.Long)` has a second argument that can be set to `true` to save the attributes of the specified map feature. This requires that the map feature has an `IlvAttributeInfoProperty` correctly set, describing the attributes that match the Oracle Spatial layer column names. This also requires that map feature has an `IlvFeatureAttributeProperty IlvFeatureAttributeProperty` that fits its `IlvAttributeInfoProperty` and has correct values.

For instance, if you have an Oracle Spatial layer called ROADS that has the following description in the database:

Name	Null?	Type
GEOMETRY		MDSYS.SDO_GEOMETRY
TYPE_DESC		VARCHAR2(512)

The third argument of the `writeFeature()` method allows you to set the SDO_SRID value of the written SDO_GEOMETRY. The value of the SDO_SRID is exactly one of the SRID values of the MDSYS.CS_SRS table, and it represents the corresponding coordinate system for the written geometry.

The following code extract shows how to write features with a unique attribute into the database:

```
IlvSDOConnection connection = new IlvSDOConnection(url,
                                                   userName,
                                                   password);
connection.createConnection();
IlvObjectSDOWriter writer =
   new IlvObjectSDOWriter(connection.getConnection(),
                          "myLayer",
                          "GEOMETRY", "X", "Y", false);
java.lang.String[] names = new String[1];
names[0] = "ATTRIBUTE_NAME";
java.lang.Class[] classes = new Class[1];
classes[0] = java.lang.String.class;
boolean[] nullable = new boolean[1];
nullable[0] = true;
// Creates the attribute info.
```

```
IlvAttributeInfoProperty info =
   new IlvAttributeInfoProperty(names, classes, nullable);

// Sets the attribute to feature.
IlvFeatureAttribute[] attributes = new IlvFeatureAttribute[1];
attributes[0] = new IlvStringAttribute("MY FOO TYPE");
IlvFeatureAttributeProperty prop =
    new IlvFeatureAttributeProperty(info,attributes);
feature.setAttributeInfo(info);
feature.setAttributes(prop);

// Writes the feature.
try {
   writer.writeFeature(feature,true, null); // no SRID
} catch (java.sql.SQLException e) {
   // Error.
   e.printStackTrace();
}
writer.close(0.0, null); // no SRID
```

The writer can update rows in the SDO layer. This is based on a key mechanism, where the row(s) having the value of the given key are updated. The update is done through the following methods from the `IlvObjectSDOWriter` class:

- `updateFeatureAttributes(ilog.views.maps.IlvFeatureAttributeProperty, int)` based on an attribute property where you have to give the position of the key in the attribute list and you can update more than one column at the same time.

- `updateFeatureAttribute(java.lang.String, ilog.views.maps.IlvFeatureAttribute, java.lang.String, ilog.views.maps.IlvFeatureAttribute)` where you update just one column (the new value is the `attributeToUpdate` passed as argument) given a key attribute.

Note: All the subclasses of `IlvMapGeometry` except `IlvMapText`, `IlvMapImage`, and `IlvMapRaster` are supported by the object model writer.

Oracle SDO export

In addition to the `IlvSDOWriter` and `IlvObjectSDOWriter` classes used to store georeferenced objects to an Oracle® database (see *The Oracle spatial reader and writer classes*), you can export part of a map to a SDO database using the map export API (see *Map Export API*). To do so, you just need to set an `IlvSDOExporter` as the vectorial exporter on the `IlvMapExportManager`.

```
IlvMapExportManager exportManager = new IlvMapExportManager();
IlvSDOExporter SDOExporter = new IlvSDOExporter();
exportManager.setVectorialExporter(SDOExporter);

// configure SDO export (connection parameters)
SDOExporter.showConfigurationDialog(null);

// Set the region of the map to export (here we assume that the map is
// in IlvGeographicCoordinateSystem.KERNEL system, i.e. in radians)
exportManager.setExportRegion(-Math.PI,-Math.PI/2, Math.PI,Math.PI/2);


// Export selected map layers of the map (assuming that mapLayersToExport is
an array of
// IlvMapLayer instances)
exportManager.exportMapLayers(mapLayersToExport);
```

The GeoTIFF reader

The GeoTIFF reader (see GeoTIFF format) is based on some extension packages such as Batik Apache™ TIFF Reader, an open source API included in the JViews Maps jar files.

The GeoTIFF format is an extension of the TIFF (Tagged Image File Format) format. The TIFF format is an image file format that allows tags to be inserted in the file. These tags give information about the image contained in the file, such as the resolution, the number of samples per pixel, and so on. The GeoTIFF extension adds specific cartographic tags that give geographic information about the image contained in the file, such as the coordinate system in which the image is represented, and the location of the image in this coordinate system.

The official TIFF specification can be found at:

http://partners.adobe.com/asn/developer/pdfs.tn/TIFF6.pdf

More information about the GeoTIFF format can be found at:

http://www.remotesensing.org/geotiff/geotiff.html

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **/jviews-maps86/samples/mapbuilder/ index.html**

The IlvRasterGeoTiffReader class

The IlvRasterGeoTiffReader class is a GeoTIFF file reader that creates reprojectable, stylable, and pixel-on-demand images.

Creating an image reader

You need first to create an image reader, and then add the image file to be read:

```
IlvRasterGeoTiffReader imageReader = new IlvRasterGeoTiffReader();
imageReader.addMap(tiffFile);
```

The geo-reference information is decoded from the GeoTIFF file, by means of an IlvGeotiffReader.

Creating a data source

Once you have created the reader, you need to create a data source, which should be linked with the manager properties by inserting it into the data source tree:

```
IlvMapDataSource imageDataSource =
IlvRasterDataSourceFactory.buildTiledImageDataSource(manager,imageReader,true,t
rue,null);
IlvMapDataSourceModel dataSourceModel =
IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(imageDataSource);
```

Reading the data

You can then start reading your data:

```
dataSourceModel.start();
```

Starting the data source creates the necessary tiled layers, tile managers, and `IlvRasterIcon` instances to manage the pixel-on-demand feature and the progressive display of the geo-referenced image.

The IlvGeotiffReader class

The `IlvGeotiffReader` class implements the `IlvMapFeatureIterator` interface. The `getNextFeature()` method returns an `IlvMapImage` geometry which contains a `TIFFImage` object. The TIFF image can then be rendered by the `IlvDefaultImageRenderer` to produce an `IlvIcon`, or be transformed by data sources into a tiled `IlvRasterIcon`.

The TIFF reader can take two parameters as arguments: the TIFF file name and a file that contains a connection between the tag describing the coordinate system used by the image and the corresponding WKT string. The reader retrieves the coordinate system of the image in a WKT format if the image contains the appropriate tag, and then retrieves the `IlvCoordinateSystem` through the `IlvWKTCoordinateSystemDictionary` class. This coordinate system is then available through the `getCoordinateSystem(int)` method.

The default constructor, however, only takes the TIFF file name parameter and uses an internal WKT file (wktdictionary.txt) found in IBM® ILOG® JViews jar files.

The reader can be used in the same way as any reader that conforms to the Maps reader framework:

```
IlvGeotiffReader reader = new IlvGeotiffReader(tiffFile);
IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
IlvCoordinateSystem coordSys = reader.getCoordinateSystem();
if (coordSys != null)
    manager.setNamedProperty(new IlvCoordinateSystemProperty(coordSys));
else
    manager.removeNamedProperty(IlvCoordinateSystemProperty.NAME);
IlvCoordinateTransformation tr =
    IlvCoordinateTransformation.CreateTransformation(coordSys, coordSys);
IlvGraphic g = renderer.makeGraphic(f, tr);
manager.addObject(g, false);
```

The TIGER/Line reader

The acronym TIGER® comes from Topologically Integrated Geographic Encoding and Referencing, which is the name for the system and digital database developed at the U.S. Census Bureau to support its mapping needs for the Decennial Census and other Bureau programs.

The TIGER/Line files are a digital database of geographic features, such as roads, railroads, rivers, lakes, legal boundaries, census statistical boundaries, and so on, covering the entire United States. The data base contains information about these features, such as, their location in latitude and longitude, the name of the feature, the type of feature, address ranges for most streets, the geographic relationship to other features, and other related information. TIGER/Line® files are a public product created from the TIGER database of the Census Bureau.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The IlvTigerDataSource class

The `IlvTigerDataSource` class is a data source that reads TIGER/Line files. This data source is filtered so that only selected features can be read from the file, the others being ignored. These features can be supplied in code as an array of strings or can be retrieved from a `IlvFeatureSelectorPanel` configured with the `TigerFeaturesEN.txt` file. This file contains the list of the Census Feature Class Codes (CFCCs) and a description of the corresponding features. Use the following lines to supply the CFCCs in the code:

```
try {
  IlvTigerDataSource source = new IlvTigerDataSource(filename);
  source.setManager(manager);
  source.setCFCCCodeList(new String[]{"A41"});
  source.start();
} catch (Exception e) {
  e.printStackTrace();
}
```

To retrieve the CFCC codes from an `IlvFeatureSelectiorPanel`, you can use the following code, which creates, instantiates and starts a TIGER/Line data source with the selected features:

```
JPanel panel = new JPanel();
panel.setLayout(new BorderLayout());
final IlvFeatureSelectorPanel spanel = new IlvFeatureSelectorPanel(manager,
"TigerFeaturesEN.txt");
JButton button = new JButton("OK");
button.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
  Vector src = new Vector();
  IlvFeatureSelectorPanel.Feature[] features = spanel.getSelectedFeatures();
  ArrayList cfccCodes = new ArrayList();
  if (features != null) {
    for (int j = 0; j < features.length; j++) {
      IlvFeatureSelectorPanel.Feature currentFeature = features[j];
     IlvFeatureSelectorPanel.Feature currentChild;
     int nChildren = currentFeature.getChildCount();
     if (nChildren != 0) {// major
        for (int i = 0; i < currentFeature.getChildCount(); i++) {
        currentChild = (IlvFeatureSelectorPanel.Feature) currentFeature.
getChildAt(i);
        cfccCodes.add(currentChild.getMajorCode() + currentChild.getMinorCode
());
       }
      } else {
      cfccCodes.add(currentFeature.getMajorCode() + currentFeature.getMinorCode
());
       }
      }
     }
     String[] s = (String[])cfccCodes.toArray(new String[0]);
     try {
       source = new IlvTigerDataSource(filename);
       source.setManager(manager);
       source.setCFCCCodeList(s);
       source.start();
     } catch (Exception e1) {
        e1.printStackTrace();     }
}});
panel.add(button, BorderLayout.SOUTH);
JScrollPane pane = new JScrollPane(spanel);
panel.add(pane, BorderLayout.CENTER);
JFrame f = new JFrame();
f.getContentPane().add(panel);
f.pack();
f.setVisible(true);
```

The IlvTigerReader class

The `IlvTigerReader` class reads TIGER/Line files. It can be used independently, but it is usually created through the use of a `IlvTigerDataSource`. The method `recordMatches (ilog.views.maps.IlvFeatureAttributeProperty)` can be overridden to select features to be read or discarded by the TIGER/Line reader.

```
File file = new File(filename);
IlvTigerReader reader =
  new IlvTigerReader(file.toURL().toExternalForm()) {
    public boolean recordMatches(IlvFeatureAttributeProperty properties) {
     if(properties == null)
       return false;
       return properties.getAttribute("CFCC").toString().equals("A41") ;
    }
  };
IlvMapFeature f = null;
IlvCoordinateSystem source = IlvGeographicCoordinateSystem.WGS84;
IlvCoordinateSystem dst = IlvGeographicCoordinateSystem.WGS84;
IlvCoordinateTransformation tr = IlvCoordinateTransformation.
```

```
CreateTransformation(source, dst);
while ((f = reader.getNextFeature()) != null) {
  IlvGraphic g = reader.getDefaultFeatureRenderer().makeGraphic(f, tr);
  manager.addObject(g, false);
}
```

List of CFCC codes

See *http://www.census.gov/geo/www/tiger*.

The DXF reader

There are two ways of reading DXF format files:

♦ Using an `IlvMapDXFReader` instance directly. In this case, you must write all of the code required to render the DXF features into graphic objects, and then add them to the manager.

  The `IlvMapDXFReader` class reads DXF features from a specified DXF file or catalog. It implements the `IlvMapFeatureIterator` interface to iterate over the read features.

♦ Using an `IlvDXFDataSource`. This is a convenient way of performing all the above operations at once and is more integrated with the data model of the map.

  The `IlvDXFDataSource` class provides a convenient way of creating a set of layers containing DXF data in a manager. You can also georeference the geographic objects to create, as DXF datasets are usually non-georeferenced.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

To read the DXF features and create vector data using the DXF readerobject:

1. Create an I`lvMapDXFReader` instance from the path of the DXF file:

```
String DXFpath = " C:/maps/DXF/map.dxf";
IlvMapDXFReader reader = new IlvMapDXFReader(DXFpath);
```

2. Set the transformation to use to render DXF, for example:

```
reader.setDestinationBounds(new Rectangle2D.Double
   (lonMinRad,latMinRad,lonMaxRad,latMaxRad));
```

3. Get the default DXF renderer:

```
IlvFeatureRenderer renderer = reader. getDefaultFeatureRenderer ();
```

4. Iterate over the features, render them, and assign them to a manager:

```
IlvMapFeature feature = reader.getNextFeature();
while(feature != null) {
  // Render map feature into graphic object
  IlvGraphic graphic = renderer.makeGraphic(feature,null);
  // Add this object on the first layer of the manager
  manager.addObject(graphic, 0, false);
  feature = reader.getNextFeature();
}
```

To read DXF features and create vector data using the DXF data source:

1. Create an `IlvDXFDataSource`:

```
String DXFpath = " C:/maps/DXF_0606_ed8/DXFT";
IlvDXFDataSource source = new IlvDXFDataSource(DXFpath);
```

2. Connect this data source with the manager of the view:

```
source.setManager(manager);
```

3. Set the transformation to use to render DXF into geo-referenced objects, for example:

```
source.setDestinationBounds(lonMinRad,latMinRad,lonMaxRad,latMaxRad);
```

   Alternatively you can use a tailored transformation through the use of: `setInternalTransformation(ilog.views.maps.srs.coordtrans.IlvMathTransform)`.

4. Start the DXF data source:

```
source.start();
```

The KML reader and writer

Describes the KML reader and writer and exporting KML files.

In this section

The KML reader and writer

There are two ways of reading KML/ KMZ files:

♦ Using an `IlvKMLReader` instance directly. In this case, you must write all the code required to render the KML features into graphic objects, and then add them to the manager.

  This class reads KML features from a specified KML file or catalog. It implements the `IlvMapFeatureIterator` interface to iterate over the read features.

♦ Using an `IlvKMLDataSource`. This is a convenient way of performing all the above operations at once and is more integrated with the data model of the map.

  The `IlvKMLDataSource` class provides a convenient way of creating a set of layers containing KML data in a manager.

The source code for the Map Builder demonstration, which contains all the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

To read KML features and create vector data using the `IlvKMLReader` object:

1. Create an `IlvKMLReader` instance from the path of the KML catalog:

```
String KMLpath = " C:/maps/KML/places.kmz";
IlvKMLReader reader = new IlvKMLReader(KMLpath);
```

2. Create a default renderer:

```
IlvFeatureRenderer renderer = new IlvDefaultFeatureRenderer();
```

3. Iterate over the features, render them with an appropriate `IlvFeatureRenderer`, and assign them to a manager:

```
IlvMapFeature feature = reader.getNextFeature();
while(feature != null) {
  // Render map feature into the graphic object.
  IlvGraphic graphic = renderer.makeGraphic(feature,null);
  // Add this object on the first layer of the manager.
  manager.addObject(graphic, 0, false);
  feature = reader.getNextFeature();
}
```

To read KML features and create vector data using the `IlvKMLDataSource`:

1. Create an `IlvKMLDataSource`:

```
String KMLpath = "C:/maps/KML_0606_ed8/KMLT";
IlvKMLDataSource source = new IlvKMLDataSource(KMLpath);
```

2. Connect this data source to the manager of the view:

```
source.setManager(getView().getManager());
```

3. Start the KML data source:

```
source.start();
```

Note: JViews Maps does not support KML styles. This is because JViews Maps styles are made for layers whereas KML styles are made for individual objects and are therefore not compatible.

Exporting KML files

You can export part of a map to a KML (or KMZ) file using the Map Export API (see *Map Export API*). To do so, set an `IlvKMLExporter` as the vectorial exporter on the `IlvMapExportManager`. You can also use the raster exporter, if you want to export an overlaid image.

- If you set a KML file name, all images used inside that file are saved at the same location, with names such as *image0.png*.
- If you set a file name that ends with `.kmz`, all the files are zipped into a single `kmz` file:

```
IlvMapExportManager exportManager = new IlvMapExportManager();
IlvKMLExporter KMLExporter = new IlvKMLExporter();
exportManager.setVectorialExporter(KMLExporter);
exportManager.setRasterExporter(KMLExporter);

// Configure KML export (file name).
KMLExporter.setFileName("export.kml");

// Set the region of the map to export.
exportManager.setExportRegion(-Math.PI,-Math.PI/2, Math.PI,Math.PI/2);

// Export selected map layers of the map.
IlvMapLayer mapLayersToExport[]={/* table of map layers to export*/...};
exportManager.exportMapLayers(mapLayersToExport);
```

The DEM/GTOPO30 reader

GTOPO30 is a global digital elevation model (DEM) with a horizontal grid spacing of 30 arc seconds (approximately 1 kilometer). It covers the full extent of latitude from 90 degrees south to 90 degrees north, and the full extent of longitude from 180 degrees west to 180 degrees east. The vertical units represent elevation in meters above mean sea level. In the DEM, ocean areas have been masked as "no data" and have been assigned a value of -9999.

A full sample of a GTOPO30 database can be downloaded from: *http://edcdaac.usgs.gov/gtopo30/gtopo30.asp*.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **/jviews-maps86/samples/mapbuilder/index.html**.

The `IlvGTopo30Reader` class is a file reader that creates reprojectable, stylable and pixel-on-demand images that can provide altitude data.

To create images:

1. First create an image reader and add the image file you want to be read:

```
IlvGTopo30Reader imageReader = new IlvGTopo30Reader();
imageReader.addMap(gtopoFile);
```

2. Create a data source and link it with the manager properties by inserting it into the data source tree:

```
IlvMapDataSource imageDataSource = IlvRasterDataSourceFactory.
   buildTiledImageDataSource(manager,imageReader,true,true,null);
IlvMapDataSourceModel dataSourceModel =
   IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(imageDataSource);
```

3. Start reading your data:

```
dataSourceModel.start();
```

Starting the data source creates the necessary tiled layers, tile managers and `IlvRasterIcon` instances to manage the pixel-on-demand feature and the progressive display of the geo-referenced image.

This icon associates altitude properties that can be used as altitude data sources, see *Using the altitude provider property*.

The Web Map Server reader

The OpenGIS® Web Map Service (WMS standard) Implementation Specification produces maps of spatially referenced data dynamically from geographic information. This international standard defines a map to be a portrayal of geographic information as a digital image file suitable for display on a computer screen. The OpenGIS standard provides three operations (`GetCapabilities`, `GetMap`, and `GetFeatureInfo`) in support of the creation and display of registered and superimposed map-like views of information that come simultaneously from multiple remote and heterogeneous sources. For more information about the OpenGIS standard, see *http://www.opengeospatial.org/*.

There are two ways of reading images from a Web Map Server (WMS):

- ♦ Using an `IlvWMSReader` instance directly. In this case, you must write all the code required to render the WMS features into graphic objects and add them to the manager.

- ♦ Using an `IlvWMSDataSource`. This is a convenient way of performing all the above operations at once and is more integrated with the data model of the map.

The source code for the Map Builder demonstration, which contains all the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The `IlvWMSReader` class reads image features from a specified Web Map Server URL. It implements the `IlvMapFeatureIterator` interface to iterate over the read features.

The `IlvWMSDataSource` class provides a convenient way of creating a set of layers containing images retrieved from a Web Map Server in a manager.

To read WMS features from a Web Map Server using the `IlvWMSReader` class and create raster data:

1. Create an `IlvWMSReader` instance from a server URL. A WMS server URL can be the full http WMS request for capabilities, or at least the WMS request URL for this server. This information is dependent on the server itself and is usually available in the server's documentation. For instance :

```
URL url = new URL("http://wms.jpl.nasa.gov/wms.
cgi?request=GetCapabilities"); // "http://wms.jpl.nasa.gov/wms.cgi" will
 work as well
IlvWMSReader reader = new IlvWMSReader(url);
```

2. Set the name of the layers to be rendered. The layer names can be retrieved from the server capabilities:

```
String[] layers = reader.getAvailableLayers();
reader.setLayerNames(new String[]{layers[0]});
```

3. Set the transformation to use to render WMS images, for example:

```
IlvCoordinateSystem cs = IlvCoordinateSystemProperty.
   GetCoordinateSystem(manager);
reader.setTransformation(IlvCoordinateTransformation.CreateTransformation
   (cs, IlvGeographicCoordinateSystem.KERNEL));
```

4. Iterate over the features, render them with an appropriate `IlvFeatureRenderer`, and assign them to a manager:

```
IlvMapFeature feature = reader.getNextFeature();
IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
while(feature != null) {
  // Render the map feature into a graphic object.
  IlvGraphic graphic = renderer.makeGraphic(feature,null);
  // Add this object to the first layer of the manager.
  manager.addObject(graphic, 0, false);
  feature = reader.getNextFeature();
}
```

To read WMS features using the `IlvWMSDataSource` class :

1. Create an `IlvWMSDataSource` instance from a server URL. A WMS server URL can be the full http WMS request for capabilities, or at least the WMS request URL for this server. This information is dependent on the server itself and is usually available in the server's documentation. For instance:

```
URL url = new URL("http://wms.jpl.nasa.gov/wms.
cgi?request=GetCapabilities"); // "http://wms.jpl.nasa.gov/wms.cgi" will
 work as well
IlvWMSDataSource source = new IlvWMSDataSource(url);
```

2. Set the layers to be retrieved:

```
IlvWMSReader reader = source.getReader();
String[] layers = reader.getAvailableLayers();
source.setLayers(new String[]{layers[0]});
```

3. Optionally set tiling parameters, to enable load-on-demand and better image resolution when zooming in:

```
source.setTilingParameters(true,5,5);
```

4. Connect this data source with the manager of the view:

```
source.setManager(manager);
```

5. Finally, start the WMS data source:

```
source.start();
```

The SVG reader

There are two ways of reading SVG files:

♦ Use an `IlvMapSVGReader` instance directly. When you use `IlvMapSVGReader`, you must write all of the code required to render the SVG features into graphic objects, and to add them to the manager.

  This class reads SVG features from a specified SVG file or catalog. It implements the `IlvMapFeatureIterator` interface to iterate over the features to be read.

♦ Use an `IlvSVGDataSource`. This is a convenient way of performing all of the above operations at once. Using `IlvSVGDataSource` is better integrated with the data model of the map.

  The `IlvSVGDataSource` class provides a convenient way to create a set of layers containing SVG data in a manager.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

To read SVG features and create vector data:

1. Create a new `IlvMapSVGReader` instance using the path to the SVG catalog.
2. Set the transformation to use to render the SVG data. The following code example shows an specimen transformation:
3. retrieve the default SVG renderer.
4. Iterate over the features, render them, and assign them to a manager:

```
String SVGpath = " C:/maps/SVG/map.svg";
IlvMapSVGReader reader = new IlvMapSVGReader(SVGpath);
reader.setDestinationBounds(lonMinRad,latMinRad,lonMaxRad,latMaxRad);
IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
IlvMapFeature feature = reader.getNextFeature();
while(feature != null) {
  // Render map feature into a graphic object
  IlvGraphic graphic = renderer.makeGraphic(feature,null);
  // Add this object to the first layer of the manager
  manager.addObject(graphic, 0, false);
  feature = reader.getNextFeature();
}
```

To read SVG features and create vector data:

1. Create a new `IlvSVGDataSource` instance.
2. Connect this data source with the manager of the view.
3. Set the transformation to render the SVG data into geo-referenced objects.
4. Start the SVG data source.

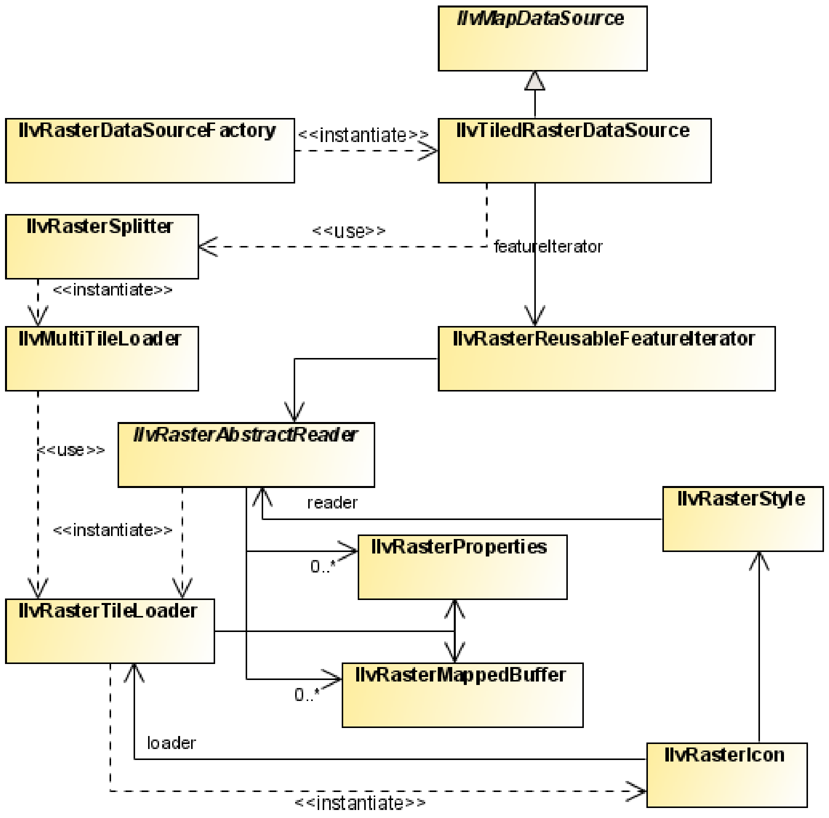
Raster image management

Describes the management of raster images including tile loading, subsampling, persistence, and storage.

In this section

Raster image management classes

The class diagram for raster image management is shown in *Raster Image Management UML Diagram*.



Raster Image Management UML Diagram

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

The IlvRasterAbstractReader class

The IlvRasterAbstractReader class contains the methods for all readers of raster image formats. It contains a built-in tiling mechanism for tiling the image to be displayed.

Geo-referenced image list

The IlvRasterAbstractReader class is an abstract class that handles an indexed list of geo-referenced images. Each image consists of two objects:

♦ The raster data (the pixel table values) that are managed by an IlvRasterMappedBuffer .

♦ The raster properties (color model, size, location, and so on) that are stored in an IlvRasterProperties.

Subclasses should provide a method (often called addImage) that creates both an IlvRasterProperties and an IlvRasterMappedBuffer and adds them to the list using the addRaster method.

Projection transformation

The IlvRasterAbstractReader needs the projection of the original raster and the projection to use in the view. From these it can dynamically (on-demand) compute every pixel visible in the tile to be loaded. Sub-classes must implement getInternalTransformation(int) in order to provide the coordinate system in which the original data is stored. As the getLowerRightCorner() and getUpperLeftCorner() methods return the IlvRasterProperties bounds, these bounds must be provided in the same coordinate system.

Image metadata

Image metadata subclasses also have to implement the getProperties(int) method to provide the IlvFeatureAttributeProperty attached to each of the images.

Image tiling and subsampling

Image data is loaded asynchronously (potentially in another thread) into the view using the tile loading mechanism of the `getTileLoader(int, boolean)` method, which returns an `IlvRasterTileLoader`.

The tile loader is created through the `createRasterTileLoader(ilog.views.maps.raster.IlvRasterProperties, ilog.views.maps.raster.IlvRasterMappedBuffer, int, boolean)` method, which you can override if you want to implement a specific mechanism (for example, to load some of the file content on demand). By default, this method returns either an `IlvRasterSubsamplingLoader` or an `IlvRasterTileLoader` according to the value of the `subsampling` parameter.

The `getDefaultFeatureRenderer()` method returns an `IlvRasterImageRenderer` that creates an `IlvRasterIcon` linked to the `IlvRasterTileLoader`.

The loader `getScaledImageProducer(int, ilog.views.IlvRect)` method can be invoked by the `IlvRasterIcon` when the zoom factor changes, or when a styling parameter changes the image properties (such as the color model) in order to recreate the Java™ `Image` object displayed on the view.

Persistence of images

The base reader is made persistent through the implementation of the `IlvPersistentObject` interface. This implies implementing a specific constructor (to retrieve data from an input stream) and writing methods.

To manage dataless persistence, it is also necessary to implement a reload method, able to read the raw files from serialized information (such as file names), see *Writing a raster reader for DEM data*.

The IlvRasterMappedBuffer class

The `IlvRasterMappedBuffer` class manages raster data for readers. It uses temporary files to store the data when it is not needed.

The buffer stores a table of [width x height] pixel values. Pixel values can use different primitive types, such as `byte`, `short` or `integer` values. The pixel value type must be compatible with the color model of the associated `IlvRasterProperties` object. The buffer is backed up by two mechanism to reduce memory consumption:

♦ Memory mapped temporary files: image data is saved in a temporary file that is then mapped in memory (as in the case of the `java.nio.MappedByteBuffer` class), giving access times close to those of direct memory storage.

♦ Random access files: image data is saved in a temporary file. When a pixel value needs to be read, it is accessed through a `java.io.RandomAccessFile`.

If machine dependant constraints prevent the memory mapped mechanism from succeeding (on 32 bits machines, for example, this happens when more than 2-4GB of images have been loaded), the random access file mechanism is put in place immediately.

The `IlvRasterTemporaryFileManager` class handles temporary files. The temporary file folder can become cluttered, for example, after an application has been interrupted brutally by errors, exceptions, or even debugging session stops. To clean the temporary file folder, you can call:

```
IlvRasterTemporaryFileManager.removeAllFiles();
```

This call searches for all temporary files created by JViews Maps applications and try to remove them. Temporary files that are in use, for example, if another application is running, are not removed.

Note: When using the `IlvRasterMappedBuffer` class in non-signed applets, the creation of temporary files is forbidden. Before loading the data into the model (usually through an `addMap` call), the application must change the memory policy management to avoid the creation of temporary files. This is done using:

```
IlvRasterMappedBuffer.setDefaultMemoryPolicy(IlvRasterMappedBuffer.
USE_MEMORY);
```

With this call, all operations use direct memory to store pixel data. An alternative solution is to sign the applet to allow temporary file creation.

You can even improve this mechanism by providing a Just In Time (JIT) loader. Instead of loading and filling the image bytes at creation, you can provide an instance of `JITLoader`, that loads the image bytes only when the image is about to be displayed. For example:

```
IlvRasterMappedBuffer.JITDataLoader jitLoader = new
   IlvRasterMappedBuffer.JITDataLoader() {
      public void loadData(IlvRasterMappedBuffer source, IlvRasterProperties
          properties) {
             System.out.println("loading "+properties.getBaseName());
             // load the data...
          }
      public void unloadData(IlvRasterMappedBuffer source,
         IlvRasterProperties properties) {
         // raz the stored bytes.
      source.setBytes(new byte[0]);
      }
   };
source.setLoader(jitLoader);
```

If your memory policy is `USE_MAP`, and you use a JIT loader (or a raster format that internally uses one, such as GeoTIFF, DTED® or CADRG), you should be aware that the loading will first happen in memory. To save load time, the creation of the disk-mapped memory will happen in a background thread. This means that the memory necessary to store the image pixels will be kept for some time in the JVM™ , until the disk map is ready to use. In some cases, for example when your application loads lots of large images at the same time, this can lead to out of memory errors, which are by default ignored (the image load restarts again after some time). See also `callJITLoaderIfNecessary` and `setLoadRetryingOnOutOfMemory` for more control on this.

The IlvRasterProperties class

The `IlvRasterProperties` class collects information about the raster image to be tiled and the image to be displayed. This information comprises:

- ♦ The raster bounds (coherent with the coordinate system chosen).
- ♦ The pixel density (in most cases, this is the ratio between image size and pixel count in each direction).
- ♦ The number of blocks in the raster and the number of pixels in a line and in a column of a block. (usually 1).
- ♦ The ordering of the pixels in the lines and columns of a block.
- ♦ The size that the destination tiles of the raster should adopt.
- ♦ The number of pixels the raster has in both directions.
- ♦ The transparent pixel value (if any).
- ♦ The `ColorModel` to use with the image.

To create and setup raster properties, you can write, for example:

```
IlvAdjustableDelegateColorModel csm=new
IlvAdjustableDelegateColorModel(myColorModel);
IlvRasterProperties p=new IlvRasterProperties(csm);
p.setX(xmin);
p.setY(ymax);
p.setWidth(xmax-xmin);
p.setHeight(ymin-ymax);
p.setColumnPixelCount(nbCols);
p.setLinePixelCount(nbRows);
p.setTransparentColorIndex(noDataValue);
p.setHorizontalPixelDensity(p.getWidth() / p.getColumnPixelCount());
p.setVerticalPixelDensity(p.getHeight() / p.getLinePixelCount());
```

In the example above, the `IlvAdjustableDelegateColorModel` is needed to provide brightness, saturation and contrast settings for the user.

If the variables `xmin`/`xmax` and `ymin`/`ymax` are in longitudes, latitudes in radians, and the attached coordinate system is `IlvGeographicCoordinateSystem`, the reader uses the WGS84 transformation to be coherent with these raster properties.

```
 IlvCoordinateTransformation.CreateTransformation(
     IlvGeographicCoordinateSystem.KERNEL,
     IlvGeographicCoordinateSystem.WGS84).getTransform();
```

Graphical User Interface beans and interactors

JViews Maps provides a number of Graphical User Interface (GUI) JavaBeans™ components that you can use to build your own application. These components are either Bean-like or interactor-like.

Bean-like components are the standard bricks from which the main application window should be composed. These components are ready to put into an Integrated Development Environment (IDE), which means that you can import and use them in any JavaBeans IDE compliant editor without writing a single line of code. The Beans included in the JViews Maps package provide the basic features with which to view and navigate through a map.

Interactor-like components are usually associated with action buttons on the toolbar and trigger an action when you interact with the view (for example, putting additional data on the map or creating additional Beans).

For more information about Beans and interactors, see *Using the GUI beans* and *Map GUI interactors*.

For more information about how to use the Beans in an IDE, see Creating a simple applet using IBM® ILOG® JViews Beans in *The Essential JViews Framework*.

The Beans and interactors are listed in *GUI Beans and Interactors*.

GUI Beans and Interactors

Package\Bean	Class	Description
ilog.views.maps.beans		
Map Overview	`IlvJOverview`	Displays an overview of a map in a manager view.
Area of Interest Panel	`IlvJAreaOfInterestPanel`	Displays selected areas of a map of particular interest, which you can select and redisplay.
Scale Bar	`IlvJAutomaticScaleBar`	Displays the black and yellow scale bars below a manager view.
Scale Control Bar	`IlvJMapScaleControl`	Controls the numerical value of the map scale, for

Package\Bean	Class	Description
		example: 1/100,000.
Zoom Control Panel	`IlvJAdvancedZoomControl`	Displays a slider type zoom control in the Overview Panel.
Legend Panel	`IlvMapLegendPanel`	Displays a list of map elements and an explanation of what each element represents.
Coordinate System Editor Panel	`IlvJCoordinateSystemEditorPanel`	Displays a panel for viewing and editing coordinate systems.
Display Preferences Editor Panel	`IlvJDisplayPreferencesEditorPanel`	Displays a panel for viewing and editing Bean preferences such as the scale bar and coordinate viewer.
Coordinate Viewer	`IlvJMouseCoordinateViewer`	Displays the mouse coordinates when the mouse is on top of a manager view displaying a map.
Map Layer Tree Panel	`IlvLayerTreePanel`	Displays a panel for editing a map layer tree.
Data Source Tree	`IlvDataSourcePanel`	Displays a set of data

Package\Bean	Class	Description
		sources as a tree.
Compass	`IlvJCompass`	Displays a compass that shows the direction of the geographic or cartographic north of a map displayed in a manager view.
Annotation toolbar	`IlvMapAnnotationToolBar`	Displays a toolbar which allows the creation of map annotations.
Maps toolbar	`IlvJMapsManagerViewControlBar`IlvJMapsManagerViewControlBar	Displays a customizable toolbar for maps applications.
ilog.views.swing		
Toolbar	`IlvManagerViewControlBar`	Displays a customizable toolbar to which buttons can be added.
Multithread Monitor	`IlvThreadedActivityMonitorPanel`	Displays the progress of managed, multiple threaded activities.
ilog.views.maps.projection		
Alpha Property Editor	`IlvAlphaPropertyEditor`	Enables editing of layer transparency.
Color Model Property Editor	`IlvColorModelPropertyEditor`	Displays a color model

Package\Bean	Class	Description
		property editor.
Percent Property Editor	`IlvPercentPropertyEditor`	Controls the brightness, saturation, and contrast of a map.
Latitude Editor	`IlvLatitudeEditor`	Displays an editor for editing latitude values in degrees, minutes and seconds.
Longitude Editor	`IlvLongitudeEditor`	Displays an editor for editing longitude values in degrees, minutes and seconds.
Coordinate Panel Factory	`IlvCoordinatePanelFactory`	Generates a GUI that enables selection of a rectangle or point on a map.
ilog.views.maps.propertysheet		
Map Style Property Sheet	`IlvMapStylePropertySheet`	Provides editing support for the map layer properties.
ilog.views.maps.interactor		
Map Pan	`IlvMapPanInteractor`	Drags the map around the manager view to the position required.
Zoom Rectangle	`IlvMapZoomInteractor`	Zooms in on an area of a map contained

Package\Bean	Class	Description
		within a rectangle drawn by the user.
Manager View Rotate Interactor	`IlvManagerViewRotateInteractor`	Enables interactive rotation of a view.
Magnify Interactor	`IlvMagnifyInteractor`	Magnifies part of a manager view when the mouse is dragged over it.
Continuous Zoom Interactor	`IlvContinuousZoomInteractor`	Zooms in and out continuously while the mouse is pressed.
See Through Interactor	`IlvSeeThroughInteractor`	See through some layers interactively.
ilog.views.maps.measures		
Distance Measuring	`IlvMakeMeasureInteractor`	Displays a line drawn on a map. The line is the shortest path between 2 points on the surface of the earth (orthodromy).

Geodetic computation and date line wrapping

The `IlvGeodeticComputationIlvGeodetic` class is useful for calculating distances, azimuths, and point coordinates. Calculations of this kind are needed to compute the shortest distance between two objects positioned on the globe, to draw the shortest trajectory of a plane, to obtain the current distance of a plane to its destination, and so on. Computations of class `IlvGeodeticComputation` are very useful when JViews Maps objects have to be placed or moved with precision over a given geographic map to simulate real-world scenarios. Geodetic computations include date line wrapping.

Utilities

The `IlvMapUtil` class contains a set of static utility methods:

- ♦ Geometric methods, such as intersection computation.
- ♦ Resources access methods.
- ♦ Generic computations on the view, such as scale.
- ♦ Garbage collecting method.

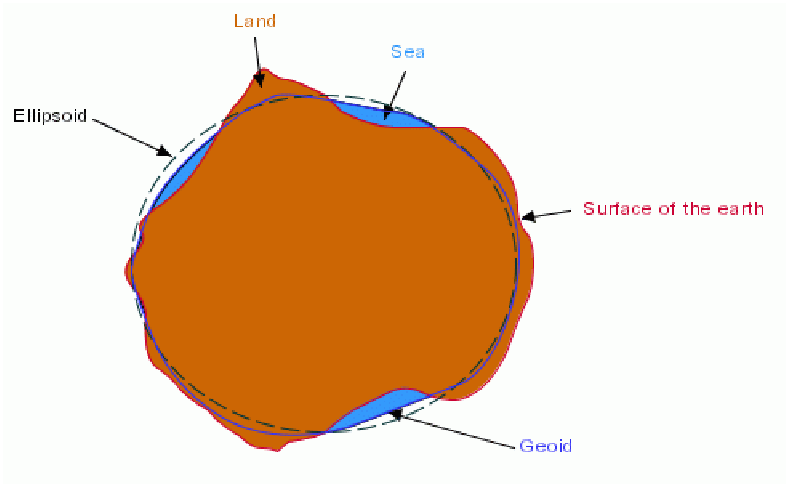
Ellipsoid and geodetic datums

Explains how the earth is modeled and the use of ellipsoids, geodetic datums, map projections, and coordinate systems.

In this section

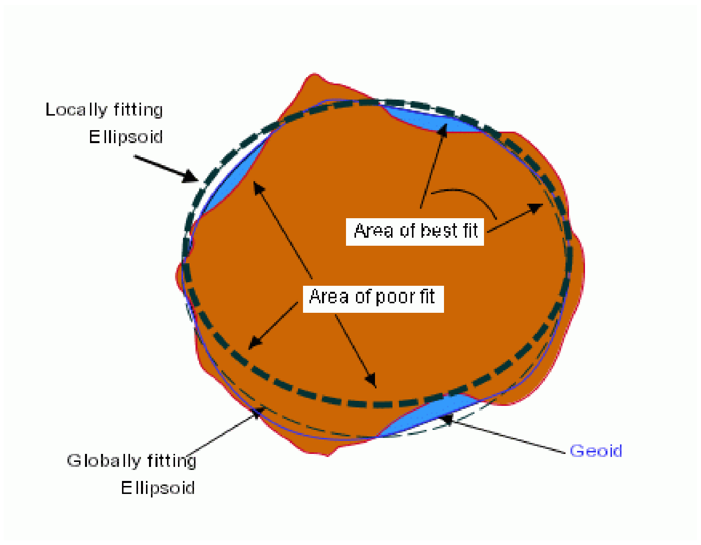
Modeling the earth

The surface of the Earth is complex, even if the topography is removed so that only the *geoid* (the gravity surface approximating to mean sea level) is taken as the shape of the Earth. In fact, because of Earth’s internal composition, which leads to local gravity anomalies, the geoid is very irregular. Therefore, the most widely used model of the Earth approximates the geoid to an oblated ellipsoid (the spheroid), or even simpler, a sphere.



Model of the earth

Of course since the use of an ellipsoid is an approximation, there is no universal ellipsoid that fits the geoid everywhere. For a given location, the best fitting ellipsoid can have different dimensions compared to other locations. That is why a large number of ellipsoids are used in maps.



Area of best fit

Defining an ellipsoid is not sufficient. It is also necessary to define the spatial relationship (position and orientation) between the ellipsoid and the geoid. This is achieved through the definition of a geodetic datum, or horizontal datum, giving the position and the orientation of the ellipsoid relative to the center of the Earth.

Ellipsoids

Overview of ellipsoids

Ellipsoids are used to represent the shape of the Earth. For many applications, and especially in small-scale mapping (typically world maps), the Earth can be represented as a sphere.

Most of the projections supplied in the `ilog.views.maps.projection` package assume by default that the Earth is a sphere with a radius of approximately 6371 kilometers. However, because the Earth rotates on its axis, it is slightly flattened at the poles, and is therefore better approximated by an ellipsoid rotating on the polar axis. Ellipsoidal projections are used for accurate, large-scale maps and flat *coordinate systems*. However, in very large scale maps, representing a continent or the whole planet, it is recommended that you use spherical projections. Indeed, the elliptical form of most of the projections in the projection package is accurate only for a few degrees of latitude or longitude around the projection center.

Defining new ellipsoids

Ellipsoids can be described by many parameters. The ellipsoids provided in the JViews Maps package are defined by two parameters:

♦ The equatorial radius or semi-major axis (a) of the ellipsoid.

♦ The eccentricity squared of the ellipsoid.

 If the eccentricity squared is `null`, the ellipsoid is a sphere.

Defining a spherical ellipsoid

If only one parameter is provided, the ellipsoid is assumed to be a sphere. The following example defines a sphere with a radius of 6 000 kilometers (6000000 meters).

```
IlvEllipsoid ellipsoid = new IlvEllipsoid(6000000D);
```

Most of the mapping applications use the ellipsoid `IlvEllipsoid.SPHERE` that defines a sphere having dimensions very close to those of the Earth. Generally, the selected ellipsoid should be as close as possible to the actual shape of the Earth as far as the region to be represented is concerned. The radius of the sphere is expressed in meters.

The following example defines an ellipsoid with an equatorial radius of 6000 kilometers and an eccentricity squared of 0.0067:

```
IlvEllipsoid ellipsoid = new IlvEllipsoid(6000000D,0.0067D);
```

If you prefer to provide some other parameter than the eccentricity squared, you can use the conversion methods provided by the `IlvEllipsoid` class.

The following example defines an ellipsoid with an equatorial radius of 6000 kilometers and a polar radius of 5900 kilometers:

```
IlvEllipsoid ellipsoid = new IlvEllipsoid(6000000D,
                      IlvEllipsoid.ESFromPolarRadius(6000000D, 5900000D));
```

The polar radius is converted to an eccentricity squared value with the `ESFromPolarRadius()` method.

The class `IlvEllipsoid` provides the following conversion methods for polar radius and flattening:

- ♦ `ESFromPolarRadius(double, double).`
- ♦ `ESFromFlattening(double).`

Predefined ellipsoids

The `IlvEllipsoidCollection` class manages lists of predefined ellipsoids. A list of predefined ellipsoids, or *kernel collection*, can be retrieved using the method `GetKernelCollection()`.

Ellipsoid collections are read from XML files containing the ellipsoid definitions. The Document Type Definition (DTD) for these definition files is as follows:

```
<!DOCTYPE ellipsoid-list [
   <!ELEMENT ellipsoid EMPTY>
  <!ATTLIST ellipsoid
     a       CDATA   #IMPLIED
     b       CDATA   #IMPLIED
     invf    CDATA   #IMPLIED
     name    CDATA   #REQUIRED
     comment CDATA   #IMPLIED
  >

  <!ELEMENT ellipsoid-ref EMPTY>
  <!ATTLIST ellipsoid-ref
     ref     CDATA   #REQUIRED
     id      CDATA   #REQUIRED
  >

  <!ELEMENT ellipsoid-list (ellipsoid|ellipsoid-ref)* >
]>
```

In an ellipsoid definition file, you can find:

- ♦ An ellipsoid

  Defined either by its name, or by its semi-major axis `a` and inverse flattening `invf`, or its semi-major axis `a` and semi-minor axis `b`.

- ♦ An alias for an ellipsoid

  Defined in the kernel collection of Maps. For an alias, the required information is the `id` of the ellipsoid (the name used to retrieve the ellipsoid) and the name of the ellipsoid in the kernel. When retrieving an ellipsoid alias from a collection, the ellipsoid is searched for using its `id` as a key, while the name of the returned ellipsoid is the one defined in the kernel collection.

The following XML file defines the Clarke 1880 ellipsoid, modified for IGN. This ellipsoid will be available as “Clarke 1880 (IGN)”. Then you must set “WGS 1984” as an alias for the kernel “WGS 84” ellipsoid:

```
<ellipsoid-list>
  <ellipsoid name="Clarke 1880 (IGN)"
     comment="Clarke 1880 (Modified for IGN)"
     a="6378249.2"
     invf="293.4660213"
  />

  <ellipsoid-ref id="WGS 1984"
                 ref="WGS84"
  />

</ellipsoid-list>
```

Geodetic datums

Datums

Geodetic datums (or “horizontal datums”) help in the process of approximation of the Earth surface by providing a translation and an optional rotation of an ellipsoid relative to an arbitrary center of Earth. These translation and rotation parameters are called the “to WGS84” parameters, with reference to the WGS84 datum that defines no translation and no definition.

The most used datums specify only a translation from the center of the Earth and no rotations. That is why JViews Maps does not provide standard support for datums with rotations, though it is possible to use them in coordinate transformations. See the section *Affine Transform*.

Defining a new horizontal datum

JViews Maps supports the definition of three-parameter datums, that is datums defined by the shift on three axes. This kind of datum is represented by the `IlvHorizontalShiftDatum` class.

A horizontal shift datum is defined by the following elements:

- Its name

  The name should be set to `null` in case it is not defined.

- The area of definition

  This is a String describing the area of use of the datum.

- A default ellipsoid

  Although the datums can be used with different ellipsoids, they were designed to work with this specific ellipsoid.

- The three-axis shift parameters (expressed in meters)

  To define a new horizontal shift datum, initialize a new instance with the three parameters, as follows:

```
IlvHorizontalDatum datum =
   new IlvHorizontalShiftDatum("European 1979","Mean for europe",
      IlvEllipsoidCollection.GetKernelCollection().getEllipsoid("intl"),
         86, -98, -119)
```

Predefined datums

The `IlvHorizontalDatumCollection` class manages lists of predefined horizontal datums. The list of predefined datums in the JViews Maps package, or *kernel collection*, can be retrieved using the `GetKernelCollection()` method.

Like Ellipsoid collections, horizontal datum collections are read from XML files containing the definitions. The DTD for these definition files is as follows:

```
<!DOCTYPE datum-list [
  <!ELEMENT datum EMPTY>
  <!ATTLIST datum
     name    CDATA   #REQUIRED
     region  CDATA   #IMPLIED
     ellipsoid  CDATA #REQUIRED
     dx      CDATA   #REQUIRED
     dy      CDATA   #REQUIRED
     dz      CDATA   #REQUIRED
     ex      CDATA   #IMPLIED
     ey      CDATA   #IMPLIED
     ez      CDATA   #IMPLIED
     ppm     CDATA   #IMPLIED
  >

  <!ELEMENT datum-ref EMPTY>
  <!ATTLIST datum-ref
     ref     CDATA   #REQUIRED
     id      CDATA   #REQUIRED
  >

  <!ELEMENT datum-list (datum|datum-ref)* >
]>
```

In a horizontal datum definition file, you can find:

- A horizontal datum

  Defined by its name and its parameters. The required parameters are the ellipsoid to be used with this datum and the three-axis shifts dx, dy and dz. Note that you can define the parameters for seven-parameter datums, but the ex, ey, ez and ppm fields are not used in the current version of JViews Maps.

- An alias for a horizontal datum

  Defined in the kernel collection of Maps. For an alias, the required information is the `id` of the datum and the reference of the datum. When retrieving a datum alias from a collection, the datum is searched for using its `id` as a key, while the name of the returned datum is the one defined in the kernel collection.

The following XML file defines the *Afgooye* datum, while setting an alias for the NAD 27 datum

```
<datum-list>
  <datum name="Afgooye"
         region="Somalia"
         ellipsoid="krass"
         dx="-43"
         dy="-163"
         dz="-45"
  />
```

```
   <datum-ref id="NAD27"
              ref="NAD27 (CONUS)"
   />
</datum-list>
```

Note: JViews Maps comes with a default list of datums. Over time, many datums were tuned for given regions of use, while keeping the same name. This causes many datums to have the same name but different parameters. In applications where strict conformance to some datums is needed, you should use your own datum definition instead of relying on the list provided in JViews Maps.

Map projections

Tells you all about map projections.

In this section

Introducing map projections

A map is a projected representation of the Earth, or part of it, on a flat surface, which can be a piece of paper or a computer screen. Since the Earth has an ellipsoidal shape, it is best represented as a “globe”, and attempts to portray it by projecting its points onto a flat surface always result in some form of distortion in the regions that are far from the projection center. In other words, it is impossible to faithfully represent all the properties of the Earth, such as distances, shapes, and directions, on the same map. To minimize distortion, many different types of projections have been developed over the years. While certain projections preserve distances, others maintain shapes or angles. When creating a map, you have to choose the projection system that is best suited to the area to be represented or to the particular interests that your map application is designed for.

Projections can be classified into three main categories:

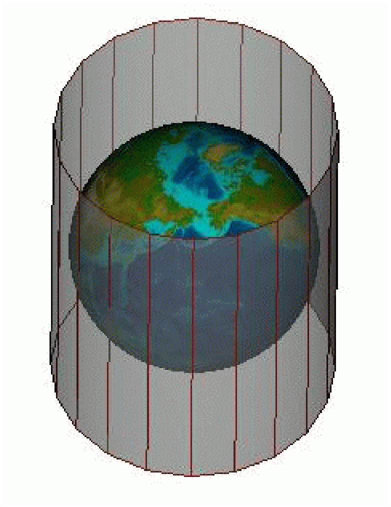
- *Cylindrical projections*
- *Conic projections*
- *Azimuthal projections*

Projections can also be:

- *Equal area or conformal projections*

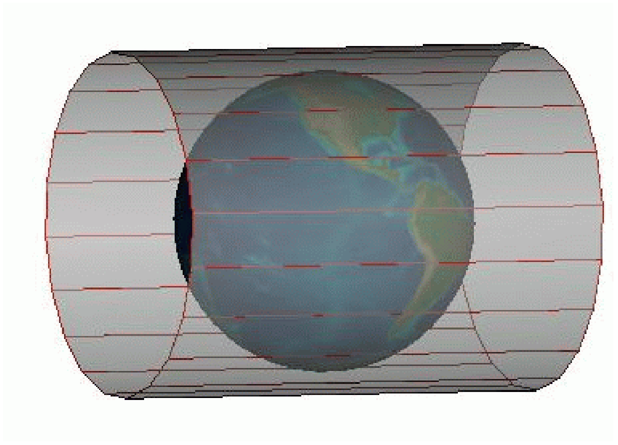
Cylindrical projections

A cylindrical projection is obtained by wrapping a large, flat plane around the globe to form a cylinder. In the following figure, the cylinder is tangential to the equator. The closer the zone of tangency the less the distortion.



A cylindrical projection (1)

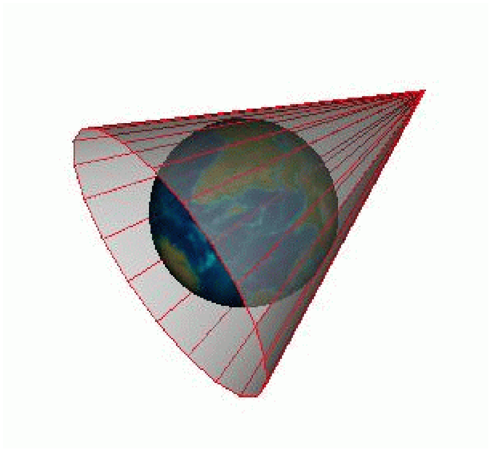
The position of the cylinder can be changed. For example, in a transverse cylindrical projection, the cylinder is tangential to a meridian.

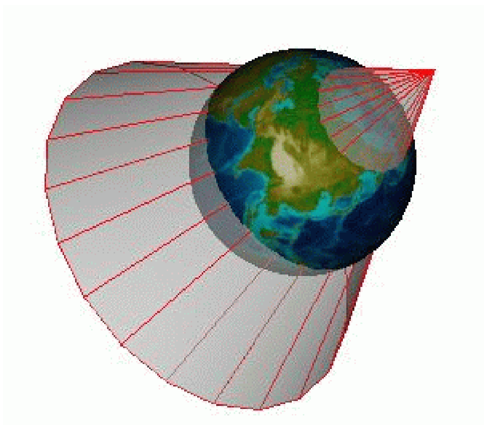


A cylindrical projection (2)

Conic projections

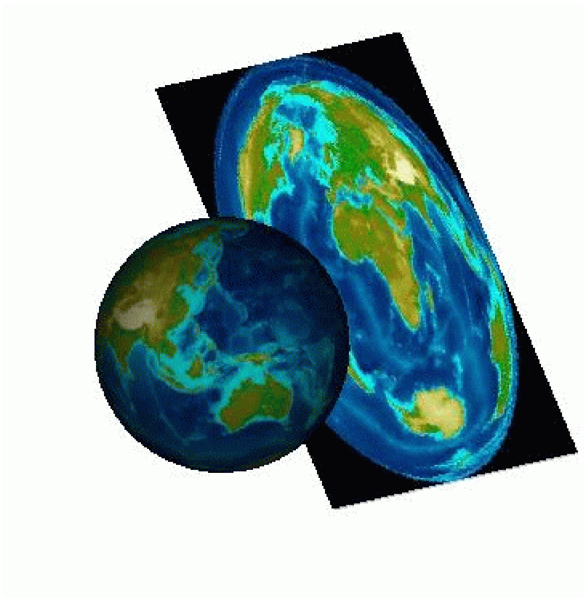
A conic projection transfers the image of the globe to a cone that forms either a secant or a tangent with the surface of the Earth.




Examples of conic projections

Azimuthal projections

With azimuthal projections, also called planar projections, the spherical globe is projected onto a flat surface.


An azimuthal projection

Equal area or conformal projections

All map projections show some kind of distortion in the areas that are far from the projection center. Depending on the kind of projection used, the distortion may be of angle, area, shape, size, distance, or scale. In this respect, projections fall into two main categories, Equal Area and Conformal.

- ♦ Equal area projections maintain a true ratio between the various areas represented on the map.
- ♦ Conformal projections preserve angles, and locally also preserve shapes.

Other projections have properties that are worth noting, such as maintaining the distances measured from the center of the projection (azimuthal equidistant projection). Others offer a good compromise between angular distortion and distortion of the area.

Projections should therefore be configured and selected according to the areas to be represented (for example, it is impossible to represent the polar regions with the Mercator projection) and the domains they apply to (navigational or air-route applications, small-scale or large-scale maps, and so on). Navigational applications, for example, generally use conformal projections.

For more information on map projections, refer to these books:

- ♦ *Map Projections - A Working Manual* (Snyder, 1987)
- ♦ *An Album of Map Projection*s (Snyder and Voxland, 1989)

Predefined projections

The following table provides a list of the predefined projections that are available.

Projection name	**Classification**	**Conformal**	**Equal Area**
Albers Equal Area	Conic	No	Yes
Azimuthal Equidistant Projection	Azimuthal	No	No
Cassini	Cylindrical	No	No
Cylindrical Equal Area also known as: ♦ Lambert Cylindrical Equal Area ♦ Behrmann ♦ Gall Orthographic ♦ Peters	Cylindrical	No	Yes
Eckert IV	Pseudo Cylindrical	No	Yes
Eckert VI	Pseudo Cylindrical	No	Yes
Equidistant Cylindrical also known as: ♦ EquiRectangular ♦ Plate Carré	Cylindrical	No	No
French Lambert	Conic	Yes	No
Geographic	N/A	-	-
Gnomonic	Azimuthal	No	No
Lambert Azimuthal Equal Area also known as: ♦ Lorgna ♦ Zenithal Equal Area ♦ Zenithal Equivalent	Azimuthal	No	Yes
Lambert Conformal Conic	Conic	Yes	No
Lambert Equal Area Conic	Conic	No	Yes
Mercator also known as: ♦ Wright	Cylindrical	Yes	No
Miller Cylindrical	Cylindrical	No	No

Projection name	Classification	Conformal	Equal Area
Mollweide also known as: ♦ Homolographic ♦ Babinet ♦ Elliptical	Pseudo Cylindrical	No	Yes
Oblique Mercator also known as: ♦ Hotine Oblique Mercator	Cylindrical	Yes	No
Orthographic	Azimuthal	No	No
Polyconic	Polyconic	No	No
Robinson	Pseudo Cylindrical	No	No
Sinusoidal also known as: ♦ Sanson-Flamsteed ♦ Mercator Equal-Area	Pseudo Cylindrical	No	Yes
Stereographic	Azimuthal	Yes	No
Transverse Mercator also known as: ♦ Gauss Conformal ♦ Gauss-Krüger ♦ Transverse Cylindrical Orthomorphic	Cylindrical	Yes	No
Universal Polar Stereographic	Azimuthal	Yes	No
Universal Transverse Mercator	Cylindrical	Yes	No
Wagner IV	Pseudo Cylindrical	No	Yes

Projection methods and parameters

Forward and inverse methods

Projections are implemented using the `forward` and `inverse` methods:

- ♦ The `forward(ilog.views.maps.IlvCoordinate)` method converts a geographic point, defined by a longitude and a latitude, to its Cartesian coordinates.
- ♦ The `inverse(ilog.views.maps.IlvCoordinate)` method converts Cartesian coordinates to a latitude and a longitude.

These methods can throw exceptions of two different types that both inherit from the class `IlvProjectionException`:

- ♦ The `IlvUnsupportedProjectionFeature` exception is thrown when a feature that is not implemented is called. It originates from the following actions:
  - When trying to perform a forward projection on a nonspherical ellipsoid when the projection does not support nonspherical ellipsoids (`IlsEquidistantCylindricalProjection` for example).
  - When trying to inverse a projection that cannot be reversed.
- ♦ The `IlvToleranceConditionException` exception is thrown when an error occurs during computation.

To know whether these features are implemented in the projection you are using, use the methods `isEllipsoidEnabled()` and `isInverseEnabled()`.

Projection parameters

You can set the following parameters for a projection:

- ♦ The ellipsoid that specifies the figure of the Earth.

  For more information on ellipsoids, refer to section *Ellipsoids*.

  Each projection is associated with an ellipsoid. By default, most of the projections use the ellipsoid `SPHERE`. Only some specific projections, such as the Universal Transverse Mercator or the Universal Polar Stereographic, use a nonspherical ellipsoid by default.

  You will obtain more accurate projections using an appropriate ellipsoid, especially with large scale maps. Note, however, that computations are more complex and slower than when using a sphere.

  To specify the ellipsoid you want to use for a projection, use the method `setEllipsoid (ilog.views.maps.projection.IlvEllipsoid)`.

  ```
  IlvProjection projection = new IlvMercatorProjection();
  projection.setEllipsoid(IlvEllipsoid.WGS84);
  ```

You can either use a static member of the class `IlvEllipsoid`, which defines a number of commonly used ellipsoids, or create your own ellipsoid as explained in the section *Defining new ellipsoids*. You can also use one of the predefined ellipsoids listed in the section *Predefined ellipsoids*.

♦ The unit converter that specifies the measurement unit in which Cartesian coordinates should be expressed.

♦ The central meridian and the central parallel of the projection.

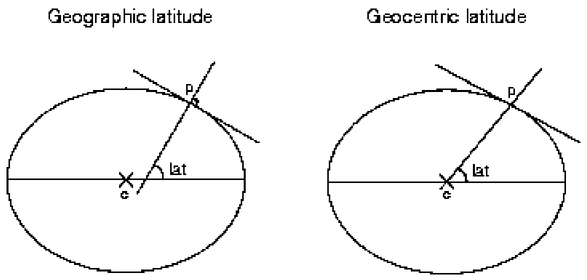
  These parameters can be set with the `setLLCenter(double, double)` method. Projections produce less distortion near their center.

♦ The offset applied to the Cartesian coordinates, also called false easting and false northing. These parameters can be set with the method `setXYOffset(ilog.views.maps.IlvCoordinate)`. The offset can be used in conjunction with the unit converters to control the range of projected coordinates for a region. For example, the range of the region may be set so that the region fits into a square of size 200 x 200. In JViews Maps applications, the range of the data is not an issue, since a transformer can be automatically applied to fit all the graphics contained in an `IlvManager` into a window. Therefore, false easting and false northing are mainly used to adapt a projection to geographic data that has already been projected using a Cartesian offset.

You can also:

♦ Specify whether the coordinates are geodetic (the default value) or geocentric using the `setGeocentric(boolean)` method.

  The geocentric latitude of a point is defined by the angle formed by a line joining the point to the center of the Earth and the equatorial plane, whereas the geodetic (or geographic) latitude of a point is defined by the angle formed by the vertical line passing through this point and the equatorial plane. The two values differ since the Earth is not exactly a sphere but rather an ellipsoid. Both latitudes are related through the relation `tan phiG = (1 - e ^ 2) tan phi` where `e` is the eccentricity of the ellipsoid used to model the shape of the Earth.



If an application handles geocentric data, this parameter must be set. Most of the available cartographic data available is expressed with geographic latitudes.

♦ Specify whether the projection uses longitude reduction, that is, forces longitude to be in the range `[-PI;PI]` or accepts any longitude, using the method `setUsingLongitudeReduction(boolean)`.

The above parameters are common to all the projections. They can be set with the API of the class `IlvProjection`, which is the base class of all the projections in the package. Some projections have additional specific parameters. For example, secant latitudes can be specified for a conic projection, or the latitude of the true scale can be specified for most cylindrical projections. For more information, refer to the documentation of the API for each projection.

Projection utilities

The class `IlvProjectionUtil` provides conversion utilities to convert radians to degrees and degrees to radians.

Creating a new projection

The procedure is based on an example that implements a simplified version of the Mercator projection. The complete code for this example can be found in the following file:

Mercator example.

Defining a new projection class

1. To define a new class, you must first import the projection library located in the package `ilog.views.maps.projection`.

   The complete source code of the examples presented in the next sections can be found in the following file:**Mercator example**.

2. Your projection class must extend the class `IlvProjection`, which is the base class for all the projections in the package.

```
import ilog.views.maps.projection.*;
import ilog.views.maps.*;

class MercatorProjection
extends IlvProjection
```

Writing the constructor

♦ You must call the constructor of the superclass `IlvProjection`.

```
MercatorProjection()
{
   super(true, true, IlvProjection.CONFORMAL);
}
```

This constructor takes the following three arguments:

♦ The first argument is a `boolean` value specifying whether the projection supports nonspherical ellipsoids. In our example, this argument is set to `true` since the projection supports the equations for these ellipsoids.

♦ The second argument is a `boolean` value indicating whether the projection supports an inverse method. In our example, this argument is set to `true` since the projection supports an inverse method.

♦ The third argument is an `int` value that indicates the geometric properties of the projection. In our example, this argument is `IlvProjection.CONFORMAL` since the Mercator projection is conformal.

Writing the Forward Projection

Before writing the `forward()` method for the Mercator projection, you must be familiar with the `IlvProjection.forward()` method.

The `IlvProjection.forward()` public method is called by the user to project data. This method prepares data for projection computation and scales it appropriately. It then redirects the calls to either one of the `eforward` or `sForward` protected methods which are defined in the projection subclass (the Mercator class in our example). In most cases, the `forward` method should not be overridden.

The `forward(ilog.views.maps.IlvCoordinate)` method does the following:

1. It adjusts the latitude, if the coordinates are geocentric.
2. It adjusts the longitude to the central meridian of the projection.
3. It adjusts the longitude to the range `[-PI;PI]`, if longitude reduction is used (the default value).
4. It calls either the method `sForward(ilog.views.maps.IlvCoordinate)` or `eForward (ilog.views.maps.IlvCoordinate)` depending on whether the Earth is represented as a sphere or as an ellipsoid.
5. Adjusts the projected data to the dimensions of the ellipsoid and to the Cartesian offset, and converts it to the selected measurement unit.

Projecting data from a sphere

The `sforward()` protected method implements the projection of a sphere.

Since the appropriate scaling is actually carried out by the method `IlvProjection.forward ()`, the `sForward()` method always assumes that the radius of the sphere is 1.

In the example, the Mercator projection is the projection of a sphere onto a cylinder that is tangential to the equator. The `x` coordinate is equal to the longitude because it is assumed that the radius of the sphere is 1, and longitude is expressed in radians. In this case, you do not need to change the `x` value of `ll`.

Because the Mercator projection cannot show regions near the poles, the exception `IlvToleranceConditionException` is thrown if the latitude is too close to `PI/2`.

♦ Apply the equation to compute the `y`-coordinate of the projected data.

```
protected void sForward(IlvCoordinate ll)
  throws IlvToleranceConditionException
{
  if (Math.abs(Math.abs(ll.y) - Math.PI / 2D) <= 1e-10D)
     throw new IlvToleranceConditionException();

  ll.y = Math.log(Math.tan(Math.PI / 4D + .5D * ll.y));
}
```

Projecting data from an ellipsoid

The `eforward()` protected method is called by the `IlvProjection.forward()` method if data is projected from a nonspherical ellipsoid.

♦ It is not necessary for you to implement the `eForward()` method for your projection. If you are projecting data from a nonspherical ellipsoid and if the projection you are using does not support this kind of ellipsoid, the `forward()` method will throw the exception `IlvUnsupportedProjectionFeature`. In this case, you can use any spherical ellipsoid or create an equivalent sphere using the appropriate conversion methods of the class IlvEllipsoid.

The eForward() method is slightly more complex than the sForward() method although their formulas are equivalent if getEllipsoid().getE() returns 0.

```
protected void eForward(IlvCoordinate ll)
   throws IlvToleranceConditionException
{
  if (Math.abs(Math.abs(ll.y) - Math.PI / 2D) <= 1e-10D)
     throw new IlvToleranceConditionException();

  double e = Math.sqrt(getEllipsoid().getES());

  double sinphi = e * Math.sin(ll.y);
  ll.y = Math.tan (.5D * (Math.PI/2D - ll.y)) /
         Math.pow((1D - sinphi) / (1D + sinphi),
                   .5D * e);
  ll.y = -Math.log(ll.y);
}
```

Writing the inverse projection

Before writing the inverse() method for the Mercator projection, you should be familiar with the inverse(ilog.views.maps.IlvCoordinate) method.

The IlvProjection.inverse() method prepares the data for inversion and processes it for the appropriate offset. In most cases, you should not have to override the IlvProjection.inverse() method.

This method does the following:

1. Suppresses the offset produced by the Cartesian coordinates and converts these coordinates to meters.
2. Reverts the coordinates to their geographic values and applies them to a standard ellipsoid with a semi-major axis of value 1.
3. Calls the method sInverse() or eInverse() depending on whether the ellipsoid is a sphere or not.
4. Adds the value of the central meridian to the longitude and adjusts the longitude to the range [-PI;PI] if longitude reduction is used (the default value).
5. Converts the latitude if the coordinates are geocentric.

Inverse projection onto a sphere

The inverse projection onto a sphere is performed via the sInverse(ilog.views.maps.IlvCoordinate) method.

1. It is not necessary for you to implement the sInverse() method. If you call the IlvProjection.inverse() method for a projection that does not support the inverse() method, the exception IlvUnsupportedProjectionFeature will be thrown.

2. The `sInverse()` method can throw the exception `IlvToleranceConditionException` like all the forward methods. But since the inverse equation of the Mercator projection is defined for all the possible values, this method does not throw any exceptions.

3. As with the `sForward()` method, the projection does not modify the `x` value, therefore, the inverse equation is applied only to the `y` value.

```
protected void sInverse(IlvCoordinate xy)
{
  xy.y = Math.PI/2D - 2D * Math.atan(Math.exp(-xy.y));
}
```

Inverse projection onto an ellipsoid

The inverse projection onto an ellipsoid is performed via the `eInverse(ilog.views.maps.IlvCoordinate)` method.

This method assumes that the value of the semi-major axis of the ellipsoid is 1.

♦ In the particular case of the Mercator projection, the implementation of this method is more complex for an ellipsoid than for a sphere. It requires iterations and might fail, since there is no simple analytical inverse equation of the Mercator projection from a nonspherical ellipsoid.

```
protected void eInverse(IlvCoordinate xy)
  throws IlvToleranceConditionException
{
  double ts = Math.exp(- xy.y);
  double e = Math.sqrt(getEllipsoid().getES());
  double eccnth = .5D * e;

  double Phi = Math.PI/2D - 2D * Math.atan(ts);
  int i = 15;
  double dphi;
  do {
    double con = e * Math.sin (Phi);
    dphi = Math.PI/2D - 2D * Math.atan(ts * Math.pow((1D - con) /
           (1D + con), eccnth)) - Phi;
    Phi += dphi;
  } while ((Math.abs(dphi) > 1e-10D) && (--i != 0));
  if (i <= 0)
    throw new IlvToleranceConditionException("non-convergent inverse
phi2");
  xy.y = Phi;
}
```

Spatial reference system

Spatial Reference Systems (SRS) are a way to link coordinates to a reference, so that objects whose coordinates are expressed in different systems can be displayed in the same manager.

JViews Maps features, or generally speaking graphic objects on maps, are representation of real objects. These map features have to be linked to real-life objects, and this is performed by attaching the attributes to the map features. These attributes can be a location, a time, or any descriptive quality or quantity. For example, you can describe the position of a restaurant either using its coordinates (The restaurant coordinates are 2D28’30’’E, 48D59’05’’N), or a description (The restaurant is at the crossing of X and Y streets, on the same walkway as the cafe). This link between objects and their real life counterpart is called a *reference system*.

Coordinate system base class

The JViews Maps supports Spatial Reference Systems where the reference attributes are coordinates. The abstract class `IlvCoordinateSystem` serves as base class for all the coordinate systems.

An `IlvCoordinateSystem` is defined by:

♦ A optional name.

♦ An array of `IlvUnit` defining the units to be used on each axis.

♦ An array of `String` defining the name of each axis.

♦ The dimension of the coordinates in use in the system are defined by the number of axes.

There are three major classes of coordinate systems useful for mapping software:

♦ Geocentric Coordinate System

  Represents coordinates in a three-axes Cartesian system, whose origin is the center of Earth. This coordinate system is mainly used in datum conversion.

♦ Geographic Coordinate System

  Represents the coordinates specified by angles on an ellipsoid (longitude and latitude). An optional height above the ellipsoid can be used here. For example, this is the standard longitude and latitude given by GPS.

♦ Projected Coordinate System

  Represents the coordinates of the Earth on a 2-D surface. There are as many projected coordinate systems as the number of existing projections. Projecting coordinates on a 2-D surface is mandatory to display data on a map. As the projection process introduces some errors, not all projections are well suited to represent an area on Earth. For more information, see *Map projections*.

Geocentric Coordinate System

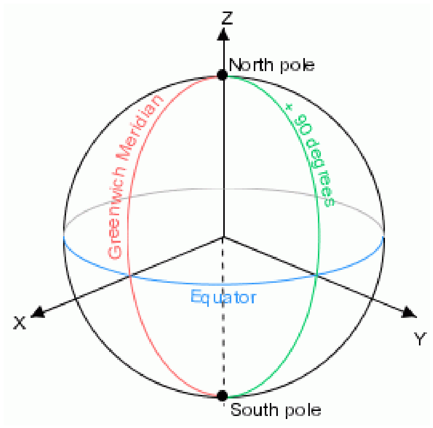
The `IlvGeocentricCoordinateSystem` class defines a geocentric coordinate system, that is, a three-dimensional Cartesian system. The origin point of this Cartesian system is the center of the Earth.

The axes are perpendicular and defined as follows:

♦ the x-axis lies in the plane containing the equator, and has positive values towards the Greenwich meridian

♦ the y-axis lies also in the plane containing the equator, and is positive towards the longitude 90 degrees east of Greenwich

♦ the z-axis corresponds to the polar axis, and is positive northwards



The geocentric coordinate system

Coordinates in a geocentric coordinate system are expressed in linear units along the axis.

The geocentric coordinate system is mostly used as the base reference system from which geographic and projected coordinates are derived. For example, geodetic datums (horizontal datum) are defined by the shift, rotate and scaling parameters to convert a geocentric coordinate system to a reference geocentric coordinate system (in most cases, the WGS84 datum).

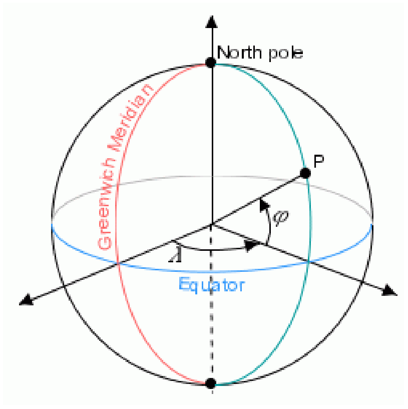
Geographic Coordinate System

The `IlvGeographicCoordinateSystem` class defines an ellipsoidal coordinate system where coordinates are specified as latitude and longitude on an ellipsoid, with an optional third coordinate which represents the ellipsoidal height (altitude above the ellipsoid).

♦ Longitude is specified in angular units from the prime meridian of the coordinate system. By convention, coordinates less than 180 degrees east of the prime meridian are positive, coordinates more than -180 degrees west are negative.

♦ Latitude is specified in angular units from the equator. By convention, northward latitudes are positive and southward coordinates are negative.

- The convention for representing the poles are longitude set to 0 and latitude set to 90 degrees for the north pole or -90 degrees for the south pole.



The geographic coordinate system

Projected Coordinate System

The geographic and geocentric coordinate systems are not well suited to display maps, since these coordinate systems define map features in a three dimensional world. Before displaying objects, the dimension of coordinates must be reduced to 2. This is performed using map projections (see *Map projections*).

A projected coordinate system includes all the parameters that allow you to describe a coordinate system in which each planar coordinate is computed from geographic coordinates using a mathematical function. These include:

- The reference geographic coordinate system (the surface modeling the Earth).
- The mathematical transformation itself (the projection).

Creating a map application using the API

Describes how to create a map using the API.

In this section

Using threads in data sources
Explains how to start a data source in a separate thread so that you can check when it has started.

Using the IlvThreadMonitor
Explains how to monitor tasks running in separate threads.

Generic code sample for creating a map
Explains how to create a basic map using a generic code sample.

Using readers
Describes the predefined readers, the map loader for supported (predefined) formats, and how to write a new reader.

Map GUI interactors
Contains information about the GUI map interactors on the Map Builder toolbar.

The See-Through interactor
Describes the see-through interactor.

Using the GUI beans
Describes the JavaBeans™ for GUIs with maps.

Handling map features
Describes map features and how they are handled.

Using load-on-demand
Describes load-on-demand and how to use it.

Manipulating renderers
Describes the attributes of graphic objects that will be displayed and the use of renderers to transform map features to graphic objects.

Handling spatial reference systems
Describes various conversions.

Pregenerating tiled images for a thin client
Describes a class for thin clients.

Overview

The JViews Maps API is a fully documented class library that you can customize and extend to meet your application requirements. For more detailed information on how the JViews Maps library is structured, see *Introducing the main classes*.

Creating data source objects

Describes how to create data source objects for vector and raster data sources.

In this section

Data source

A data source is an object that connects a feature iterator, a renderer, and a map layer.

A data source uses the feature iterator to get global data information, such as the coordinate system and bounding box, and the list of map features to parse. Then it uses the renderer to create graphic objects. Finally, it uses the layer to manage the order and style of those objects.

In class terms, a data source is a class that connects an `IlvMapReusableFeatureIterator`, an `IlvFeatureRenderer` and an `IlvMapLayer`. It uses the `iterator` object to get global data information, such as the coordinate system and bounding box, and the list of `IlvMapFeature` objects to parse. Then it uses the `renderer` object to create graphic objects. Finally it uses the `layer` object to manage the order and style of the graphic objects.

Vector data sources

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

ESRI shapefile

When you create a Shapefile format data source the shapefiles can be loaded with or without the tiling mechanism. When using tiling, the data is loaded in a background thread.

The complete source code for an ESRI shapefile demonstration can be found at ***<installdir>*** **`/jviews-maps86/shape/mapbuilder/index.html`**

Loading a shapefile using tiling

To load a shapefile using tiling, use the following code:

```
IlvTiledShapeDataSource tiledSource = new
IlvTiledShapeDataSource(shpFileName,true);
```

If the shapefile does not have a tiling index file, you can create the index file as follows:

```
IlvShapeFileTiler tiler = new IlvShapeFileTiler(shpFileName, shxFileName,
indexFileName, tileWidth, tileHeight);
while(tiler.getNextFeature() != null) {
  tiler.addInfo();
}
tiler.close();
```

The resulting index file, which is used in the reader, can be set with a call to:

```
tiledSource.setIdxFilename(idxFileName);
```

Loading a shapefile without using tiling

To load a shapefile without using tiling, use the following code:

```
IlvShapeDataSource shpDataSource = new IlvShapeDataSource(shpFileName, true);
```

MID/MIF

To create a MID/MIF file data source, use the following code:

```
IlvMapDataSource source = new IlvMIDMIFDataSource(fileName);
source.setManager(getView().getManager());
```

TIGER/Line

To create a TIGER/Line data source, use the following code:

```
IlvTigerDataSource source = new IlvTigerDataSource(fileName);
source.setManager(getView().getManager());
```

TIGER/Line® data contains many different features. You can select the features to import into your map by choosing the CFCC codes of the features you want:

```
source.setCFCCCodeList(CFCCCodes);
```

DXF (AutoCAD)

The DXF format (Drawing Interchange Format) is the interchange format for AutoCAD. This format supports vector graphics (polygons, arcs, lines, points...) and layers.

The specifications of the various releases of the DXF format can be found at the following URL: *http://usa.autodesk.com/adsk/servlet/item?siteID=123112&id=5129239*.

For details of the limitations with a DXF data source, refer to Drawing Exchange Format (DXF) in the section on managers in *The Essential JViews Framework*.

To create a DXF data source, use the following code:

```
IlvDXFDataSource source = new IlvDXFDataSource("C:/maps/DXF/maps.dxf");
source.setManager(getView().getManager());
```

You must then setup the transformation you want to use to read DXF.

If you want to transform the DXF extent into the latitude/longitude range provided, call:

```
source.setDestinationBounds(lonMinRad,latMinRad,lonMaxRad,latMaxRad);
```

If you want to use a more complex transformation (such as reprojection), call:

```
IlvMathTransform mathTransform =…/create mathematical transformation
   source.setInternalTransformation(mathTransform);
```

An example of how to create a complex transformation is given in the Map Builder demonstration, through use of the `DXFControlModel` class.

KML or KMZ

KML/ KMZ is an XML-based numerical map format containing both vectorial and raster information. KML is published by Google™ , for use in their Google Earth© Product.

You can find more information at *http://earth.google.com/kml/kml_intro.html*.

JViews Maps supports import of the KML elements shown in *KML element support*:

KML element support

Element	Support
Placemarks	Geometric Shapes, Location.
Geometry	Points, Lines, Polygons.
Image Overlays	Ground Overlays.
Styles	No Support.
Grouping Mechanisms	Documents, Folders, Geometry Collections.
Network Links	Locations.

KML files can refer to other files, either locally or through a network link URL. KMZ is a zipped file containing a single `doc.xml` entry and a set of auxiliary files, such as images or icons.

To create a KML data source, use the following code:

```
IlvKMLDataSource source = new IlvKMLDataSource("C:/maps/KML/places.kmz");
source.setManager(getView().getManager());
```

SVG

SVG (Scalable Vector Graphic) is a language for describing two-dimensional graphics and graphical applications in XML. The specifications of the SVG format can be found at *http://www.w3.org/Graphics/SVG*.

The JViews Maps SVG reader is based on features in IBM® ILOG® JViews Framework. These features are described in the section on Scalable Vector Graphics in *The Advanced JViews Framework*.

The JViews Maps SVG reader ignores the following SVG elements:

♦ Images defined through the image element.

♦ Text defined through the text-path element.

All the other elements, including text elements, are transformed into `IlvMapGeneralPath` instances.

The graphic styles in the resulting map are rendered to look as close as possible to the original version. However, the styles created are limited to the data available for styling such an object.

To create an SVG data source, use the following code:

```
IlvSVGDataSource source = new IlvSVGDataSource("C:/maps/SVG/maps.svg");
source.setManager(getView().getManager());
```

You then need to setup the transformation you want to use to read SVG. To transform the SVG extent into the latitude/longitude range provided, call:

```
source.setDestinationBounds(lonMinRad,latMinRad,lonMaxRad,latMaxRad);
```

To use a more complex transformations such as reprojection, call:

```
IlvMathTransform mathTransform = //create mathematical transformation
source.setInternalTransformation(mathTransform);
```

An example of how to create a complex transformation using of the `ControlModel` example class can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Raster data sources

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

All raster (image) data sources have a common API. To create your data source, you need to create an `IlvRasterAbstractReader` for the type of data to be read, and then use this reader to build a tiled image data source.

Raster reader classes

The raster reader classes for the different image formats are as follows.

```
GEOTIFF
IlvRasterGeoTiffReader
DTED
IlvRasterDTEDReader
GTOPO30
IlvGTopo30Reader
Non-geo-referenced images(*)
IlvRasterBasicImageReader
Images from OpenGIS compliant Web Map Servers
IlvWMSReader
```

(*) Non georeferenced images must be geo-referenced, see *Georeferencing a nongeoreferenced image*.

Creating a raster reader

To create a raster reader and add all image files to be read you can, for example, use the following code:

Creating a tiled data source

To create a data source for the reader:

```
IlvRasterDTEDReader reader = new IlvRasterDTEDReader();
for(int i=0;i<fileName.length;i++) {
  reader.addMap(fileName[i]);
}
IlvMapDataSource DTEDDataSource =
IlvRasterDataSourceFactory.buildTiledImageDataSource(manager,reader,true,true,n
ull);
DTEDDataSource.setName("name in data source panel");
```

The `IlvTiledRasterDataSource` returned by the `IlvRasterDataSourceFactory` executes image reading in a background thread.

Georeferencing a nongeoreferenced image

To georeference a nongeoreferenced image, you can set the longitude and latitude image bounds:

```
reader.setImageBounds(0,-Math.PI,Math.PI/2,Math.PI,-Math.PI/2);
```

Alternatively, you can compute a more complex mathematical transformation and set it on the reader:

```
reader.setInternalTransformation(trans);
```

Note: The non georeferenced image reader does not support multiple calls to addMap.

Images from OpenGIS-compliant Web Map Servers

This section gives information about the OpenGIS® Web Map Server (WMS standard). This International Standard specifies the behavior of a service that produces spatially referenced maps dynamically from geographic information. It specifies operations to retrieve a description of the maps offered by a server and to query a server about features displayed on a map. The standard is not applicable to the retrieval of actual feature data or coverage data values, but is applicable to pictorial renderings of maps in a graphical format. These capabilities are provided by a Web Feature and Web Coverage Service.

In a basic WMS, only a limited number of predefined styles can be applied to features. The mechanism that enables users to define their own styles is defined in the OGC Styled Layer Descriptor Implementation Specification. An SLD-enabled WMS retrieves feature data from a Web Feature Service and applies explicit styling information provided by the user in order to render a map.

The ISO/TC 211 also defines a standard for Web Map Servers, see *ISO 19128*.

Importing a WMS image

JViews Maps supports the import of images from a WMS server.

To create a WMS data source, use the following code:

```
URL url = new URL("http://geo.compusult.net/scripts/mapman.dll?
   Name=weather&REQUEST=GetCapabilities");
IlvWMSReader reader = new IlvWMSReader(url);
IlvWMSDataSource source = new IlvWMSDataSource(reader);
source.setManager(manager);
```

Using data sources

Describes how to use data sources.

In this section

Overview

Once you have created your data source (see *Creating data source objects*), integrate it with the manager properties by:

- Inserting it into the data source tree.
- Inserting the associated map layer into the map layer tree.

You can then use layer styling methods to change the styles. Additionally, at some point, the creation of all graphic objects must be started (usually in a background thread).

Integrating the data source

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>* /jviews-maps86/samples/mapbuilder/ index.html**

To integrate the data source and map layer with the manager properties:

1. Insert the data source into the data source tree. You first need to retrieve the data source model from the property of the manager:

```
IlvMapDataSourceModel dataSourceModel =
IlvMapDataSourceProperty.GetMapDataSourceModel(manager);
dataSourceModel.insert(source);
```

2. Once all data sources have been inserted, you can start loading the data source model by starting all data sources of this model recursively:

```
dataSourceModel.start();
```

   Alternatively, you can start each data source when you want it by calling.

```
source.start();
```

3. You should also retrieve (and possibly setup) the map layer attached to the data source, for example:

```
IlvMapLayer layer = dataSource.getInsertionLayer();
layer.setName("name in layer tree panel");
```

   Other settings can be done at this stage, such as changing the layer style.

```
layer.getStyle().setAttribute(IlvPolylineStyle.FOREGROUND,Color.black);
layer.getStyle().setAttribute(IlvPolylineStyle.BACKGROUND,new
Color(1,1,1,0.25f));
```

4. Insert the layer into the map layer tree of the manager. You first need to retrieve the layer tree model from the property of the manager. Here you can arrange the layer structure by selecting the parent layer as appropriate.

```
IlvMapLayerTreeModel ltm =
IlvMapLayerTreeProperty.GetMapLayerTreeModel(manager);
ltm.addChild(parent, layer);
```

   You can reuse the same parent layer for different data sources, possibly retrieving it from the tree model:

```
IlvMapLayer parent = ltm.findChildLayer(null,"my parent");
if(parent==null) {
  parent = new IlvMapLayer();
  parent.setName("my parent");
  IlvMapCompositeStyle parentStyle= new IlvMapCompositeStyle();
  parent.setStyle(parentStyle);
  ltm.addChild(null, parent);
}
```

Layer styling considerations

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Managing the style parent

To provide style inheritance (accessible in the Layer Tree Panel), the parent layer style must be a composite style, and you need to manage the style parent on its own:

```
IlvMapStyle childStyle=layer.getStyle();
childStyle.setParent(parent.getStyle());
```

Using dynamic styles

When the application uses dynamic styles, you need to access the style controller property of the manager:

```
IlvMapStyleController themeController =
IlvMapStyleControllerProperty.GetMapStyleController(manager);
```

With the controller, you can decide on specific style settings for scale intervals. For example, to change the layer visibility:

```
themeController.addTheme(1/100000.0,source.getInsertionLayer(),"Visible");
themeController.getStyle(source.getInsertionLayer(),1/
100000.0).setVisibleInView(true);
themeController.getStyle(source.getInsertionLayer(),1/
100000.0).setVisibleInOverview(false);
```

If you use multiple dynamic styles, you have to take care of style inheritance in a slightly more complex way because more than one style is used:

```
IlvMapDynamicStyle []t=themeController.getThemes(layer);
for (int i = 0; i < t.length; i++) {
  t[i].getStyle().setParent(parent.getStyle());
}
```

Once this is done, you can apply the style you want to use for the current scale of the view:

```
themeController.updateTheme(view,layer);
```

Alternatively, you can setup the theme for all layers in one single call:

```
themeController.updateCurrentTheme();
```

Layer ordering

With multithreading data sources, it is often impossible to predict the order in which the underlying manager layers are created. The manager layer order determines which graphic objects are on top of the map or in the background. When data sources have been created and inserted, you can arrange the manager layer order to be coherent with the map layer tree organization, with a call to:

```
ltm.arrangeLayers();
```

Clearing map data

When creating data sources and map layers, many different manager properties are modified and refer to the data structures needed to manage the map at its different scales. Furthermore, many graphic objects and manager layers are created in the `IlvManager` object. You may later want to clear the data.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

To clear the map data entirely:

1. Stop all potential threaded image loaders by looking for `IlvThreadedTileLoader` which may still be running:

```
for (int il=0;il<manager.getLayersCount();il++) {
  IlvManagerLayer layer = manager.getManagerLayer(il);
  if (layer instanceof IlvTiledLayer) {
    IlvTileLoader loader = ((IlvTiledLayer) layer).getTileLoader();
    if (loader instanceof IlvThreadedTileLoader) {
      ((IlvThreadedTileLoader) loader).dispose();
    }
  }
}
```

2. Clear the map layer model by removing all manager layers and graphic objects:

```
IlvMapLayerTreeModel model =
IlvMapLayerTreeProperty.GetMapLayerTreeModel(manager);
model.clearAllObjects();
while (manager.getLayersCount() > 0) {
 manager.removeLayer(0, false);
}
manager.removeNamedProperty(IlvMapLayerTreeProperty.NAME);
```

3. Clear the data source model:

```
manager.removeNamedProperty(IlvMapDataSourceProperty.NAME);
```

4. Clear the area of interest model:

```
manager.removeNamedProperty(IlvAreasOfInterestProperty.NAME);
```

5. Clear the style controller:

```
manager.removeNamedProperty(IlvMapStyleControllerProperty.NAME);
```

6. Remove all temporary raster files. JViews Maps relies on a memory mapped data file for quick access to raster pixel data. Temporary files are not removed when applications exit abnormally, and they remain in the temporary directory forever unless removed. To prevent this, the `IlvRasterTemporaryFileManager` class provides a method to remove all temporary files created by JViews Maps that are no longer in use:

   ```
   IlvRasterTemporaryFileManager.removeAllFiles();
   ```

Developing a new data source

Understanding the data source backup paradigm

The base class `IlvMapDataSource` includes a fallback mechanism to allow map reprojection and export even when associated source data (file, network path, database connection...) is not available at the time the map needs to be manipulated. For that mechanism to operate, a subclass of `IlvMapDataSource` must override the method `isSourceDataAvailable()` and perform appropriate checks on related data availability.

If this method returns `false`, the `IlvMapDataSource` tries to perform operations on the map (change of coordinate system, export) from the current map objects instead of reading them from the original source data. Note that this may lead to loss of precision or even data as this is the expected behavior when chaining-up several non-invertible projections.

Another way to force the use of a backup data source is by setting a flag using the method `setForceUsingBackupDataSource(boolean)`. This will result in much faster operations (such as reprojection) as the source data is not read back from its original format. However, as mentioned above, this leads to potential precision or data loss.

Should you need to access these backup data sources of a given data source directly, you can do so by calling `getBackupDataSources()`. It returns an array of data sources of the class `IlvGraphicLayerDataSource`.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**.

Renderer management

Unless you need specific rendering, you should use the base `IlvMapDataSource` class renderer management, which either finds the renderer associated with the `IlvMapReusableFeatureIterator` or creates a new `IlvDefaultFeatureRenderer`.

Layer management

By default, the base `IlvMapDataSource` class layer manager creates a map layer when needed, and connects it to the manager (by creating an `IlvManagerLayer`).

Usually, the only layer management method you have to rewrite is the `initInsertionLayer(ilog.views.maps.beans.IlvMapLayer)` method, when you need to create a tiled layer instead of a standard `IlvManagerLayer`:

```
protected void initInsertionLayer(IlvMapLayer layer) {
    layer.insert(new IlvTiledLayer(new IlvRect(), null,
IlvTileController.FREE));
}
```

If you want your data source to manage more than a single map layer you may have to write more complex layer management code, or create a subclass of the `IlvHierarchicalDataSource`.

Data tiling

It you have a data format or readers that support tiles, you may have to create the tiles and tile loaders in a specific `start` method. Here is an example used in the shapefile data source:

```
public void start() throws Exception
{
// construct a tiled shape tile loader
  IlvShapeFileTileLoader tileLoader = new IlvShapeFileTileLoader(shp, dbf,shx,

idx);
  tileLoader.setCoordinateSystem(getCoordinateSystem());
tileLoader.setFeatureRenderer(getFeatureRenderer());
// create a threaded tile loader to load the shape data on a background thread.

  IlvTiledLayer tiledLayer = (IlvTiledLayer)getInsertionLayer().
     getManagerLayer();
  IlvThreadedTileLoader threadedLoader = new IlvThreadedTileLoader(tileLoader,

     true);
tiledLayer.setTileLoader(threadedLoader);
...
// for each tile known by the tile loader
  for (int i = ...) {
    for (int j = ...) {
     // Compute projected tilebounds, ie the bounds of the tile in the manager

coordinates
      IlvRect r = IlvMapUtil.computeTransformedBounds(...);
      Point2D.Double ul = new Point2D.Double(r.getX(), r.getY());
      Point2D.Double lr = new Point2D.Double(r.getX() + r.getWidth(), r.getY
()
+ r.getHeight());
tiledLayer.getTileController().addTile(new IlvMapFreeTile(ul, lr,
   tiledLayer.getTileController(), i, j));
    }
  }
}
```

Feature management

The `IlvMapReusableFeatureIterator` is a subinterface of `IlvMapFeatureIterator`. It adds to the base interface the capability to restart the iteration more than once. This is necessary when, for example, projection parameters have changed and the data source needs to render all the graphic objects again. This is also used for load-on-demand or save/reload mechanisms.

You can transform a feature iterator (such as for readers written for a previous version of JViews Maps) into a reusable feature iterator by using the `IlvMapDelegateFeatureIterator` abstract class. For example, the code below uses the feature iterator returned by an `IlvMapLoader`:

```
String fileName="some file name";
public IlvMapReusableFeatureIterator getFeatureIterator() {
   return new IlvMapDelegateFeatureIterator() {
      public void restart() {
         IlvMapLoader loader = new IlvMapLoader(null);
            try {
               setDelegate(loader.makeFeatureIterator(fileName));
            } catch (IOException e) {
               e.printStackTrace();
            }
        }
    };
}
```

Printing

IBM® ILOG® JViews Maps provides specialized printing classes derived from the JViews Framework printing classes. These classes are use to print a map with or without a legend. The printing of the legend can be configured in the map configuration tab of the setup dialog.

To print a map, do the following:

```
// construct a new IlvMapPrintingController for the given view.
IlvMapPrintingController controller = new IlvMapPrintingController(view);
// configure the document
IlvPrintableDocument document = controller.getDocument();
document.setName(<Name of the Document>);
document.setAuthor(controller.getPrinterJob().getUserName());
document.setPageFormat(controller.getPrinterJob().defaultPage());
```

At this point, you can do one of the following:

- ♦ Access the print preview screen

  ```
  // print preview
  controller.printPreview(frame);
  ```

- ♦ Access the setup dialog

  ```
  // print setup
  controller.setupDialog(frame, true, true);
  ```

- ♦ Access the printing window

  ```
  // print
  controller.print(true);
  ```

For more information, see the section on The Generic Printing Framework in The Advanced JViews Framework.

Overview of multithreading

Heavy tasks are performed in threads to prevent the application GUI from freezing. Some of the threads are hidden, others can be controlled.

Use of threads in the Map Builder

All Map Builder import tasks are performed in separate threads to ensure GUI responsiveness. This means that you can open a batch of files of different formats without having to wait for the current file to finish being imported. This is basically achieved by calling the start methods of the associated data sources in a dedicated thread.

Use of threads in map labelling

The map labelling mechanism in JViews Maps is implemented in a background thread so that the user can continuously zoom, pan, and scroll the view without being slowed down by the sometimes CPU-intensive layout algorithms.

As soon as user interaction with the view stops, a timer starts. If no further change occurs in the view within a certain length of time (a few hundreds of milliseconds typically), the label layout process starts in the dedicated background thread. If it does not finish before the view changes again, the current task is canceled so that the next one can start as soon as possible.

Using threads in tile loaders

When tiles are displayed on a view, the tile loading is performed for the `IlvTiledLayer`s by an `IlvTileLoader`, mainly through the call of the `load(ilog.views.tiling.IlvTile)` method. Up until JViews Maps 8.1, this was performed in the Swing thread, which could cause GUI freezes because the code could be CPU intensive.

You can execute the tile loading code in separate threads by means of the `IlvThreadedTileLoader` class. This class implements the `IlvTileLoader` interface, and acts as a wrapper for a delegate `IlvTileLoader` executing the tile loading code.

Internally, an `IlvThreadedTileLoader` object holds a queue in which `load` and `release` calls (from the `IlvTileController` on the `IlvThreadedTileLoader`) are stacked, and processed as soon as possible by a dedicated thread. This way, all tile loading is performed asynchronously and the application remains responsive even if tens of megabytes of data are loaded for display on the view.

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

The following code sample shows how to use an `IlvThreadedTileLoader` that encapsulates another tile loader:

```
IlvTiledLayer tiledLayer;
// this is the tile loader that performs the real loading, such as
IlvShapeFileTileLoader
IlvTileLoader tileLoader;

// Create a threaded tile loader that encapsulates the previous one
IlvThreadedTileLoader threadedTileLoader =
                         new IlvThreadedTileLoader(tileLoader,true);

// Configure this threaded tile loader :

// set the minimum java thread priority for loading tiles
threadedTileLoader.setThreadPriority(Thread.MIN_PRIORITY);

// set to repaint the view after each tile is loaded
threadedTileLoader.setRepaintPolicy(IlvThreadedTileLoader.REPAINT_AFTER_EACH_TI
LE);

// Set this threaded tile loader on the IlvTiledLayer
tiledLayer.setTileLoader(threadedTileLoader);
```

Using threads in data sources

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

The `IlvTiledRasterDataSource`, performs its start method in a separate thread if called from the Swing event dispatch thread. This is useful to know if you want to wait for the start method to finish. To do this, call the `start()` method of the `IlvTiledRasterDataSource` from a different thread to the event thread to be sure it has completed as it returns.

For instance, to ensure that an `IlvRasterDTEDDataSource` has completed its start method, you could use the following code.

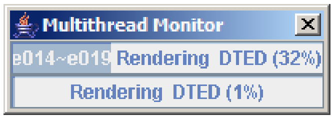
```
IlvMapDataSource DTEDDataSource;
...
// create reading thread
Thread loader = new Thread() {
    public void run() {
        DTEDDataSource.start();
    }
};
// start reading thread
loader.start();
// wait for the thread to complete
loader.join();
```

Using the IlvThreadMonitor

`IlvThreadedActivityMonitor` is a class that monitors the progress of several tasks running in separate threads.

These tasks first register with this monitor through the `registerThreadedActivity(java.lang.Object)` method, and then notify the monitor of any progress they make by calling the `updateActivityProgress(java.lang.Object, int, java.lang.String)` method. Eventually, a task can be removed from the list by calling `unregisterThreadedActivity(java.lang.Object)` method. A task that reports a progress reading of one hundred percent is automatically unregistered from the monitor.

In its turn, the monitor notifies any registered listeners of the progress being made by the monitored tasks. *Multithread Monitor* shows the `IlvThreadedActivityMonitorPanel` class, which is a Swing component implementing the `IlvThreadedActivityMonitor.ActivityListener` interface. It displays progress bars for monitored activities.



Multithread Monitor

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/86/samples/mapbuilder/index.html`**

If you want to display the progress of your own threaded tasks on this `IlvThreadedActivityMonitorPanel`, here are the basic actions you should take:

```
IlvManagerView view; // the main view of the application

// Get the manager of this view (provided that it was set somewhere before)
IlvManager manager = view.getManager();

// Get the activity monitor associated with this manager
IlvThreadedActivityMonitor monitor =
IlvThreadedActivityMonitorProperty.GetThreadedActivityMonitor(manager);
if (monitor != null) {
 // Notify the monitor of the progress of this activity
monitor.updateActivityProgress(this,100,null);
};
```

For further information about this bean, see *Multithread Monitor*.

Generic code sample for creating a map

The complete source code for this example can be found in the following file:

<installdir> **/jviews-maps86/codefragments/readers/src/GenericReader.java**

1. Instantiate the reader. In this example, you are using the `IlvShapeReader`.

   ```
   try
   {
       featureIterator = new IlvShapeFileReader(shapeFileName, dbfFileName)
   ;
   }
   catch (IOException e)
   {
      System.err.println("IOError while instantiating reader");
   }
   ```

2. Retrieve the feature renderer from the reader.dted.

   ```
   IlvFeatureRenderer renderer =
   featureIterator.getDefaultFeatureRenderer();
   ```

3. Create coordinate transformation.

   ```
   IlvCoordinateTransformation identity = IlvCoordinateTransformation.
   CreateTransformation(null, null);
   ```

4. Retrieve the first map feature.

   ```
   IlvMapFeature feature = null;
   try
   {
       feature = featureIterator.getNextFeature();
   }
   catch (IOException io)
   {
       System.err.println("IOExeption while getting next feature"
                          +io.getMessage());
   }
   // Loop on all the available map features.
   while (feature != null)
   {
     try {
   ```

5. Create the graphic object representing the map feature.

   ```
   IlvGraphic graphic = renderer.makeGraphic(feature, identity);
   ```

6. Add the graphic object to the manager.

```
manager.addObject(graphic, layerIndex, false);
```

7. Handle exceptions.

```
} catch (IlvMapRenderException e)
{
// Should not occur: renderer provided by the feature iterator.
System.out.println("Rendering Exception " + e.getMessage());
} catch (IlvCoordinateTransformationException cte) {
System.err.println("Coordinate transformation exeception " + cte.getMessage
());
}
}
```

Using readers

Describes the predefined readers, the map loader for supported (predefined) formats, and how to write a new reader.

In this section

Code examples for using readers

For more information on the predefined readers, see *Readers and writers*.

The Shapefile reader

The complete source code for this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/readers/src/ShapeReader.java`**.

```
/**
* Simple file reader for ESRI shape file format.
* Attributes are read from the dbf file.
*/
  public void loadFile(String shapeFileName, String dbfFileName)
  throws IOException,
         IlvMapRenderException,
         IlvCoordinateTransformationException {

    IlvShapeFileReader reader = null;
    // Instantiate a new reader.
    try {
      reader = new IlvShapeFileReader(shapeFileName, dbfFileName);
    } catch (IOException e) {
      e.printStackTrace();
    }
    // Retrieve the default feature renderer.
    IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();

     //  Coordinate Transformation from IlvGeographicCoordinateSystem.WGS84
    // to IlvGeographicCoordinateSystem.WGS84 (no transformation)
    IlvCoordinateSystem sourceCoordinateSystem =
                           IlvGeographicCoordinateSystem.WGS84;
    IlvCoordinateSystem targetCoordinateSystem =
                           IlvGeographicCoordinateSystem.WGS84;
    IlvCoordinateTransformation transform =
              IlvCoordinateTransformation.CreateTransformation
                (sourceCoordinateSystem, targetCoordinateSystem);
    IlvMapFeature mapFeature;
    // Loop on all map features.
    while ((mapFeature = reader.getNextFeature()) != null) {
      IlvGraphic graphic = renderer.makeGraphic(mapFeature, transform);
      if (mapFeature.getAttributes() != null)
        graphic.setNamedProperty(mapFeature.getAttributes().copy());
      // IlvGraphic has to be stored in an IlvManager.
      manager.addObject(graphic, false);
    }
  }

/**
* Create a Shape load-on-demand layer.
*/
```

```
  public void loadLOD(String shpFileName,
                      String dbfFileName,
                      String shxFileName,
                      String idxFileName)
  throws IOException {

    // Instantiate tile loader.
    IlvShapeFileTileLoader shpTileLoader = new IlvShapeFileTileLoader(
        shpFileName,
        dbfFileName,
        shxFileName,
        idxFileName);
    // instantiate tiled layer.
    IlvTiledLayer tiledLayer =
             new IlvTiledLayer(shpTileLoader.getTileOrigin());
    // Affect tile loader.
    tiledLayer.setTileLoader(shpTileLoader);
    // Add the layer to the IlvManager.
    manager.addLayer(tiledLayer, 0);
    // Fit to tile 0, 0.
    tiledLayer.fitTransformerToTile(view, 0, 0);
  }
```

The MID/MIF reader

The complete source code for this example can be found in the following file:

<installdir> **/jviews-maps86/codefragments/readers/src/MIDMIFReader.java**.

```

/**
 * Example of how to use the MIDMIF reader package.
 */
public class MIDMIFReader {
  IlvManager manager = new IlvManager();

  /**
    * Load a MIDMIF file by providing the MIF file name and the MID file name.

   */
  public void loadMIDMIF(String mif, String mid)
  throws IlvMapFormatException,
         IOException,
         IlvMapRenderException,
         IlvCoordinateTransformationException {

    // Create the MIDMIF reader.
    IlvMapFeatureIterator reader = new IlvMIDMIFReader(mif, mid);
    // Retrieve the default renderer.
    IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
    // Initialize the coordinate transformation.
    IlvCoordinateSystem managerCS =
        IlvCoordinateSystemProperty.GetCoordinateSystem(manager);
    // The MIDMIF file can contain coordinate system information.
    IlvCoordinateSystem fileCS = reader.getCoordinateSystem();
    if ((managerCS == null) && (fileCS != null))
      manager.setNamedProperty(new IlvCoordinateSystemProperty(fileCS));
    // Create the transformation accordingly.
    IlvCoordinateTransformation tr =
          IlvCoordinateTransformation.CreateTransformation(fileCS, managerCS)
;
    // Retrieve the first feature.
    IlvMapFeature f = reader.getNextFeature();
    int layersCount = manager.getLayersCount();
    manager.setContentsAdjusting(true);
    while (f != null) {
      // Create graphic object.
      IlvGraphic g = renderer.makeGraphic(f, tr);
      if (g != null) {
        // Add it to the manager.
        manager.addObject(g, layersCount, false);
        // Retrieve attributes.
        IlvFeatureAttributeProperty ap = f.getAttributes();
        if (ap != null) {
          // Copy the attributes into the graphic, if any.
          g.setNamedProperty(ap.copy());
        }
      }
      // Loop on all features.
      f = reader.getNextFeature();
    }
  }
```

The DTED file reader

The complete source code for the DTED® file reader example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/readers/src/DTEDReader.java`**.

```
/**
 * Example of how to use the DTED reader package.
 */
public class DTEDReader {
  IlvManager manager = new IlvManager();
  IlvManagerView view = new IlvManagerView(manager);

  /**
   * Loads a single DTED tile.
   */
  public IlvGraphic loadSingleFrame(String fileName)
  throws IlvMapFormatException,
         FileNotFoundException,
         IOException,
         IlvMapRenderException,
         IlvCoordinateTransformationException {
    // Instantiate the reader.
    IlvDTEDReader reader = new IlvDTEDReader(fileName);
    // Retrieve the unique feature.
    IlvMapFeature feature = reader.getNextFeature();
    // Retrieve the image renderer.
    IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
    // Transformation.
    IlvGeographicCoordinateSystem gcs = IlvGeographicCoordinateSystem.WGS84;
    IlvCoordinateTransformation tr =
               IlvCoordinateTransformation.CreateTransformation
               (feature.getCoordinateSystem(),gcs);
    // Make the graphic object to be inserted in a IlvManager.
    IlvGraphic graphic = renderer.makeGraphic(feature, tr);
    // Returns it.
    return graphic;
  }

  private int level = 0;

  /**
   * Create a load-on-demand DTED layer providing the directory name.
   */
  public void makeLODLayer(String dirName) {
    // New DTED layer.
    IlvDTEDLayer layer = new IlvDTEDLayer(dirName, level);
    // Add it to the manager.
    manager.addLayer(layer, -1);
    // Center on tile (0, 0).
    layer.fitTransformerToTile(view, 0, 0);
  }

  public static void main(String a[]) {
    javax.swing.SwingUtilities.invokeLater(
     new Runnable() {
      public void run() {
        DTEDReader reader = new DTEDReader();
        try {
          reader.loadSingleFrame(a[0]);
        } catch (IlvMapFormatException e) {
          e.printStackTrace();
        } catch (FileNotFoundException e) {
          e.printStackTrace();
        } catch (IOException e) {
          e.printStackTrace();
        } catch (IlvMapRenderException e) {
          e.printStackTrace();
        } catch (IlvCoordinateTransformationException e) {
          e.printStackTrace();
        }
      }
    }
   );
  }
}
```

The image file reader

The complete source code for this example can be found in the following file:

<installdir> **/jviews-maps86/codefragments/readers/src/ImageReader.java**.

```
/**
 * Example showing how to use the Image reader package.
 */
public class ImageReader {
  IlvManager manager = new IlvManager();

  /**
   * Load a single image. Upper left and lower right coordinates
   * have to be provided.
   */
  public void loadImage(String imageName, IlvCoordinate ul, IlvCoordinate lr)

  throws IlvMapFormatException,
         FileNotFoundException,
         IOException,
         IlvMapRenderException,
         IlvCoordinateTransformationException {
    // Create image reader.
    IlvImageReader reader = new IlvImageReader(imageName, ul, lr);
    // Retrieve the unique feature.
    IlvMapFeature feature = reader.getNextFeature();
    // Retrieve the default renderer.
    IlvFeatureRenderer renderer = reader.getDefaultFeatureRenderer();
    // No reprojection for images.
    IlvCoordinateTransformation tr =
      new IlvCoordinateTransformation(IlvGeographicCoordinateSystem.WGS84,
                                      IlvGeographicCoordinateSystem.WGS84,
                                      new IlvMapAffineTransform());
    // Create the graphic object.
    IlvGraphic graphic = renderer.makeGraphic(feature, tr);
  }

  /**
   * Create a load-on-demand image layer.
    * The pattern, colFmt, rowFmt are used to retreive the image file name
   * from the tile coordinates
   */
  public void loadLOD(IlvRect tileOrigin,
                      String pattern,
                      String colFmt,
                      String rowFmt)
  {
    // Create the tile loader.
    IlvImageTileLoader tileLoader = new IlvImageTileLoader(pattern,
                                                           colFmt,
                                                           rowFmt);
    // Create the tiled layer.
    IlvTiledLayer tiledLayer = new IlvTiledLayer(tileOrigin);
    // Affect the tile loader to the tiled layer.
    tiledLayer.setTileLoader(tileLoader);
    // Add the layer into the manager.
    manager.addLayer(tiledLayer, -1);
  }
```

The map loader

The package `ilog.views.maps.format` provides the class `IlvMapLoader` that you can use to load in a very simple way any file format for which JViews Maps provides a predefined reader (Shapefile, DTED® , and MID/MIF). These predefined readers are described at length in *Readers and writers*.

Loading a predefined map format

To load a predefined map format, use the `load(java.lang.String)` method. This method first tries to determine the file format according to the naming rules set forth in the specifications of the different formats, and initializes the appropriate reader. The method then loads the map into the manager associated with the map loader.

The following example shows how to import a `.shp` file (Shapefile format) into a JViews Maps manager using the map loader:

```
IlvMapLoader loader = new IlvMapLoader(manager);
try {
  loader.load("myShapeFile.shp");
} catch (IlvMapFormatException e) {
  // Occurs if there is a format error in the file.
  e.printStackTrace();
} catch (IOException e) {
  // Occurs if there is another IO error.
  e.printStackTrace();
}
```

A complete example of how to import a file using the map loader and save it in the `.ilv` format can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/use_map_loader/src/UseMapLoader.java`**

Loading nongeoreferenced files

When you load a map into a JViews Maps manager using the map loader, this map is automatically displayed in the coordinate system associated with the manager provided that the format of the source data is georeferenced. The `IlvMapFeatureIterator` interface has an `isGeoreferenced()` method that you can use to know whether a file is georeferenced. Most of the cartographic files are georeferenced. This is the case for files of the DTED format. Some other cartographic formats, such as Shapefile, are not georeferenced.

When loading data from a file that is not georeferenced, and in the absence of any other indications, the map loader is unable to reproject the source data within the target coordinate system (the one associated with the manager).

Note: If you load several source files of the Shapefile format whose projection is unknown in the same manager, objects are positioned correctly. However, if you try to import data of another format in the manager, the relative position of objects from different source formats are inaccurate.

If you read a file whose format is not georeferenced, but you know the coordinate system in which the data is expressed, you can provide this information to the `IlvMapLoader` using the `setDefaultCoordinateSystem(ilog.views.maps.srs.coordsys.IlvCoordinateSystem)` method.

The following example shows how to import a Shapefile whose projection is known to be geographic into a manager that is in a Mercator projected coordinate system:

```
// Initialize the manager for the mercator projection.
IlvProjection p = new IlvMercatorProjection();
IlvProjectedCoordinateSystem pcs =
       new IlvProjectedCoordinateSystem("mercator", p);
IlvCoordinateSystemProperty csProperty =
      new IlvCoordinateSystemProperty(pcs);
manager.setNamedProperty(csProperty);

// Create a map loader.
IlvMapLoader mapLoader = new IlvMapLoader(manager);

// Load other data.
....

// Load a Shapefile expressed in geographic coordinate system. mapLoader.
setDefaultCoordinateSystem(IlvGeographicCoordinateSystem.WGS84);
mapLoader.load("myShapeFile.shp");
```

Specifying a renderer

If you want an `IlvMapLoader` object to use a specific renderer, specify it as the second argument of its `load(ilog.views.maps.IlvMapFeatureIterator, ilog.views.maps.IlvFeatureRenderer)` method, as shown below:

```
IlvFeatureRenderer renderer = new IlvDefaultCurveRenderer();
mapLoader.load(myIterator, renderer);
IlvMapLoader mapLoader = new IlvMapLoader(manager);
```

The file ***<installdir>*** **`/jviews-maps86/codefragments/renderer/src/Viewer.java`** shows how to use specific renderers with a map loader reading data that has the Shapefile format. For specific renderers, see the examples provided in *Creating a colored line renderer* and *Extending an existing renderer*.

Executing this file loads the following two files into the viewer:

♦ The `<installdir>/doc/usermansrc/maps/renderer/HYLINE.SHP` file defines contour lines for the region of Manila (Philippines). The `HYLNVAL` attribute holds the elevation associated with each contour line. Colors are applied to contour lines using the default elevation color model supplied with JViews Maps. See `IlvIntervalColorModel.MakeElevationColorModel`.

♦ The `<installdir>/doc/usermansrc/maps/renderer/PPPOINT.SHP` file defines points representing populated areas in the region of Manila (Philippines). The `PPPTNAME` attribute holds the name of the town, if it is known. In this case, this name appears next to the `IlvMarker` object that represents the town.

Attaching attributes to graphic objects

The map loader can automatically attach attributes to the graphic objects that it creates when it loads a map. For this, use the `setAttachingAttributes(boolean)` method, as shown in the following example:

```
IlvMapLoader loader = new IlvMapLoader(manager);
loader.setAttachingAttributes(true);
try {
  loader.load("myShapeFile.shp");
} catch (IlvMapFormatException e) {
  // Occurs if there is a format error in the file.
  e.printStackTrace();
} catch (IOException e) {
  // Occurs if there is an other IO error.
  e.printStackTrace();
}
```

Extending the IlvMapLoader class

This section shows how to subtype the `IlvMapLoader` class so that it can recognize a file format other than the JViews Maps predefined formats.

The method `makeFeatureIterator(java.lang.String)` of this class creates the reader that recognizes the format of the file specified as its parameter. In the following example, the `IlvMapLoader` class is derived and the method overridden so that it can recognize the format of the polyline file presented in the section *Developing a new reader* and initialize the appropriate reader. It is assumed that the file has the `.pol` extension.

```
public class MyMapLoader extends IlvMapLoader
{
  /**
   * Constructor.
   */
  public MyMapLoader(IlvManager manager) {
    super(manager);
  }
  /**
   * Overrides the makeFeatureIterator method from super class.
   */
  public IlvMapFeatureIterator makeFeatureIterator(String fileName)
    throws IOException
   {
    // Does superclass know the format of provided file?
    IlvMapFeatureIterator result = super.makeFeatureIterator(fileName);
    // If not, try with the polygon reader.
    if (result == null) {
      // Test extension.
      int length = fileName.length();
      // .pol are polylines files.
      if (length > 4) {
        String suffix = fileName.substring(length - 4);
        if (suffix.toLowerCase().equals(".pol")) {
          try {
            return new OptimizedPolylineReader(fileName);
          } catch (IlvMapFormatException e) {
            return null;
          }
        }
      }
    }
    return result;
  }
```

The `makeFeatureIterator` method first attempts to get an `IlvMapFeatureIterator` from its superclass. If the file is not recognized, it tries to determine whether the file extension provided (in this example, `.pol`) corresponds to that of the file to be read. If the result of the test is `true`, it creates the appropriate reader, which in this case is the optimized reader created in *Optimizing the reader*.

If the file does not contain a header, an `IlvMapFormatException` is thrown and the method returns the null pointer to indicate that it was not able to identify the file format.

The complete source code for this customized map loader can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/readers/src/MyMapLoader.java`**

Developing a new reader

In addition to the predefined readers available in JViews Maps, you can write your own reader and customize it.

This section contains an example of an `IlvMapFeatureIterator` that you can use to read polylines that were saved in an ASCII file.

Note: The classes that implement the `IlvMapFeatureIterator` interface are not necessarily file readers. They can also iterate, for example, over the result of a query to a map server.

The file to be read

The ASCII file to be read has been created especially for this example. Its format is very simple and its specifications are as follows:

- ♦ It has a header specifying its format.
- ♦ There is one pair of coordinates (latitude and longitude) per line. These coordinates are expressed in degrees.
- ♦ Lines can contain comments. These comments, when they exist, are merged to form an attribute.
- ♦ Polylines are separated by a blank line.
- ♦ The file has the `.pol` extension.

The ASCII file is shown below:

```
ascii polylines
-1.0 40.0   A 1x1 degree rectangle centered on the
 1.0 40.0    (0,39) point
 1.0 38.0
-1.0 38.0
-1.0 40.0

0.0  90.0   A meridian extending from the North pole to the South pole
0.0 -90.0
```

The reader

This section shows the reader you can use to read this polyline file.

The complete source code for this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/newreader/src/SimplePolylineReader.java`**

Note: Only the portions of code that require comments are reproduced here.

As shown below, the `SimplePolylineReader` implements the `IlvMapFeatureIterator` interface:

```
class SimplePolygonReader implements IlvMapFeatureIterator
{
 ... member variables ...
 /**
 * Constructor
 */
 public SimplePolygonReader (String fileName) throws FileNotFoundException
 {
 Initializing member variables
 }
 ... methods ...
}
```

The georeferencing methods

Since latitude and longitude in the polyline file are expressed in degrees, the coordinate system is geographic. This is why the `isGeoreferenced()` method returns `true` and the `getCoordinateSystem()` method returns `IlvGeographicCoordinateSystem.WGS84`. The `getCoordinateSystem` method would return `null` if the projection of the file to be read was unknown. See the description of the `isGeoreferenced` method in *The IlvMapFeatureIterator interface*.

```
public boolean isGeoreferenced()
  {
    return true;
  }
 public IlvCoordinateSystem getCoordinateSystem()
  {
    return IlvGeographicCoordinateSystem.WGS84;
  }
```

Bounding box methods

Because of the data format, the bounding box of the polyline cannot be retrieved until the data has been read. Here, the methods `getUpperLeftCorner()` and `getLowerRightCorner()` return `null` to indicate that these points are not known. Another option is to read the data, place it in an array, and then compute the bounding box.

```
public IlvCoordinate getLowerRightCorner()
  {
    return null;
  }
```

Rendering methods

The `getDefaultFeatureRenderer()` method must return a renderer able to transform into graphic objects all the map features read by this feature iterator. The `IlvDefaultCurveRenderer` can process map features whose geometry is of type `IlvMapLineString`.

```
public IlvFeatureRenderer getDefaultFeatureRenderer()
  {
    return new IlvDefaultCurveRenderer();
  }
```

If the geometries of the returned map features are not predefined but instead are instances of a derived class, or if the map feature attributes store drawing parameters to be used in rendering operations such as color or line width, it is necessary to provide renderers that can process these attributes or derived geometries. See the section *Creating a colored line renderer*.

The getNextFeature method

The `getNextFeature()` method reads the geometry of a map feature and creates an `IlvMapFeature` object that will hold all the information required to process the geometry. The geometry read in the code example that follows is an `IlvMapLineString`, which is the class to define polyline geometries.

```
public IlvMapFeature getNextFeature()
  throws IOException
{
  return readPolyline();
}
```

The polyline points are read by the private method `readPolyline`. This method reads each line in the file to extract the coordinates of the points and the related comments, if any.

It is broken up as follows:

1. A geometry of the type `IlvMapLineString` is created, which will be associated with the map feature.

   ```
   private IlvMapFeature readPolyline()
     throws IOException
   {
     // Concatenates all the comment lines.
     StringBuffer buffer = new StringBuffer();
     // Reads the current map feature.
     IlvMapFeature feature = new IlvMapFeature();
   ```

```
  // Reads the current line string geometry.
  IlvMapLineString geometry = new IlvMapLineString();
```

2. The points making up this line string are read and the related comment is stored as an attribute.

```
// Stores the line of text that is read.
String line;

// Reads a line.
while ((line = file.readLine()) != null) {
  [...]
```

3. The longitude coordinate values are read. Note that an exception of type `IlvMapFormatException` is thrown if a format error is detected while reading.

```
// Process longitude.
 IlvCoordinate c = new IlvCoordinate();

 if (tokenizer.hasMoreElements() == false)
   throw new IlvMapFormatException("Longitude coordinate expected");
 try {
   currentToken = (String)tokenizer.nextElement();
   c.x = decimalParser.parse(currentToken).doubleValue();
 } catch (ParseException e) {
   throw new IlvMapFormatException("Error while parsing longitude");
 }
```

   These comments also apply to latitude coordinates.

4. Each point read from the file is added to the line string geometry.

```
// Add this point to geometry.
   geometry.addPoint(c);
   [...]
```

5. The following `if` statement tests whether the end of the file has been reached. In this case, the `getNextFeature` method should return a `null` pointer.

```
// End of file.
 if ((line == null) && (geometry.getPointCount() == 0))
   return null;
```

6. The geometry is associated with the map feature.

```
// Initialize the map feature.
 feature.setGeometry(geometry);
```

7. The comments are extracted to form attributes, which are associated with the map feature. The `attributeInfo` object, which is shared by the attributes of the map features, was initialized in the reader's constructor.

```
// Set attribute.
IlvStringAttribute[] attribute = new IlvStringAttribute[1];
if (buffer.length() > 0) {
  attribute[0] = new IlvStringAttribute();
  attribute[0].setString(buffer.toString());
} else {
  attribute[0] = null;
}
feature.setAttributeInfo(attributeInfo);
feature.setAttributes(new IlvFeatureAttributeProperty(attributeInfo,
                                                           attribute));
```

8. The read map feature is returned.

```
// Returns the read map feature.
   return feature;
 }
```

Optimizing the reader

The polyline reader presented in the section *Developing a new reader* is not the best because it generates a large number of temporary objects. What happens is that each time a polyline is read, a list, and hence memory, is allocated to store its points. If the number of polylines in the file is very high, this might cause memory to become fragmented and also the frequent running of the garbage collector, thus impairing performance.

To improve performance, the simple polyline reader can be optimized so that the `IlvMapFeatureIterator` always returns the same instance of `IlvMapFeature`. The list for storing the points will be allocated only once--when these points are read for the first time. During subsequent readings, the list will be reallocated only if necessary. To make this possible, the map feature returned by the `getNextFeature()` method is volatile, meaning that its geometry and attributes must be used before the method is called again. All the readers provided in the JViews Maps library that implement the `IlvMapFeatureIterator` interface work this way.

The complete source code for this optimized reader example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/newreader/src/OptimizedPolylineReader.java`**

The `readPolyline` method of the class `OptimizedPolylineReader` resets the points making up the `IlvMapLineString` geometry to 0. Note that this geometry is now a field of the class.

```
 geometry.removeAll();
```

The points of each polyline read are stored in a coordinate buffer. New instances of `IlvCoordinate` are created only if the polyline being read has more points than each of the polylines previously read.

```
if (currentPointNum < oldPointNum) {
  // Use an IlvCoordinate that was already allocated.
  c = (IlvCoordinate)points.elementAt(currentPointNum);
} else {
  c = new IlvCoordinate();
  points.addElement(c);
}
```

Once the `x,y` coordinates of a polyline point are read, this point is added to the geometry. Geometries are implemented in such a way that the `addPoint` method does not reallocate memory if one of the previously read geometries had at least as many points as the current geometry that is being read.

```
  geometry.addPoint(c);
```

Also, the geometry attributes are stored as fields of the `OptimizedPolylineReader` class and modified in the `readPolyline` method.

```
if (buffer.length() > 0) {
  stringAttribute.setString(buffer.toString());
  attributeProperty.setAttribute(0,stringAttribute);
} else {
  attributeProperty.setAttribute(0,null);
}
feature.setAttributes(attributeProperty);
```

Map GUI interactors

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**

Pan

The `IlvMapPanInteractor` allows the user to pan a manager view by pressing and holding the space key. Once installed, it is always available.

To create this interactor and attach it to a `IlvManagerView`, you can use the following code:

```
IlvManagerView view = ...
...
IlvMapPanInteractor pan = new IlvMapPanInteractor();
pan.setView(view);
```

Zoom rectangle

The `IlvMapZoomInteractor` allows the user to select and zoom in on a rectangle in a manager view. The Zoom Rectangle interactor includes the Zoom In and Zoom Out interactors. It also provides simple click zoom and unzoom interactions.To create this interactor, you can use the following code:

```
IlvMapZoomInteractor interactor = new IlvMapZoomInteractor();
view.setInteractor(interactor);
```

Continuous zoom

The `IlvContinuousZoomInteractor` allows the user to continuously zoom into or out from a map by pressing and holding a mouse button. The zoom center, that is, the point in a map that will be zoomed on, is the point in the map where the mouse is clicked. By default, pressing the left button zooms into the zoom center, pressing the right mouse button zooms out from the zoom center. When you drag the mouse the current zoom activity is paused and the map is panned to follow mouse displacement.

This interactor also redirects mouse wheel actions. After this interactor is activated, the user scrolls the mouse wheel backwards to zoom in, and forwards to zoom out.

The delay and zoom factor can be configured for each interactor instance.

```
IlvContinuousZoomInteractor interactor = new IlvContinuousZoomInteractor();
interactor.setContinuousZoomFactor(factor);
interactor.setPeriod(period);
view.setInteractor(interactor);
```

To activate this interactor, use the following code:

Rotate

The `IlvManagerViewRotateInteractor` allows the user to rotate the entire content of a manager view. Various modes are available, such as immediate rotation, delayed rotation, and key controlled rotation.

To create this interactor, you can use the following code:

```
IlvManagerView view = …;
...
IlvManagerViewRotateInteractor interactor = new
   IlvManagerViewRotateInteractor();
interactor.setMode(IlvManagerViewRotateInteractor.DYNAMIC_CONTINUOUS_MODE);
view.setInteractor(interactor);
```

Distance measuring

The `IlvMakeMeasureInteractor` allows the user to draw a line on a map in a manager view and automatically display the distance represented by the line on the map.

```
final IlvMakeMeasureInteractor measure = new IlvMakeMeasureInteractor();
 JButton b = new JButton();
    b.addActionListener(new ActionListener() {
      public void actionPerformed(ActionEvent e) {
        view.setInteractor(measure);
      }
    });
```

To create a `IlvMakeMeasureInteractor`you can use the following code:

The See-Through interactor

The IlvSeeThroughInteractor allows the user to temporarily see a part of a map independently from the current layer visibility states. This means that users can interactively reveal layers beneath a layer over which the mouse is pressed.

With this interactor, the user drags a square over the map, which displays the layers registered for this interactor. IlvSeeThroughConfigurationPanel is provided as a default configuration tool to interactively select layers in a Swing GUI.

```
IlvSeeThroughInteractor seeThrough = new IlvSeeThroughInteractor();
// configure the interactor
IlvSeeThroughConfigurationPanel panel = new
          IlvSeeThroughConfigurationPanel(manager, seeThrough);
JOptionPane pane = new JOptionPane();
pane.setMessage(panel);
JDialog dialog = pane.createDialog(view, title);
panel.update();
dialog.setVisible(true);
// attach to the view.
view.setInteractor(seeThrough);
```

To create this interactor and attach it to an IlvManagerView, use the following code:

Using the GUI beans

Describes the JavaBeans™ for GUIs with maps.

In this section

The Coordinate Viewer
Describes the Coordinate Viewer bean available for GUIs with maps.

The Map Layer Tree
Describes the Map Layer Tree beans and property editors

The Toolbar
Describes the Toolbar bean available for GUIs with maps.

Multithread Monitor
Describes the Multithread Monitor bean available for GUIs with maps.

Coordinate Panel Factory
Describes the Coordinate Panel Factory bean available for GUIs with maps.

Compass
Describes the Compass bean available for GUIs with maps.

Using annotations
Explains how to use the Annotations Toolbar bean available for GUIs with maps.

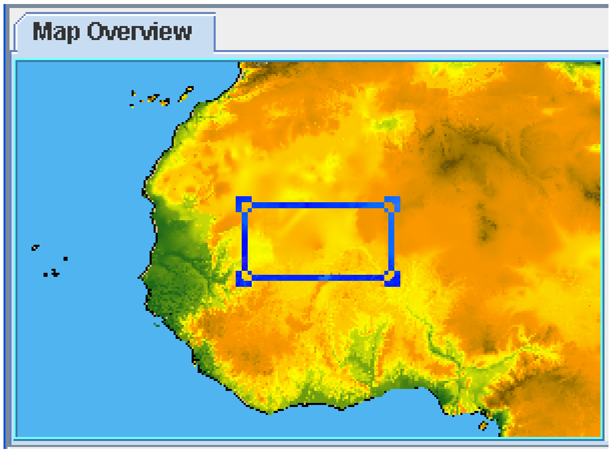
The Symbology Tree View bean
Describes the Symbology Tree View bean and how to use it.

Map Overview

The Map Overview bean is represented by the `IlvJOverview` class. The Map Overview bean displays a representation of a target manager view and enables users to navigate to an area of the view that is of particular interest.

An example of the Map Overview panel is shown in *Overview panel* .

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/index.html`**



Overview panel

Including the bean in an application

To include the Map Overview bean in your application, write the following lines of code:

```
IlvJOverview overview = new IlvJOverview();
overview.setView(view);
```

Adding the bean to a Swing component

Then add it into a swing component:

```
panel.add(overview, BorderLayout.CENTER);
```

You should also make sure that the size of your overview remains the same regardless of any Swing layout constraints applied. To do this, use lines such as the following:

```
overview.setSize(200, 100);
overview.setPreferredSize(overview.getSize());
overview.setMinimumSize(overview.getSize());
```

Customizing interactions and capabilities

You can customize the way users interact with the overview, and with the capabilities of the control rectangle as follows:

♦ To change the way the interactor is drawn:

```
overview.setDrawingStyle(IlvManagerMagViewInteractor.Wire);
overview.setLineWidth(3);
overview.setResizeHandleSize(6);
```

♦ To allow users to resize the overview:

```
overview.setResizingAllowed(true);
```

♦ To make sure the visible area of the view always remains inside the overview window:

```
overview.setAutoTranslating(true);
```

♦ To maintain a constant zoom ratio between the overview and the view:

```
overview.setAutoZooming(true);
```

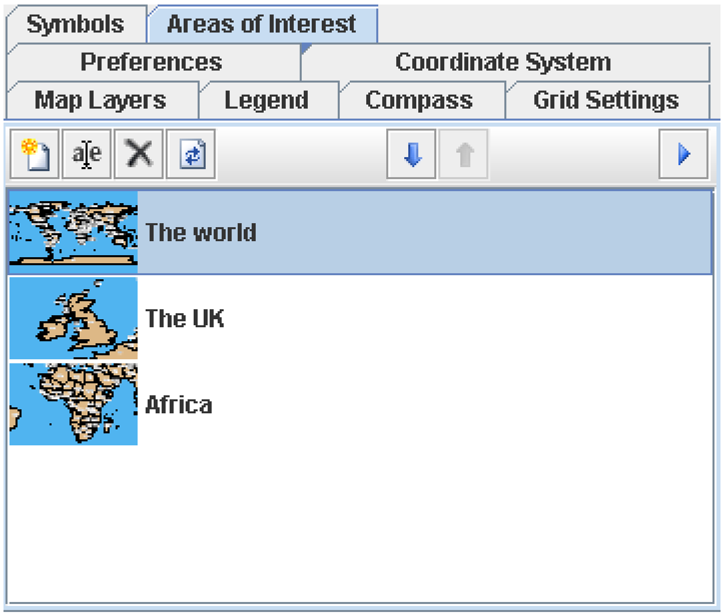
♦ You can improve the performance of your application by limiting the overview refresh rate:

```
overview.getOverviewView().setRepaintSkipThreshold(500);
```

Area of Interest panel

The Area of Interest panel bean is represented by the `IlvJAreaOfInterestPanel` class. The Area of Interest panel bean enables users to select and display frequently used areas of a map.

An example of the Area of Interest panel is shown in *Area of Interest panel*.



Area of Interest panel

Including the bean in an application

To include the Area of Interest panel bean in your application, write the following line of code:

```
IlvJAreaOfInterestPanel areaPanel = new IlvJAreaOfInterestPanel(view, true,
true, true);
```

The first boolean value indicates whether users can add areas of interest, the second boolean allows them to remove areas of interest, and the third boolean allows them to rename areas of interest.

Adding the bean to a Swing container

You then have to insert the bean into your swing user interface:

```
panel.add(areaPanel, BorderLayout.CENTER);
```

The Area of Interest panel bean then attaches itself and listens to the `IlvAreasOfInterestProperty` property of the manager of the view. Whenever an area is added to the underlying `IlvAreaOfInterestVector`, its name and preview icon are displayed on the Area of Interest panel bean.

Managing the Area of Interest and preview images

This panel also provides interesting static utility methods to manage the preview images and the creation of the area of interest.

To create an area for everything visible on the view (this is the method usually called when clicking the **New Area of Interest** button), you can insert the following lines of code in your application:

```
IlvAreaOfInterest
currentArea=IlvJAreaOfInterestPanel.createLocationFromView(view,64,false);
currentArea.setName("Current Area");
```

If you want the users to provide the area name themselves (by means of a dialog box popup) use:

```
IlvAreaOfInterest
currentArea=IlvJAreaOfInterestPanel.createLocationFromView(view,64,true);
```

This bean also provides a method that updates the area of interest preview icon with what would be visible with the current map settings and contents:

```
IlvJAreaOfInterestPanel.refreshPreview(view,area,maxDimension);
```

For example:

♦ To create an area for Europe bounds:

  ```
  IlvRect rectangle=new
  IlvRect((float)Math.toRadians(15),(float)Math.toRadians(35),(float)Math.toRa
  dians(45),(float)Math.toRadians(25));
  IlvAreaOfInterest europe=new IlvAreaOfInterest("Europe",rectangle,0,null);
  ```

♦ To update its preview image:

  ```
  IlvJAreaOfInterestPanel.refreshPreview(view,europe,64);
  ```

♦ You would then have to add the area to the property of the manager for proper management by the Area of Interest bean:

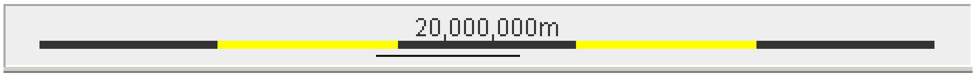
  ```
  IlvAreaOfInterestVector
  ```

```
areas=IlvAreasOfInterestProperty.GetAreasOfInterest(view.getManager());
areas.addElement(europe);
```

Scale Bar

The Scale Bar bean is represented by the `IlvJAutomaticScaleBar` class. This bean enables users to estimate distances using the displayed scale bar.

An example of the Scale Bar is shown in *Scale Bar* .



Scale Bar

Including the bean in an application

To include the Scale Bar bean in your application, write the following lines of code:

```
IlvJAutomaticScaleBar graphicScale = new IlvJAutomaticScaleBar();
graphicScale.setView(view);
```

Adding the bean to a Swing container

You can then add this bean into your Swing hierarchy.

```
panel.add(graphicScale, BorderLayout.SOUTH);
```

The Scale Bar bean then attaches itself and listens to the `IlvDisplayPreferencesProperty` property of the manager of the view. Whenever the underlying `IlvDisplayPreferences` are changed (see *Display Preferences Editor*), the units used to display distances change.

You can also force the scale bar bean to use specific units, regardless of the `IlvDisplayPreferencesProperty` settings, by calling the `setFarUnit(ilog.views.maps.IlvLinearUnit)` and `setNearUnit(ilog.views.maps.IlvLinearUnit)` methods in your application code.

Changing the appearance of the scale bar

The following controls change the appearance of the scale bar:

♦ To set the margins between the scale bar and the border of the component:

♦ To set the text spacing between the scale text and the scale bar:

```
graphicScale.setMarginHeight(2);
graphicScale.setMarginWidth(2);
graphicScale.setTextSpacing(2);
```

♦ To set the style used to draw the scale bar (double or single scale – alternate colors or not):

```
graphicScale.setScaleStyle(IlvScaleBar.DOUBLE_DASH_SCALE_EVEN);
```

♦ To change the way labels are displayed:

```
graphicScale.setScaleTextMode(IlvScaleBar.THREE_LABELS);
```

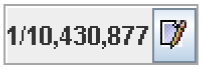
♦ To set the height of the scale bar:

```
graphicScale.setScaleHeight(10);
```

Scale Control Bar

The Scale Control Bar bean is represented by the `IlvJMapScaleControl` class. This bean enables users to see the scale of the map. The button also enables users to control the scale interactively by entering a new setting.

An example of the Scale Control Bar is shown in *Scale Control Bar*.



Scale Control Bar

Including the bean in an application

To include the Scale Control Bar bean in your application, write the following lines of code:

```
IlvJMapScaleControl scaleControl = new IlvJMapScaleControl();
scaleControl.setView(view);
```

Adding the bean to a Swing container

You can then add this bean in your Swing hierarchy.

```
panel.add(scaleControl, BorderLayout.SOUTH);
```

The Scale Control Bar bean then attaches itself and listens for scale changes for the view in order to update the displayed scale.

Customizing the appearance and behavior

You can prevent users from entering changes to the map scale by means of this bean (this makes the **Control Scale** button disappear) using:

```
scaleControl.setAllowScaleEdition(false);
```

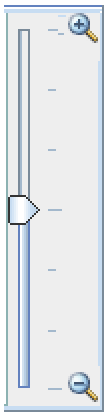
If the way the scale is presented is not appropriate for your needs, many tools are provided to change the prefix (“1/”), suffix, or even the number formatter:

```
scaleControl.setPrefix(null);
scaleControl.setSuffix("th");
scaleControl.setScaleFormat(new DecimalFormat());
```

Zoom Control panel

The Zoom Control panel bean is represented by the `IlvJAdvancedZoomControl` class. This bean enables users to zoom in and zoom out on a map at a rate determined by the displacement of the pointer.

An example of the Zoom Control panel is shown in *Zoom Control panel* .



Zoom Control panel

Including the bean in an application

To include the Zoom Control panel in your application, write the following lines of code:

```
IlvJAdvancedZoomControl zoomer = new IlvJAdvancedZoomControl();
zoomer.setView(view);
```

Adding the bean to a Swing container

You can then add this bean to a Swing container:

```
panel.add(zoomer, BorderLayout.WEST);
```

Customizing the appearance and behavior

This bean can be presented in horizontal or vertical layout, using:

```
zoomer.setOrientation(SwingConstants.VERTICAL);
```

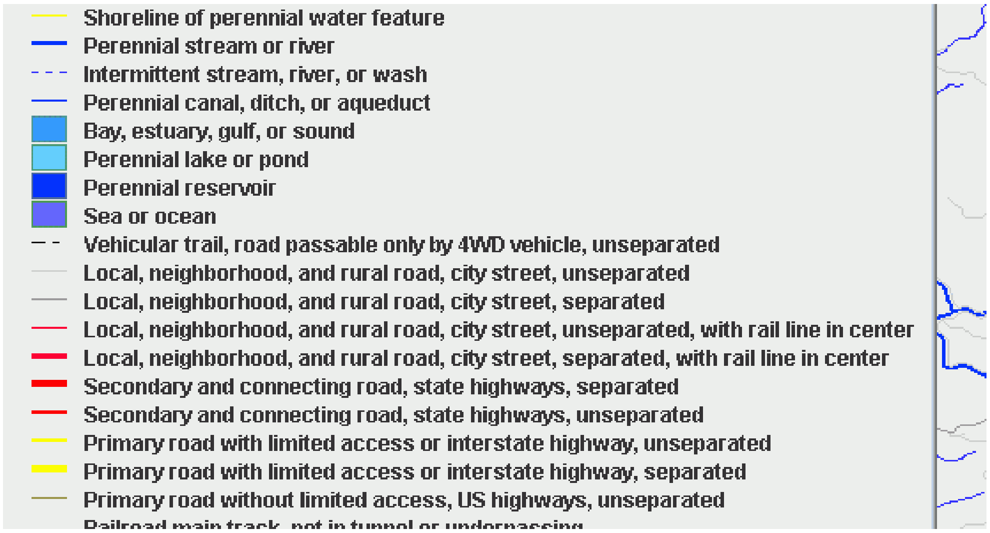
By default, this bean zooms in or out of the view at a maximum rate of 6% every 40ms. You can change these default values to better suit your needs using, for example:

```
zoomer.setMaxZoomingRatio(1.1);
zoomer.setZoomDelay(100);
```

Legend panel

The Legend panel bean is represented by the `IlvMapLegendPanel` class. This bean enables users to create a legend to define map elements.

An example of the Legend panel bean is shown in *Legend panel bean* .



— Shoreline of perennial water feature
— Perennial stream or river
- - - Intermittent stream, river, or wash
— Perennial canal, ditch, or aqueduct
🟦 Bay, estuary, gulf, or sound
🟦 Perennial lake or pond
🟦 Perennial reservoir
🟪 Sea or ocean
- - Vehicular trail, road passable only by 4WD vehicle, unseparated
— Local, neighborhood, and rural road, city street, unseparated
— Local, neighborhood, and rural road, city street, separated
— Local, neighborhood, and rural road, city street, unseparated, with rail line in center
— Local, neighborhood, and rural road, city street, separated, with rail line in center
— Secondary and connecting road, state highways, separated
— Secondary and connecting road, state highways, unseparated
— Primary road with limited access or interstate highway, unseparated
— Primary road with limited access or interstate highway, separated
— Primary road without limited access, US highways, unseparated
Railroad main track, not in tunnel or underpassing

Legend panel bean

Including the bean in an application

To include the Legend panel bean in your application, write the following lines of code:

```
IlvMapLegendPanel legend = new IlvMapLegendPanel();
legend.setView(view);
```

Adding the bean to a Swing container

You can then add this bean into the Swing hierarchy of your application:

```
panel.add(legend, BorderLayout.EAST);
```

The Legend panel bean then attaches itself and listens to the `IlvMapLayerTreeProperty` property of the manager of the view. Whenever a layer is changed in the underlying `IlvMapLayerTreeModel`, the legend updates itself.

Customizing the appearance and behavior

If you do not want an automatic `JScrollPane` around your legend, you can use:

```
IlvMapLegend legend = new IlvMapLegend ();
legend.setView(view);
```

The Legend panel uses the properties of maplayers to retrieve the `IlvMapStyle.CATEGORY` and `IlvMapStyle.LEGEND_GROUP` string properties (for details of these properties, see *Common styling properties*).

By default (when these values are null), each map layer has its own line in the legend. The user can override this by setting identical values in the `Category` or `Legend Group` fields.

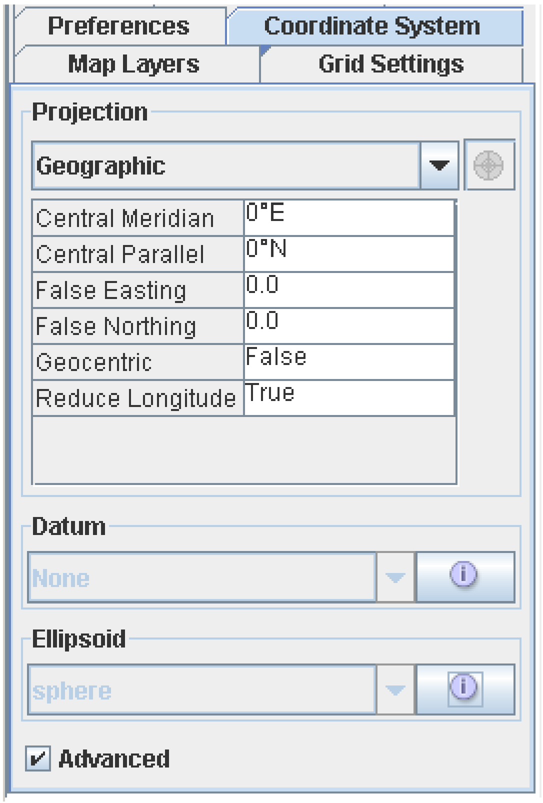
Layers are ordered according to their legend group (all layers with the same legend group are displayed together, inside the same frame).

When layers share the same legend group and category, they are displayed on a single legend line – possibly with more than one legend caption to display all the different aspects this legend item can have.

Coordinate System Editor

The Coordinate System Editor bean is represented by the `IlvJCoordinateSystemEditorPanel` class. This bean enables users to set the coordinate system used to display a map view.

An example of the Coordinate System Editor is shown in *Coordinate System Editor* .



Central Meridian	0°E
Central Parallel	0°N
False Easting	0.0
False Northing	0.0
Geocentric	False
Reduce Longitude	True

Coordinate System Editor

Including the bean in an application

To include the Coordinate System Editor in your application, you first need to create the panel:

```
IlvJCoordinateSystemEditorPanel csPanel = new
IlvJCoordinateSystemEditorPanel();
```

To display the current coordinate system of the view in the bean, retrieve the `IlvCoordinateSystemProperty` property:

```
csPanel.setCoordinateSystem(IlvCoordinateSystemProperty.GetCoordinateSystem
(vie
w.getManager()));
```

Then, you should add a listener to the bean that updates the coordinate system of the view when the bean changes, retrieving the new value selected by the user:

```
csPanel.addCoordinateSystemChangeListener(new PropertyChangeListener() {
  public void propertyChange(PropertyChangeEvent evt) {
```

```
    view.getManager().setNamedProperty(new
IlvCoordinateSystemProperty(csPanel.getCoordinateSystem()));
  }
});
```

Adding the bean to a Swing hierarchy

You can then add this bean to your Swing hierarchy.

```
panel.add(dataSourcePanel, BorderLayout.CENTER);
```

Customizing the appearance and behavior

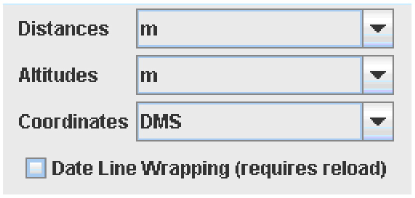
To configure the Coordinate System Editor as a simple projection choice combo-box, you can disable the Advanced Property panels. For example:

```
csPanel.setAdvancedPanelsVisible(false);
csPanel.setAdvancedCheckBoxVisible(false);
```

Display Preferences Editor

The Display Preferences Editor bean is represented by the `IlvJDisplayPreferencesEditorPanel` class. This bean enables users to set a map view to the units of their choice.

An example of the Display Preferences Editor is shown in *Display Preferences Editor* .


Distances: m
Altitudes: m
Coordinates: DMS
☐ Date Line Wrapping (requires reload)

Display Preferences Editor

Including the bean in an application

To include the Display Preferences Editor bean in your application, you first need to create the panel:

```
IlvJDisplayPreferencesEditorPanel prefsPanel = new
IlvJDisplayPreferencesEditorPanel();
```

Set the current preference properties of the view in the bean:

```
prefsPanel.setDisplayPreferences(IlvDisplayPreferencesProperty.GetDisplayPrefer
ences(view.getManager()));
```

Then, you can add a listener on the bean that changes the view preferences when the user changes a preference:

```
prefsPanel.addDisplayPreferencesChangeListener(new PropertyChangeListener() {

  public void propertyChange(PropertyChangeEvent e) {
    IlvDisplayPreferences system = (IlvDisplayPreferences) e.getNewValue();
    view.getManager().setNamedProperty(new
IlvDisplayPreferencesProperty(system));
  }
});
```

Adding the bean to a Swing hierarchy

You can then add this bean to your Swing hierarchy.

```
panel.add(prefsPanel, BorderLayout.WEST);
```

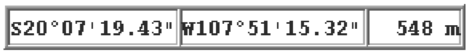
Using the API to define bean items

You can also use the API to define the distance or altitude units, date line wrapping or the coordinate formatter selected in the bean with the `setAltitudeUnit(ilog.views.maps.IlvLinearUnit)` `setDistanceUnit(ilog.views.maps.IlvLinearUnit)`, `setUsingGeodeticComputation(boolean)` or `setCoordinateFormatter(ilog.views.maps.IlvCoordinateFormatter)` methods.

The Coordinate Viewer

The Coordinate Viewer bean is represented by the `IlvJMouseCoordinateViewer` class. This bean displays the current coordinates of the mouse when it is moved over a map view.

An example of the Coordinate Viewer is shown in *Coordinate Viewer* .


S20°07'19.43" W107°51'15.32" 548 m

Coordinate Viewer

Including the bean in an application

To include the Coordinate Viewer bean in your application, write the following lines of code:

```
IlvJMouseCoordinateViewer coordViewer = new IlvJMouseCoordinateViewer();
coordViewer.setView(view);
```

Adding the bean to a Swing hierarchy

You can then add this bean to your Swing hierarchy.

```
panel.add(coordViewer, BorderLayout.SOUTH);
```

The Coordinate Viewer bean then attaches a Mouse Motion Listener to the view and displays the information according to the properties of the manager:

- ♦ The `IlvCoordinateSystemPropertyEditor` is used to transform screen location coordinates into longitude/latitude information.
- ♦ The `IlvAltitudeProviderProperty` is used to retrieve the altitude at that longitude/latitude location, if available.

The coordinates and altitude are then transformed into human readable strings using `IlvDisplayPreferencesProperty` unit management and formatters.

Depending on the coordinate formatter, latitude/longitude/altitude may not be the only fields. For example, a UTM coordinate system displays the zone, zone number, easting and northing information. The Coordinate Viewer uses `JLabel` HTML capabilities to display the different parts of the coordinates as an HTML table.

Customizing the appearance

You can configure the appearance of the bean using, for example:

```
coordViewer.setHtmlTableProperties("border=1 cellpadding=0 cellspacing=0");
```

The Map Layer Tree

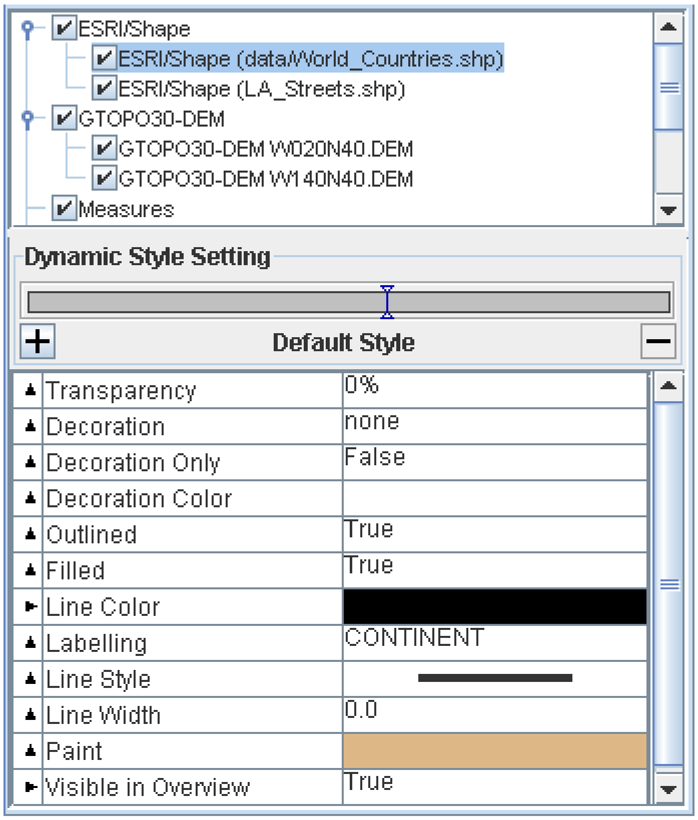
Describes the Map Layer Tree beans and property editors

In this section

The Map Layer Tree bean

The Map Layer Tree panel bean is represented by the `IlvLayerTreePanel` class. The Map Layer Tree bean displays the layer tree, the Map Style Property Sheet bean and a number of other beans that are accessed from this panel.

An example of the Map Layer Tree panel is shown in *Map Layer Tree panel* .



- ✓ ESRI/Shape
  - ✓ ESRI/Shape (data/World_Countries.shp)
  - ✓ ESRI/Shape (LA_Streets.shp)
- ✓ GTOPO30-DEM
  - ✓ GTOPO30-DEM W020N40.DEM
  - ✓ GTOPO30-DEM W140N40.DEM
- ✓ Measures

Dynamic Style Setting

Default Style

Transparency	0%
Decoration	none
Decoration Only	False
Decoration Color	
Outlined	True
Filled	True
Line Color	■
Labelling	CONTINENT
Line Style	▬
Line Width	0.0
Paint	🟫
Visible in Overview	True

Map Layer Tree panel

Including the bean in an application

To include the Map Layer Tree panel bean in your application, write the following lines of code:

```
IlvLayerTreePanel  layerTreePanel = new IlvLayerTreePanel();
layerTreePanel.setView(view);
```

Adding the bean to a Swing container

You can then add this bean to your Swing hierarchy.

```
panel.add(layerTreePanel, BorderLayout.WEST);
```

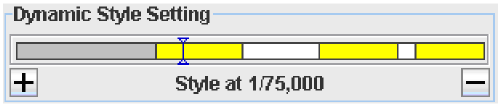
The Map Layer Tree panel bean then attaches itself and listens to the `IlvMapLayerTreeProperty` property of the specified manager. Whenever a layer is added or changed in the underlying `IlvMapLayerTreeModel`, its name is added to the tree, and the layer style properties (such as colors or view visibilities) are made available for modification in the lower part of the panel.

The Map Layer Tree panel also attaches itself to the selection mechanism of the manager in order to select the layer when a graphical map object is selected.

The Dynamic Style Setting panel bean

The Dynamic Style Setting panel bean is represented by the `IlvMapDynamicStylePanel` class.

An example of the Dynamic Style Setting panel is shown in *Dynamic Style Setting panel* .



Dynamic Style Setting panel

The Dynamic Style Setting panel displays the different styles a layer can have at different zoom factors. The gray bar furthest to the left is the default style (used for the finer/higher zoom levels of a map) – the other dynamic style settings are represented by alternating yellow and white bars. Users can also use the panel to set the map scale by clicking on the dynamic styles bar.

Accessing and updating the panel

You should not need to create a Dynamic Style Setting panel directly. You can access the Style Setting panel of the Map Layer Tree panel using the `getThemePanel` method.

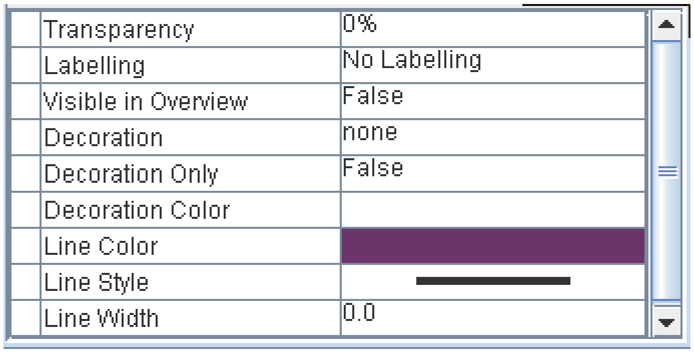
This panel is attached to the `IlvMapStyleControllerProperty` property of the manager (and its underlying `IlvMapStyleController`). It is updated when the user adds/deletes dynamic styles (using the + and − buttons, see *Dynamic Style Setting panel*), or when you add dynamic styles with API calls such as:

```
IlvMapStyleController themeControl = IlvMapStyleControllerProperty.
   GetMapStyleController(view.getManager());
themeControl.addTheme(0.001,mapLayer,"new style");
themeControl.getStyle(mapLayer,0.001).setVisibleInView(true);
```

The Map Style Property Sheet bean

The Map Style Property Sheet bean is represented by the `IlvMapStylePropertySheet` class.

An example of the Map Style Property Sheet is shown in *Map Style Property Sheet* .



Transparency	0%
Labelling	No Labelling
Visible in Overview	False
Decoration	none
Decoration Only	False
Decoration Color	
Line Color	🟪
Line Style	▬
Line Width	0.0

Map Style Property Sheet

When a layer is selected, this property sheet is updated with a property list depending on the associated map style.

You do not need to create a Map Style Property Sheet directly. You can access the Map Style Property Sheet of the Map Layer Tree panel using the `getMapStylePropertySheet` method.

Note: By using `IlvMapStyleBeanInfo.setAdvancedMode(false)`, you can limit the list of properties displayed in the Map Style Property Sheet to a restricted list, depending on the class of the map style used.

Property editors

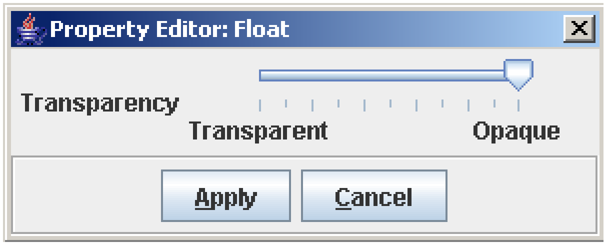
Most of the properties are integers, Booleans (true/false choices) or raw text. Some properties however need a more specific property editor.

JViews Maps does not call the constructor for any of the editors. It uses the standard Java™ `BeanInfo` mapping instead. For example, the `IlvMapStyle` class has the property `alpha` and the `IlvMapStyleBeanInfo` class returns the list of editable properties and associates an `IlvAlphaPropertyEditor` for that property. The `IlvMapStylePropertySheet` class uses this information to create the correct editor (using Java reflection).

Alpha Property Editor

The Alpha Property Editor is represented by the `IlvAlphaPropertyEditor` class.

An example of the Alpha Property Editor is shown in *Alpha Property Editor* .



Alpha Property Editor

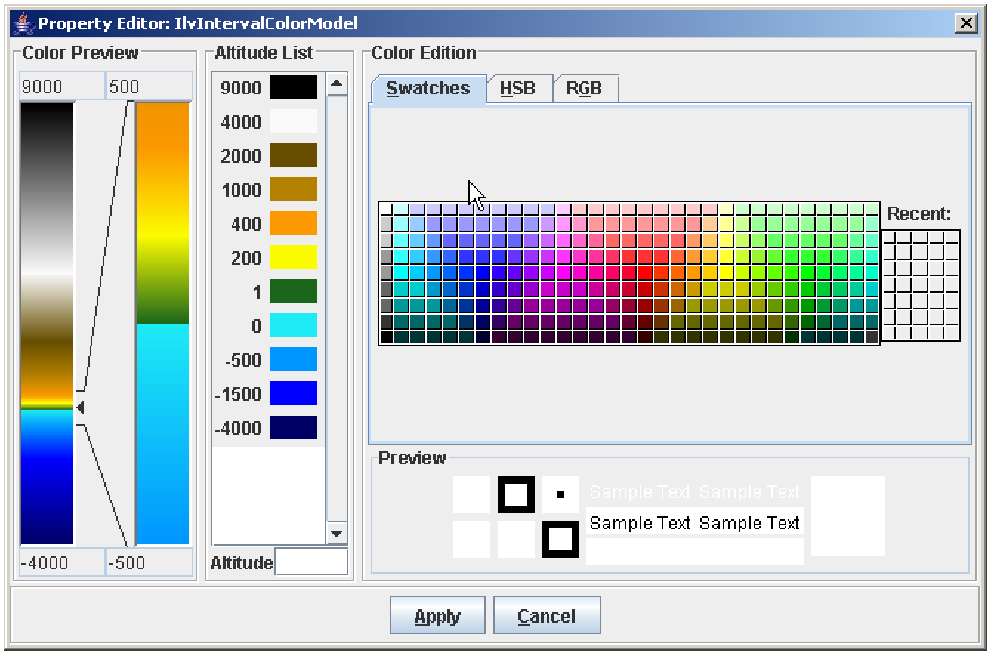
To include the Alpha Property Editor in a new bean, place the following lines in the associated `BeanInfo` class:

```
   ...
PropertyDescriptor alpha = new PropertyDescriptor("alpha",IlvMapStyle.class);
alpha.setPropertyEditorClass(IlvAlphaPropertyEditor.class);
   ...
```

Color Model Property Editor

The Color Model Property Editor is represented by the `IlvColorModelPropertyEditor` class. This property editor enables you to set, for example, the colors for different altitudes in a map that contains altitude data.

An example of the Color Model Property Editor is shown in *Color Model Property Editor* .



Color Model Property Editor

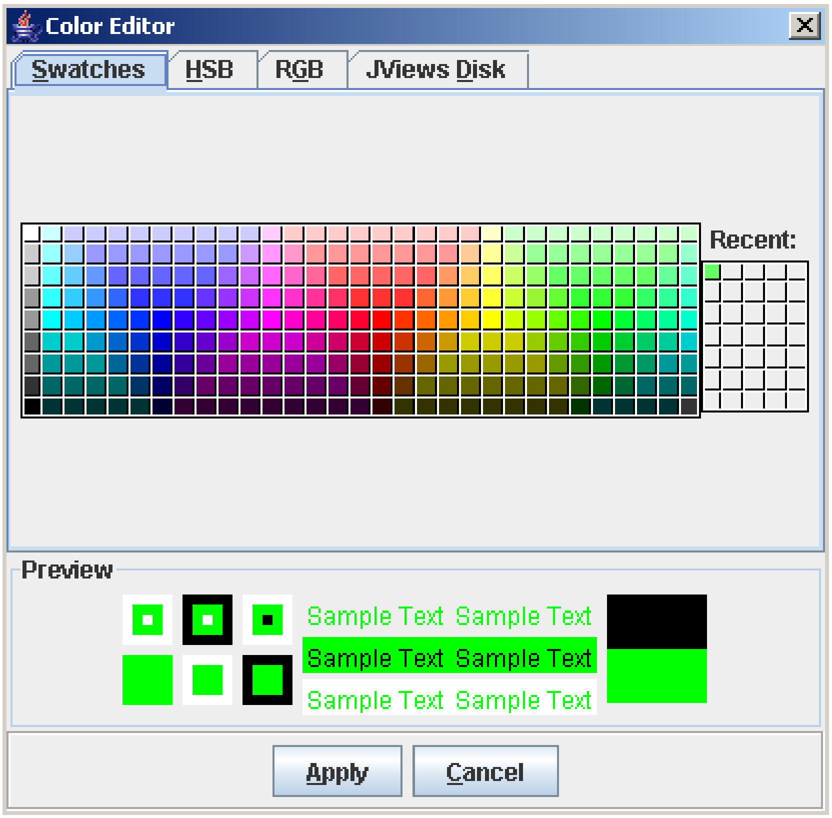
The Color Model Property Editor is implemented in a similar way to the Alpha Property Editor and is used for `IlvRasterStyle` map styles.

To include the Color Model Property Editor in a new bean, use the following lines in a `BeanInfo` class:

```
PropertyDescriptor colorModel = new
PropertyDescriptor("colorModel",IlvRasterStyle.class);
...
colorModel.setPropertyEditorClass(IlvColorModelPropertyEditor.class);
...
```

Color Property Editor

The Color Property Editor enables you to set the color of the map outlines. An example of the Color Property Editor is shown in *Color Property Editor* .



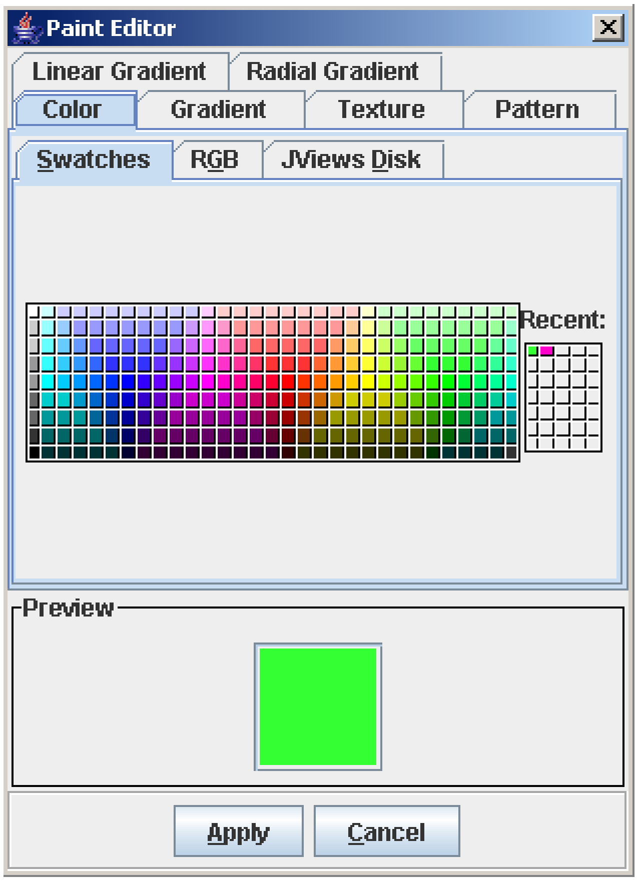
Color Property Editor

The Color Property Editor bean is implemented in a similar way to the other editors and is used in the `IlvGeneralPathStyle` and `IlvPolylineStyle` map styles.

Paint Property Editor

The Paint Property Editor enables you to set the map fill color.

An example of the Paint Property Editor is shown in *Paint Property Editor* .



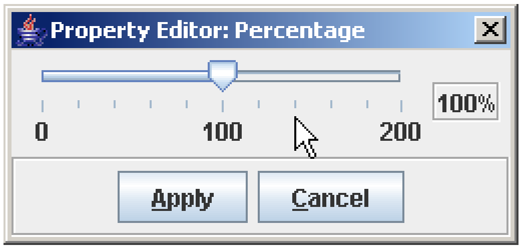
Paint Property Editor

The Paint Property Editor bean is implemented in a similar way to the other editors and is used in the `IlvGeneralPathStyle` and `IlvPolylineStyle` map styles.

Percent Property Editor

The Percent Property Editor is represented by the `IlvPercentPropertyEditor` class. This property editor enables you to set the brightness, saturation, and contrast of the displayed map.

An example of the Percent Property Editor is shown in *Percent Property Editor* .



Percent Property Editor

The Percent Property Editor is implemented in a similar way to the other editors and is used in the `IlvRasterStyle` map styles.

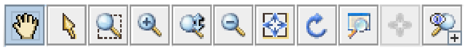
To include the Percent Property Editor in a new bean, use the following lines in a `BeanInfo` class:

...

The Toolbar

The Toolbar bean is represented by the `IlvJMapsManagerViewControlBar` class. This is a subclass of the framework class `IlvJManagerViewControlBar`.

An example of the Toolbar is shown in *Toolbar* .


Toolbar

Including the bean in an application

To include the Toolbar bean in your application, write the following lines of code:

```
PropertyDescriptor brightness = new PropertyDescriptor("brightness",
IlvRasterStyle.class);
...
brightness.setPropertyEditorClass(IlvPercentPropertyEditor.class);
IlvJMapsManagerViewControlBar toolbar = new IlvJMapsManagerViewControlBar();
toolbar.setView(view);
```

Adding the bean to a Swing container

These lines create a standard IBM® ILOG® JViews interactor toolbar that you need to integrate into your Swing GUI:

```
panel.add(toolbar, BorderLayout.NORTH);
```

Customizing the toolbar

For JViews Maps use, you may want to add more interactors or buttons to this toolbar.

Replacing an interactor

You can replace standard interactors with better-tailored interactors, such as the `IlvMapZoomInteractor`, with lines of code such as:

```
IlvMapZoomInteractor zi = new IlvMapZoomInteractor();
// chose the way the rectangle is drawn when rotation exists
zi.setRotationAllowed(true);
// when zoom is selected, it stays, contrary to default JViews.
zi.setPermanent(true);
//to change from default zoom interactor
toolbar.setZoomViewInteractor(zi);
```

Adding a new interactor

You may want to add a completely new interactor:

```
IlvManagerViewInteractor interactor = …;
JToggleButton interactorButton = new JToggleButton(interactorIcon);
```

You have to add a listener to set or pop this interactor when the toggle button is selected:

```
interactorButton.addActionListener(new ActionListener() {
  public void actionPerformed(ActionEvent e) {
        if(interactorButton.isSelected()){
          // set the interactor
          view.setInteractor(interactor);
          // and make sure the view has focus, in case the interactor manages

keyboard accelerators
          view.requestFocus();
        } else if (view.getInteractor()==interactor){
          // pop the interactor
          view.popInteractor();
        }
  }
});
```

You also need to pop this interactor when another one is selected:

```
InteractorListener interactorListener = new InteractorListener() {
  public void interactorChanged(InteractorChangedEvent event) {
        boolean isMyInteractor = (event.getNewValue() == interactor);
        if (interactorButton.isSelected() != isMyInteractor) {
          interactorButton.setSelected(isMyInteractor);
        }
  }
};
view.addInteractorListener(interactorListener);
```

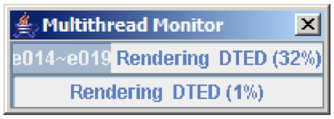
Then, you can add the interactor button to the toolbar:

```
toolbar.add(interactorButton);
```

Multithread Monitor

The Multithread Monitor bean is represented by the `IlvThreadedActivityMonitorPanel` class. This bean displays the completion status of tasks in progress.

An example of the Multithread Monitor is shown in *Multithread Monitor* .



Multithread Monitor

To include the Multithread Monitor bean in your application, you first need to get the thread monitoring model for the manager:

```
IlvThreadedActivityMonitor mon =
IlvThreadedActivityMonitorProperty.GetThreadedActivityMonitor(manager);
```

Then you can create the bean:

```
IlvThreadedActivityMonitorPanel monitor = new
IlvThreadedActivityMonitorPanel(mon);
```

You can then add this bean to your Swing hierarchy.

```
panel.add(monitor, BorderLayout.SOUTH);
```

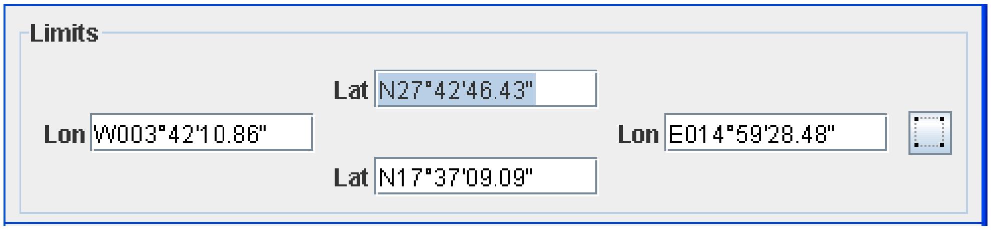
The bean attaches itself to the `IlvThreadedActivityMonitor` of the manager. It can be updated whenever a new activity is registered or updated in this model by using lines such as:

```
mon.updateActivityProgress(myActivity,10,"Doing something...");
...
mon.unregisterThreadedActivity(myActivity);
```

Coordinate Panel Factory

The Coordinate Panel Factory bean is represented by the `IlvCoordinatePanelFactory` class. This bean is used in the Symbol Definition Window and in the terrain analysis features, but can be used whenever the application needs the user to point to a single coordinate or a rectangular area.

An example of the Coordinate Panel Factory for the input of a rectangular set of coordinates is shown in *Multithread Monitor* .



Coordinate Panel Factory

Including the bean in an application

To include the Coordinate Panel Factory bean in your application, you need a coordinate formatter (to determine how coordinates are displayed). You can, for example, retrieve the preferred coordinate formatter for the manager:

```
IlvDisplayPreferences
prefs=IlvDisplayPreferencesProperty.GetDisplayPreferences(manager);
IlvCoordinateFormatter formatter=prefs.getCoordinateFormatter();
```

Then you can create a bean to input single point coordinates:

```
pointInputPanel = IlvCoordinatePanelFactory.createCoordPointInputPanel(view,
   formatter);
```

Or you can create a panel to input a rectangular zone:

```
rectInputPanel = IlvCoordinatePanelFactory.createCoordRectangleInputPanel
(view,formatter);
```

You may then have to set the initial values:

```
pointInputPanel.setLatLon(Math.toRadians(22.5),Math.toRadians(12));
```

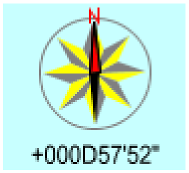
And add a listener that is invoked when the user enters or selects coordinates:

```
PropertyChangeListener listener= ...;
pointInputPanel.addPropertyChangeListener(listener);
```

Compass

The Compass bean is represented by the `IlvJCompass` class. This bean displays a compass indicating the geographic or cartographic north of the map.

An example of the Compass is shown in *Multithread Monitor* .



Compass

Including the bean in an application

To include the Compass bean in your application, write the following lines of code:

```
IlvJCompass compass = new IlvJCompass();
compass.setView(view);
```

The Compass bean then listens to view transformation changes, and check the rotation value to display north direction.

Customizing the appearance

You can customize many parts displayed on the compass such as:

♦ The appearance of the compass needle:

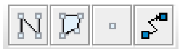
```
compass.setCartographicNeedleStyle(IlvCompass.NEEDLE_STYLE_CROSS);
```

♦ The colors used:

```
compass.setCartographicBackground(Color.blue);
compass.setCartographicForeground(Color.gray);
```

Using annotations

The Annotations Toolbar bean is represented by the `IlvMapAnnotationToolBar` class. You can annotate maps using predefined annotations created by the Annotations Toolbar. The toolbar can be used interactively to add a point, polyline, or polygon annotation. An example of the Annotations Toolbar is shown in *Annotations toolbar*.



Annotations toolbar

An annotation is a drawing made on the top of a map to describe or provide additional information about a specific zone of a map. Annotations are labeled and are projected with respect to the coordinate system of the map, which is stored in the manager as an `IlvCoordinateSystemProperty`. In JViews Maps, annotations are dedicated `IlvGraphic` objects. Labels can be displayed to provide text information and are labeled using the JViews Maps labeling mechanism.

The Annotations Toolbar bean is an extension of `JToolBar`.

To include the bean in an application:

1. To include the Annotations Toolbar bean in your application, write the following lines of code:

   ```
   IlvMapAnnotationToolBar annotations =new IlvMapAnnotationToolBar();
   annotations.setView(view);
   ```

   The toolbar is ready for interactive creation of annotation objects. You can also add annotations to a map using the API. The objects used to display annotations are specialized `IlvGraphic`s.

2. You can customize certain properties of the Annotations Toolbar using the following code:

   ```
   // Set the size of the buttons.
   annotations.setButtonSize(new Dimension(25, 25));
   // Prevent the toolbar being dragged elsewhere.
   annotations.setFloatable(false);
   ```

3. If you want your annotations to be managed as nodes, in order to make it possible for you to create link annotations, you need to indicate it through:

   ```
   annotations.setGrapherMode(true);
   ```

4. Create the annotation as an `IlvMapAnnotationToolBar.MapMarker`:

5.
   ```
       IlvPoint p = new IlvPoint(10, 50);
       IlvMapAnnotationToolBar.MapMarker m = new
          IlvMapAnnotationToolBar.MapMarker(p);
   ```

These graphic objects are stored in an `IlvGraphicLayerDataSource`. The `IlvMapAnnotationModel` class can provide such a data source.

6. Get the `IlvGraphicLayerDataSource` data source:

```
    IlvMapAnnotationModel model =
       IlvMapAnnotationProperty.GetMapAnnotationModel(manager);
    IlvGraphicLayerDataSource dataSource = model.getDataSource(manager,
       "TEST");
    String name = "TEST" + " Annotation";
    dataSource.getInsertionLayer().setName(name);
    dataSource.add(m,
       IlvCoordinateSystemProperty.GetCoordinateSystem(manager));
```

7. Set the style for the layer in which the annotation is to be inserted. The following piece of code also sets the `label` attribute so that the annotation is labeled accordingly:

```
    if (dataSource.getInsertionLayer().getStyle() == null) {
      IlvMapStyle style = new IlvPointStyle();
      style.setAttributeInfo(IlvMapAnnotationModel.info);
     style.setLabelAttribute(IlvMapAnnotationModel.info.getAttributeName
(0));
      dataSource.getInsertionLayer().setStyle(style);
    }
    IlvPointStyle ps =
       (IlvPointStyle)dataSource.getInsertionLayer().getStyle();
    m.setStyle(ps);
    ps.setSize(5);
    ps.setType(IlvMarker.IlvMarkerFilledDiamond);
    ps.setForeground(Color.pink);
```

8. Attach the `feature` attribute property; the string `A Label` will be displayed as the annotation label:

```
    String s = "A Label";
    IlvFeatureAttributeProperty properties = new
       IlvFeatureAttributeProperty(IlvMapAnnotationModel.info,
          new IlvFeatureAttribute[] { new IlvStringAttribute(s)});
    m.setNamedProperty(properties);
```

9. Start the data source to add the annotation to the manager. The labeler may also need to be started:

```
    try {

    manager.setInsertionLayer(dataSource.getInsertionLayer().
getManagerLayer().
       getIndex());
      dataSource.start();
    } catch (Exception e) {
      e.printStackTrace();
    }
    IlvMapLabeler labeler = IlvMapLabelerProperty.GetMapLabeler(manager)
;
    labeler.setView(view);
    labeler.addLayer(dataSource.getInsertionLayer());
    labeler.performLabeling();
```

The Symbology Tree View bean

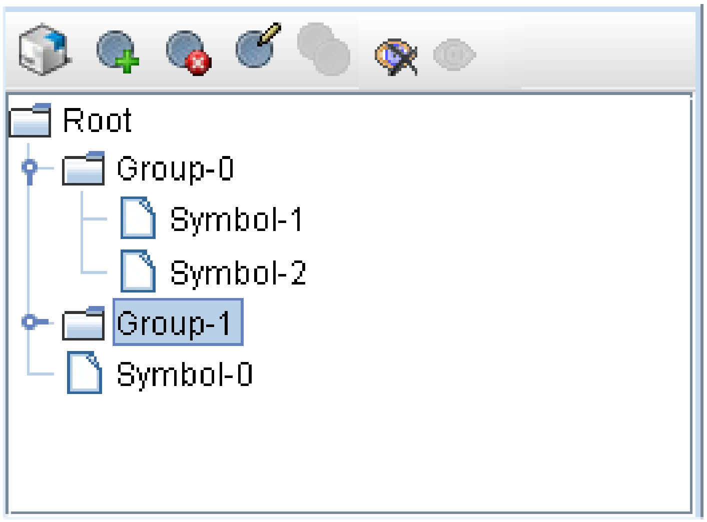
Describes the Symbology Tree View bean and how to use it.

In this section

Overview

The Symbology Tree View bean is represented by the `IlvSymbologyTreeView` class. This bean displays the symbol hierarchy of the current map.

The following figure shows an example of the symbol hierarchy.



Symbology Tree View

Adding the bean to an application

To include the Symbology Tree View bean in your application:

1. Create an `IlvSDMEngine`:

```
IlvSDMEngine engine = new IlvSDMEngine();
engine.setGrapher((IlvGrapher)view.getManager());
engine.setReferenceView(view);
```

2. Provide the Cascading Style Sheet for the SDM engine to use:

```
engine.setStyleSheets(new String[]{"myfile.css"});
```

3. Create a Symbology Tree View for the SDM engine:

```
IlvSymbologyTreeView symbPanel = new IlvSymbologyTreeView(engine);
```

4. Add this bean to your Swing hierarchy:

```
panel.add(symbPanel, BorderLayout.SOUTH);
```

Symbology panel actions

You can change the structure of your symbol model using the Symbology Tree View toolbar or a shortcut menu. You can add and remove symbols or groups, but also edit the symbols, groups or the symbology itself. To do this you must implement the `IlvSymbologyTreeViewActions` interface and set it on the Symbology Panel:

```
symbPanel.setSymbologyTreeViewActions(new MySymbologyTreeViewActions());
```

The action class should provide the methods invoked by the Symbology Tree View User Interface in order to modify the symbology engine according to the requirements of your application. It should also provide any User Interface specific to editing symbols and groups.

The action classes are also responsible for managing symbol model persistence, symbol drag and drop and location picking on the map.

For examples of custom tree view actions see ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/src/utils/sdm/CombinedTreeViewActions.java`**.

Making the model persistent

Making a symbol model persistent means providing a way to embed SDM model information inside a map file.

This can be done through the `IlvPersistentSDMNodeFactory` interface and the `IlvPersistentSDMModelProperty` manager property.

You should not use this feature if you intend to use JViews Diagrammer Designer with your map, as it stores and sets up symbol model differently, in separated xml files. You should use an `IlvMapStyleSheetRenderer` if you want to take advantage of this feature. If you do not, the manager layers that will be used to place your symbols may not be instances of the `IlvMapStyleSheetRenderer.IlvMapSymbolManagerLayer` class, which means that the graphical representation of the symbols will be saved separately in the ivl file, causing its duplication when you read the map back.

The persistent SDM model property

This class is a named property that will be attached to a manager, and trigger the saving and reading of an engine model inside an IVL file.

Example of use:

```
IlvPersistentSDMModelProperty property =
IlvPersistentSDMModelProperty.GetPersistentSDMModel
(engine,factory,listenToChanges);
```

This call will register the property on the engine's manager. It will be saved (like every named property) when the manager gets saved, and will invoke an internal `IlvXMLConnector` that will save the XML content of the engine model as a string.

At read time, this property requires the use a node factory to create SDM nodes in the engine model, detailed below.

If you use this call with `listenToChanges` set to true, you can register this property for an engine before reading the map IVL file. The listener attached will ensure that the factory and engine used when reading the property get replaced by the ones you specify before the data is read.

If you do not do this, internal XML data will be read, and you will have to attach the engine manually later.

The persistent SDM Node factory

The principal purpose of the `IlvPersistentSDMNodeFactory` interface is to provide a way for model nodes to be created when you read the model data back in. Usually creating a symbol is dependent on some application so this interface also includes methods to manage that persistent context.

The following provides a simple code example which can help you to write your own factory.

```
public class SampleFactory implements IlvPersistentSDMNodeFactory {
  static public class SampleContext extends IlvPersistentObject {
 // some implementation of a persistent object ...
  }
  IlvPersistentObject _factoryPersistentContext;

  public IlvPersistentObject getPersistentContext() {
    // the persistent context should be created or updated at this point.
    _factoryPersistentContext=new SampleContext();
    _factoryPersistentContext.setSomeData(this.getSomeData());
    return _factoryPersistentContext;
  }

  public boolean isPropertyIgnored(IlvSDMModel model, Object node, String name)
 {
   // ignore the "data" property
   if("data".equals(name)) return true;
   return false;
  }

  public Object newSymbol(String tag) {
   // we will create default SDM nodes
   IlvDefaultSDMNode node=new IlvDefaultSDMNode(tag);
   // configure the node, for example
   node.setProperty("data", this.getSomeData());
   return node;
  }

  public void setPersistentContext(IlvPersistentObject context) {
   // the local information should be updated with what was stored in the
context
   this.setSomeData(context.getSomeData());
  }
```

For other examples of custom `IlvPersistentSDMNodeFactory` see ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/src/utils/sdm/CombinedTreeViewActions.java`** and `IlvPersistentSDMNodeFactory` see ***<installdir>*** **`/jviews-maps86/samples/mapbuilder/src/utils/sdm/PaletteSymbologyTreeViewActions.java`**.

Handling map features

A map feature is an object that represents cartographic data as it was read from its source file. A map feature holds three main information fields: Its geometry, the coordinate system in which its geometry is expressed, and its attributes. If the map feature is a town, for example, its attributes can be its name and the number of inhabitants. A map feature is completely independent of the way it will be graphically represented in the application. Thus, a point marking the summit of a hill might very well be represented with graphic objects as diverse as a cross, a circle, or an icon.

A map feature carries the following information:

♦ *Map Feature Geometry*

♦ *Map Feature Attributes*

♦ The coordinate system in which the geometry is expressed. For details on coordinate systems, see *Handling spatial reference systems*.

Map Feature Geometry

Each map feature has a geometry. The geometry of a map feature is information relating to its shape and position.

In JViews Maps, map feature geometries are defined by the `IlvMapGeometry` class in the `ilog.views.maps` package. The package `ilog.views.maps.geometry` supplies a number of predefined geometries which are modeled on the “Simple Map Features” geometry specifications defined by the OpenGIS Consortium to insure interoperability between Geographic Information Systems (GIS). Note, however, that the classes in this package are not strictly equivalent to this model in terms of functionality. They provide simplified features and are mainly drawing oriented. Nevertheless, using these classes greatly facilitates the conversion of data coming from a map server, such as Oracle Spatial, for example.

This package also contains additional geometries for handling images, rasters, and text more easily and can be extended with new geometries.

Map Feature Attributes

Each map feature can also have attributes. If the map feature is a town, its attributes can be its name, or the number of inhabitants. Attributes can be used, for example, for graphical rendering. In the section *Creating a colored line renderer*, the color of polylines representing contour lines on a map is defined by the elevation attribute.

Attributes belong to the class `IlvFeatureAttribute`. They are stored in the following two classes of the `ilog.views.maps` package:

♦ `IlvAttributeInfoProperty`, which defines the attribute properties, such as name, type, mandatory, or optional characters.

♦ `IlvFeatureAttributeProperty`, which contains the values of these attributes.

The following code example lists the attributes of an `IlvMapFeature` object and displays them on the screen:

```
public void dumpAttributes(IlvMapFeature feature)
 {
   IlvAttributeInfoProperty info = feature.getAttributeInfo();
   IlvFeatureAttributeProperty attributes = feature.getAttributes();

   // Attributes are not mandatory in a feature.
   if ((info == null) || (attributes == null)) {
     System.out.println("This feature has no attribute");
     return;
   }

   for (int i = 0; i < info.getAttributesCount(); i++) {
     String name = info.getAttributeName(i);
     IlvFeatureAttribute attribute = attributes.getAttribute(i);
     System.out.println("The attribute " + name +
                        " takes the value " + attribute.toString());
   }
 }
```

The attributes are of different types, according to whether they represent whole numbers, floating-point values, character strings, and so on. The predefined attributes, all of the `IlvFeatureAttribute` class, are in the `ilog.views.maps.attribute` package.

Using load-on-demand

Describes load-on-demand and how to use it.

In this section

Saving a tiled layer
Describes layer parameters and the impact on saving tiles.

Writing a new cache algorithm
Describes how to write a custom cache algorithm.

Writing a tile loader for a custom data source
Describes how to write a tile loader for load-on-demand.

Load-on-demand for hierarchical data sources
Describes how to write a custom data source with load-on-demand facilities.

Load-on-demand

JViews Maps offers a mechanism that lets you load into memory only the data that you want to display in a manager view. This mechanism, known as load-on-demand, is extremely valuable, especially when very large maps are concerned. Consider a database storing maps of the whole world with a scale of 1/25,000. If these maps were scanned with a resolution of 300 DPI, the required storage space would be as follows:

- 654 kilobytes for 1 square kilometer
- 64 megabytes for 100 square kilometers
- Approximately 310 terabytes for the whole world

Given the volume of this data, it is crucial to have a load-on-demand mechanism that will load and display only the portion of a map of direct interest.

The IlvTiledLayer Class

The `IlvTiledLayerIlvTi` extends the `IlvManagerLayer` class and allows the loading of large sets of data, for example, large maps. It is divided into a number of rectangular areas called tiles (organized into grid cells or into a list of specified areas) and provides a notification mechanism to load only the graphic objects that are required by the application, because a tile is visible in one of the views of the manager or because it has been explicitly required by the application.

An `IlvTiledLayer` is associated with three important objects:

- An `IlvTileController`, which manages the tile events.
- An `IlvTileLoader`, which loads the tiles from a data source.
- An `IlvTileCache`, which manages the cache and deletes the unused tiles.

When an `IlvTiledLayer` is saved into an `.ivl` file, it does not save the graphic objects it contains. It saves its tiling parameters (loader, cache, tile controller, tiling structure).

The IlvTileController Class

The `IlvTileController` class manages the load-on-demand mechanism. It divides the space into a number of rectangular areas called tiles, in two different fashions according to its mode:

- In indexed mode, tiles are distributed on a regular grid defined by its origin rectangle. A tile is then referenced by its line and column indices. All tiles have the same height and width, and are automatically created as needed by the `IlvTileController`, according to the visible parts of the tiled layer in the manager views.
- In free mode, however, the `IlvTileController` holds a list of `IlvFreeTile` instances that need to be created and then added to the tile controller. These tiles can have different sizes and locations, and can also overlap.

The tiling mode for an `IlvTileController` is set at creation time and can not be modified afterwards.

Each time a tile becomes visible in a view of the manager to which it is attached, the tile controller notifies a loader (called tile loader) to load the data attached to this tile. A cache releases the invisible tiles (called the cached tiles) when it is necessary to free memory.

The IlvTileLoader Interface

The `IlvTileLoader` interface is used to load a tile for an `IlvTileController` or an `IlvTiledLayer`.

The following example shows a tile loader that fills a tile with a generated list of graphic objects.

The file is: ***<installdir>*** **`/jviews-maps86/codefragments/srs/src/Sample2.java`**

```
class SimpleTileLoader
  implements IlvTileLoader
{
  public void load(IlvTile tile)
  {
    IlvRect rect = new IlvRect();
    tile.boundingBox(rect);
    IlvPoint p = new IlvPoint();
    p.x = rect.x;
    for (int i = 0; i < 10; i++) {
      p.y = rect.y;
      for (int j = 0; j < 10; j++) {
        tile.addObject(new IlvMarker(p, IlvMarker.IlvMarkerPlus),
                       null);
        p.y += rect.height / 10;
      }
      p.x += rect.width / 10;
    }
    tile.loadComplete();
  }
  public void release(IlvTile tile)
  {
    tile.deleteAll();
  }

  public boolean isPersistent()
  {
    return false;
  }

  public void write(IlvOutputStream stream)
  {
    // do nothing
  }
}
```

The IlvTileCache Class

The `IlvTileCache` class is used to manage the cached tiles of one or more tile controllers. A cached tile is a tile that is not visible in any view and that is not locked by any application object. An `IlvTileCache` object releases the cached tiles to free memory when necessary.

The IlvTile Class

A tile represents an elementary rectangular area that is loaded or released when needed by the application. Tiles are managed by an `IlvTileController`.

In most cases, the tile controller will be associated with an `IlvTiledLayer`. This means that the tiles represent areas that must be filled with graphic objects when they become visible due to the user of the application scrolling or zooming on a view.

More sophisticated applications can use a tile controller without attaching it to a layer, for example, to load data that is needed to process an area that becomes visible.

For example, to define a geographic coordinate system expressing latitude and longitude with grads, instead of degrees, you can use the following code:

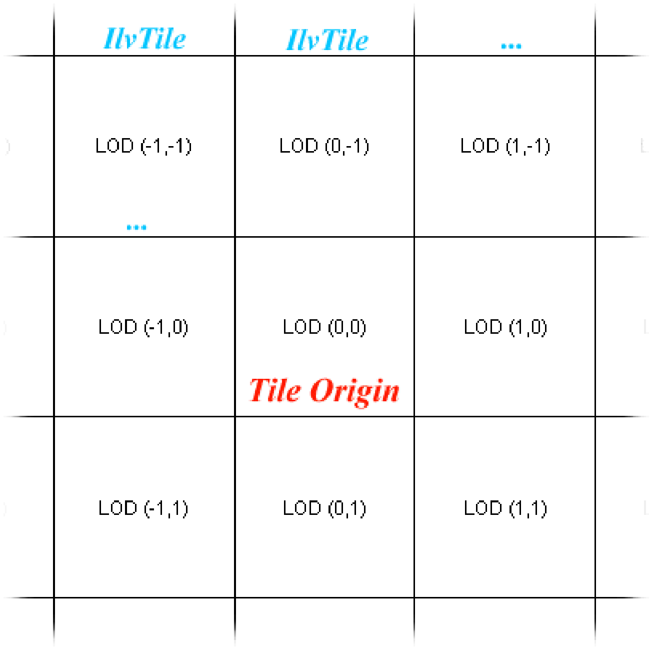
```
IlvAngularUnit unit = IlvAngularUnit.GRAD;
IlvGeographicCoordinateSystem gcs =
   new IlvGeographicCoordinateSystem("My coordsys",
                                      IlvHorizontalShiftDatum.WGS84,
                              IlvMeridian.GREENWICH,
                              unit,
                              null); // No altitude

// A transformation from WGS84 geographic coordinate system
// with degrees as unit to our own coordinate system:
IlvCoordinateTransformation ct =
   IlvCoordinateTransformation.CreateTransformation(
      IlvGeographicCoordinateSystem.WGS84,
      gcs);

// Example of conversion.
double lambda = IlvAngularUnit.DEGREE.toRadians(-45D);
double phi =    IlvAngularUnit.DEGREE.toRadians(30D);
IlvCoordinate coord = new IlvCoordinate(lambda,phi);

// Convert coordinate, letting the transformation allocate the
// result.
IlvCoordinate result = ct.transform(coord,null);

System.out.println("The expression of point 45W 30N is ");
System.out.println("x = " + coord.x + " grad");
System.out.println("y = " + coord.y + " grad");
```

The IlvFreeTile Class

This class is similar to the `IlvTile`, except that instead of being part of a regular grid (and therefore indexed by line and column), its bounds can be freely specified. This is particularly useful when overlapping areas are needed, for example, in case of a projected map. Note that although `IlvFreeTile` is a subclass of `IlvTile` (for compatibility reasons), the member variables row and column are not relevant in terms of tile location. They are used to locate a superclass `IlvTile` instance on a grid, but given that `IlvFreeTile` objects have bounds of their own, there is no need for line and column properties.

Structure of the tiling grid (indexed mode only)

A tiled layer is a type of manager layer specifically designed to support load-on-demand. If the `IlvTileController` associated with this layer works in indexed mode, the layer is divided into a set of rectangular tiles of identical size that form a tiling grid (note that a constructor of `IlvTiledLayer` takes a mode as a parameter).



Tiling grid in indexed mode

The tiling grid is defined by its origin tile that is located at the intersection of the row and column of index 0, see *Tiling grid in indexed mode*.

The other tiles in the grid are identified by their column and row number, starting from the origin tile. The following code example displays the status of the tile that is at the intersection of column 10 and row 5:

```
public static void displayTileStatus(IlvTiledLayer layer) {
  IlvTile tile = layer.getTileController().getTile(10, 5);
  if(tile == null)
    System.out.println("The tile is not loaded yet");
  else {
    int status = tile.getStatus();
    if(status == IlvTile.LOCKED)
      System.out.println("The tile is locked");
    else if(status == IlvTile.CACHED)
      System.out.println("The tile is cached");
    else
```

```
        System.out.println("The tile is empty");
    }
  }
```

You can see in the above code example that the `getTile(int, int)` method can sometimes return a null value. Because the potential number of tiles can be very great—the number of tiles is even virtually infinite—the `IlvTile` objects are allocated only if the tile is loaded or is in the cache.

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/lod/src/TileStatus.java`**.

Size of the tiling grid in indexed mode

You can set the size of the tiling grid using the following method:

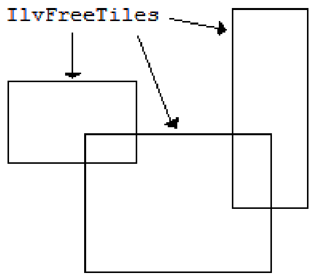
```
setSize(ilog.views.IlvRect)
```

The rectangle that delimits the tiling grid is expressed in the manager coordinates. Only the tiles that intersect with this rectangle can be loaded. You can see an example of a tiling grid whose size has been defined in the debug view illustrated in *Load-on-demand debug view*.

Note: The tiling parameters introduced in this section (size of the tiles, origin tile, and size of the tiling grid) can be configured for each tiled layer in a manager. This allows you to have in the same manager large-tile layers containing objects displayed at a small scale and small-tile layers containing objects displayed at a large scale.

Structure and size of the tiled layer (free mode only)

When the `IlvTiledLayer` is set to free mode (or its associated `IlvTileController`), the structure is not a grid (unlike indexed mode described in *Structure of the tiling grid (indexed mode only)*), but a list of rectangular areas that must be added to the `IlvTileController`.



Tiled layer in free mode

These areas can have any bounds and can even overlap.

The size of the tiled layer is simply the union of all tile bounds.

Displaying the state of tiles

You can create a debug view to display the state of the tiles in a tiled layer using the following method:

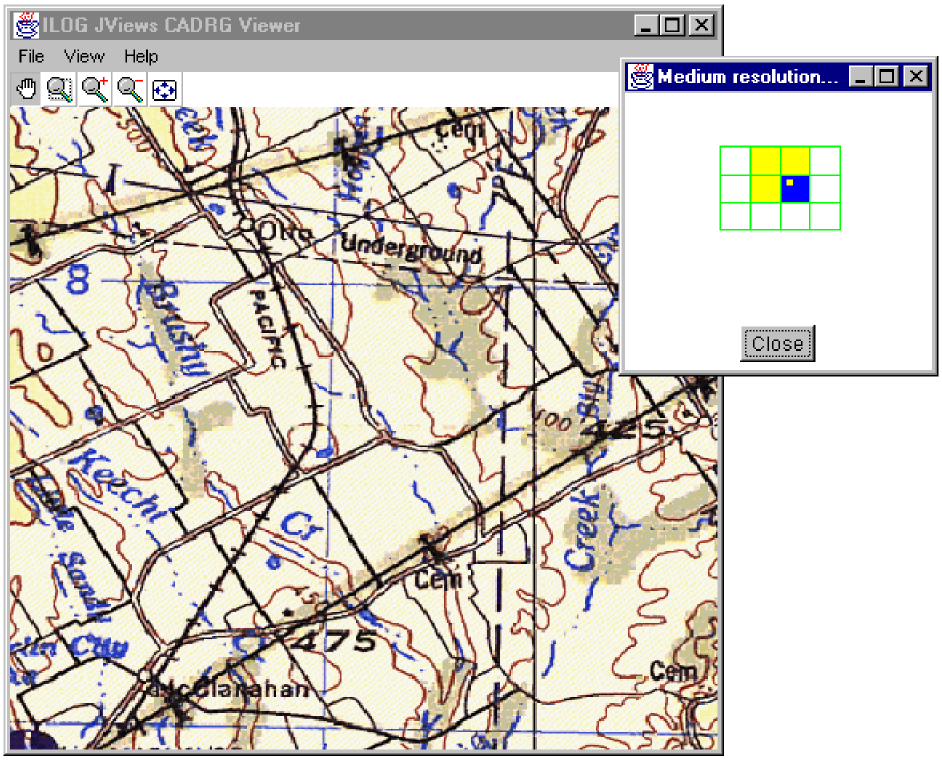
```
setDebugView(ilog.views.IlvManagerView)
```

As its name suggests, this debug view is particularly useful for debugging operations when implementing load-on-demand for a new cartographic format.

A tile can have three different states. It can be:

- ♦ Empty, meaning that its objects are not loaded into memory.
- ♦ Loaded, meaning that its objects are loaded into memory and visible.
- ♦ Cached, meaning that its objects are loaded into memory but not visible.

The debug view is of the `IlvManagerView` type, and must be attached to a tiled layer. The debug view does not increment nor decrement the tile lock counters. This is the role of the tile controller. It just displays the tiles in color according to their state, as in *Load-on-demand debug view*.



Load-on-demand debug view

The tiles whose lock counter is greater than 1 appear in blue. They are visible in at least one view. The tiles whose counter equals to 0 appear in yellow. They are cached. The white tiles are not loaded.

In the previous example, an `IlvManagerMagViewInteractor` was associated with the debug view. It is this interactor that displays a little yellow square inside the blue tile. The square shows the zone displayed by the main view.

The following is an example of code that creates a debug view for a layer.

```
public static void createDebugView(IlvTiledLayer layer) {
  IlvManagerView view = new IlvManagerView(layer.getManager());
  layer.setDebugView(view);

  // Create a swing frame to display this debug view.
  JFrame frame = new JFrame("Debug View");
  frame.setLayout(new BorderLayout());
  frame.add(BorderLayout.CENTER, view);
  view.setAutoFitToContents(true);
  frame.setSize(200, 200);
  frame.setVisible(true);
}
```

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/lod/src/DebugView.java`**

Controlling load-on-demand

Using visibility filters to control load-on-demand

If you do not want to use the dynamic styles provided by the `IlvMapStyleController` class, you can use scale visibility filters to control the display of tiles in a manager layer.

When a scale visibility filter has been set for a tiled layer, this layer will be displayed if the scale factor set for its manager view is within specified scale limits. Otherwise, it will be hidden. When a tiled layer is visible because the zoom factor of the manager view allows it, visible tiles are automatically locked and thus loaded into memory. In the same way, when a tiled layer is hidden because the zoom level of its view exceeds a certain value, all the locks set on visible tiles are released and the tiles are removed.

Scale visibility filters are generally used to activate load-on-demand for zoom factors between a minimum and a maximum scale value. Let us consider a map scanned with a scale of 1/1,000,000, you can set visibility filters so that its layer is visible for scale factors ranging from 1/2,500,000 to 1/500,00.

Setting tile locking filters

You can use the `IlvTileLockFilter` class to dynamically prevent a tile from being loaded when it becomes visible in a view. To do so, you associate an `IlvTileLockFilter` object with a tile controller using the following method:

```
setLockFilter(ilog.views.tiling.IlvTileLockFilter)
```

If the tile controller possesses a lock filter, it calls the method `isLockAllowed(ilog.views.tiling.IlvTileController, ilog.views.IlvManagerView)` of the lock filter each time a tile becomes visible in a view. If this method returns `false`, the counter of the tile is not incremented.

Using a lock filter, you can prevent the loading of data displayed at a small scale (when a map is zoomed out greatly, for example) while leaving the tiles that were already loaded visible.

Loading tiles using the API

The load-on-demand mechanism is event-driven in that cartographic data is loaded or unloaded following user's actions, such as zooming or panning. You can, however, use the `lockTile()` method for example—to preload a tile corresponding to a map zone that is visited frequently or to prevent the tile from being unloaded.

The `lockTile()` method shown below increments the tile lock counter:

```
lockTile(int, int, java.lang.Object)
```

If the tile lock counter is equal to zero, the tile is loaded into memory and will not be unloaded as long as the lock is not released—with the `IlvTileController.unlockTile` method, for example.

Updating all the visible tiles in a view

Sometimes you may need to update immediately all the tiles that are currently viewed by an `IlvManagerView`. You can then use the `updateView(ilog.views.IlvManagerView)` method. It can be useful when you reactivate a view that was ignored until then by the load-on-demand mechanism.

Managing errors and load-on-demand events

The load-on-demand mechanism might generate errors when the map cannot be entirely loaded due to memory problems, absence of data, loss of connection to a server and so on. Graphic applications must catch these errors to inform the user that the map being viewed is not complete.

To be notified of these events, and any other events related to load-on-demand, you can set a `TileListener` to an instance of the `IlvTileController` class. This listener is notified of all changes to the tiles, for example, the beginning or end of loading, moving tiles into the cache, or out of the cache for reuse, loading errors, and so on.

The following example displays all the events related to load-on-demand:

```
lodLayer.getTileController().addTileListener(new TileListener() {
  public void tileChanged(TileEvent e) {
    System.err.print(e);
  }
});
```

By default, the tile controller displays a message on the screen if an error occurs while a tile is being loaded. You can deactivate this behavior using the following method:

`setPrintingErrors(boolean)`

The different types of events sent to the tile listeners are the following:

- `TILE_ABOUT_TO_LOAD` Triggered when a tile is about to be loaded into memory.
- `TILE_LOADED` Triggered when a tile has been loaded into memory.
- `TILE_CACHED` Triggered when a tile that is no longer being used is stored in the cache.
- `TILE_RETRIEVED` Triggered when the counter of a tile stored in the cache is incremented again. The tile can no longer be unloaded.
- `TILE_RELEASED` Triggered when a tile is completely freed.
- `ERROR` Triggered when an error occurs while a tile is being processed. In this case, the `getThrowable()` method of `TileEvent` allows you to know the exception or the error that caused the problem.
- `CONTROLLER_DISPOSED` Triggered when the tile controller is deactivated. For example, if you remove its associated tiled layer from a manager.
- `NO_CHANGE` Triggered when the last event in a series of events is processed.

When an event in a view causes an action to be performed on a tile, the tile controller notifies the tile listener of the action. If this event provokes a series of transitional events, these are transmitted to the listener as a group. Therefore, modifying a scale factor can cause new tiles to be loaded and other tiles to be cached. Grouping events allows an action to be performed only when all the transitional events it causes are completed. The `isAdjusting()` method of the class `TileEvent` returns `true` if the event is part of a series of events. The `NO_CHANGE` event is sent once the last event in a series is processed and the `isAdjustmentEnd()` method returns `true`.

Caching tiles

The tile cache is the place where tiles whose lock counter has returned to 0 are stored. The tiles in the cache are eligible for unloading if memory is needed for loading new tiles.

The cache can be shared among several layers, which means that loading a tile in a layer can cause that tile to be unloaded in another layer.

The `IlvDefaultTileCache` class implements a cache algorithm consisting of a simple LRU (Least Recently Used) structure that unloads first the tiles that have been visited the least recently.

You can, however, implement another algorithm that will be more efficient with respect to the specific nature of your application. Here are a few criteria you might take into account when implementing a new backup cache algorithm.

- Unload first the tiles that require a larger number of pan or zoom operations to be reached from the current position.
- Unload first the tiles that have taken the longest to load.
- Unload first the tiles that contain the largest number of graphic objects.

An example of a simplified cache algorithm is given in *Writing a new cache algorithm*.

Saving a tiled layer

A layer has a number of associated parameters. Some parameters such as named properties, visibility filters, and names are common to all layers, whether they are tiled layers or just normal layers. Other parameters are specific to tiled layers. These are the tiling parameters, the lock filter and the tile loader, introduced in the previous sections.

Unlike normal layers, when you save a tiled layer to an `.ivl` file, only its attached parameters are saved with the layer, not the objects it holds.

Writing a new cache algorithm

This section explains how to write a custom cache algorithm to meet specific application requirements. The example in this section is a simplified version of the class `IlvDefaultTileCache` provided in the JViews Maps library.

The complete source code for this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/lod/src/SimpleTileCache.java`**

The class `SimpleTileCache` extends the class `IlvTileCache`.

```
public class SimpleTileCache
  extends IlvTileCache
{
```

`cacheSize` defines the maximum number of tiles that can be stored in the cache.

```
  // default cache size
  private int cacheSize = 5;
  // To Store the tiles
  transient private Vector tiles = new Vector();
```

The following constructor creates an instance of the default cache with the specified size (in this example, 5).

```
 public SimpleTileCache(int size)
  {
    cacheSize = size;
  }
```

The following constructor reads the cache from the provided input stream. Caches created using this constructor implement the `IlvPersistentObject` interface and can thus be saved with an `IlvTiledLayer` object.

```
  public SimpleTileCache(IlvInputStream stream)
  throws IlvReadFileException
  {
    cacheSize = stream.readInt("size");
  }
```

The `write()` method writes the cache to an output stream.

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

The following method belongs to the `IlvTileCache` interface. It is called when a tile is cached. In this implementation, the tile is added at the end of the internal tile list.

```
  public void tileCached(IlvTile tile)
  {
    tiles.addElement(tile);
  }
```

The following method belongs to the `IlvTileCache` interface. It is called when a tile is removed from the cache and locked again. With this implementation, the tile is removed from the internal tile list.

```
  public void tileRetrieved(IlvTile tile)
  {
    tiles.removeElement(tile);
  }
```

The following method belongs to the `IlvTileCache` interface. It is called when a tile is about to be loaded. With this implementation, if the number of tiles in the cache exceeds the cache size, the least recently used tiles, at the top the internal tile list, will be unloaded to make room for new tiles.

```
  public void tileAboutToLoad(IlvTile tile)
  {
    int toRemove = tiles.size() - cacheSize;
    if (toRemove <= 0)
      return;
    for (int i = toRemove; i > 0; i--) {
      IlvTile current = (IlvTile) tiles.elementAt(0);
      tiles.removeElementAt(0);
      releaseTile(current);
    }
  }
```

The following method belongs to the `IlvTileCache` interface. It is called when a tiled layer is taken out of the manager to remove the tiles managed by its tile controller from the cache.

```
  public void controllerDisposed(IlvTileController controller)
  {
    int i = 0;
    while (i < tiles.size()) {
      IlvTile tile = (IlvTile) tiles.elementAt(i);
      if (tile.getController() == controller)
        tiles.removeElementAt(i);
      else
        i++;
        }
    }
```

Writing a tile loader for a custom data source

To implement load-on-demand for a new data source, all you have to do is write a specific tile loader that implements the `IlvTileLoader` or the `IlvMapTileLoader` interface. Notice, however, that for the predefined map formats supplied with JViews Maps, load-on-demand has been implemented in a subclass of `IlvTiledLayer` that defines both the tile loader (as a private class) and the tiling parameters appropriate for the concerned format. The `IlvMapTileLoader` class and the predefined formats are described in *Readers and writers*.

For your tile loader to be fully efficient, the following requirements should be satisfied:

- ♦ You should be able to determine which objects are to be loaded on the tile. These objects can be read from a file whose name is known, or be the result of a query to a cartographic database.
- ♦ You should be able to have a direct random access to data.
- ♦ The size of the data to be loaded should be in proportion to the size of the tiles to allow fast loading. For example, raster images with a size of 100x100 are faster to load than images with a size of 6000x6000.

The following example of a tile loader simulates the loading of two graphic objects, a rectangle and a label.

Its `load()` method takes the tile to be loaded as its parameter. It generates the graphic objects to be displayed within the tile and adds them to the tiled layer by calling the `tile.addObject()` method. When loading is complete, it calls the `tile.loadComplete` method to notify the listeners that the data in the tile is ready for use.

```
public class LodLoader implements IlvTileLoader
{
  [...]

  public void load(IlvTile tile) throws Exception
  {
    IlvRect bbox = new IlvRect();
    tile.boundingBox(bbox);

    // Add a rectangle inside the tile bounding box
    tile.addObject(new IlvRectangle(bbox),null);

    // Add text
    String text = this.layerName +
      " (" + tile.getColumn() + "," + tile.getRow() + ")";
    IlvPoint textCenter = new IlvPoint(bbox.x + bbox.width / 2,
                                       bbox.y + bbox.height / 2);
    tile.addObject(new IlvLabel(textCenter,text),null);

    tile.loadComplete();
  }
   [...]
```

Its `release()` method is invoked when the tile cache releases a tile. The `tile.deleteAll` method clears the tile.

```
/**
 * Releases the objects on this tile.
 */
public void release(IlvTile tile)
{
  tile.deleteAll();
}
```

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/lod/src/LodLoader.java`**

Load-on-demand for hierarchical data sources

To write a new IlvMapDataSource for a format of your own using the load-on-demand mechanism, you should inherit from the IlvTilableDataSource class. This class performs all of the steps required to create the appropriate tiles, and to configure tiled layers to make efficient use of load-on-demand. You can choose the number of rows and columns into which to divide the map, or even deactivate the tiling mechanism if you prefer. However, you need to implement two abstract methods:

```
protected abstract void createTiledLayers();
protected IlvMapRegionOfInterestIterator createTiledIterator(IlvMapLayer
layer);
```

The source code for the Map Builder demonstration, which contains all of the code described in this section, can be found at ***<installdir>*** **/jviews-maps86/samples/mapbuilder/index.html**

Creating tiled layers

The createTiledLayers() method is called by the start() method of the IlvMapDataSource. Its role is to create the tiled layers that make up this data source. For example, if your data source sorts map features into different layers, all of the tiled layers must be created by this method.

Here is a sample implementation of this method:

```
protected void createTiledLayers() {
  // get the insertion layer of this datasource (i.e. the ‘root’ layer
  // of this data source, potentially containing children layers)
  IlvMapLayer mapLayer = getInsertionLayer();

  // Create a tiled layer to associated with this IlvMapLayer (refer to the
chapter
  // on tiled layers)
  IlvTiledLayer currentTiledLayer = new IlvTiledLayer(new IlvRect(),new
  IlvDefaultTileCache(),IlvTileController.FREE);

  // associate this layer with the IlvMapLayer
  mapLayer.insert(currentTiledLayer);
}
```

Reading map features

The createTiledIterator(ilog.views.maps.beans.IlvMapLayer) method returns a feature iterator over map features located in a specified region of interest. The tile loaders of the layers use this iterator to load the tiles when they become available on screen. That is, they read the map features of the tiles in the region of interest only.

Here is a basic implementation of this method:

```
protected IlvMapRegionOfInterestIterator createTiledIterator(IlvMapLayer layer)

throws IOException {
  // IlvMapLayer mapLayer = getInsertionLayer();
  if (layer != mapLayer) {
    // the layer in parameter has nothing to do with this data source
    return null;
  }
  // Create your region of interest feature iterator here,
  // basically an iterator over map features in specified region
  MyRegionOfInterestIterator iterator = new MyRegionOfInterestIterator();

  // configure it as needed (we assume that ther is a 'configure' method here

for
  // clarity purposes)
  iterator.configure();

  // return it
  return iterator;
}
```

If multiple tiled layers are created by the `createTiledLayers()` method in *Creating tiled layers*, a different iterator might be returned for each layer (hence the `layer` parameter in the `createTiledIterator()` method), see *Data tiling*.

Manipulating renderers

Describes the attributes of graphic objects that will be displayed and the use of renderers to transform map features to graphic objects.

In this section

Rendering with a geodetic computation

Describes renderers that offer geodetic computations.

Overview

Attaching attributes to graphic objects

In JViews Maps, you can attach properties to `IlvGraphic` objects using the class `IlvNamedProperty` of the `ilog.views` package, thereby saving the properties in an `.ivl` file together with the related object.

The `IlvFeatureAttributeProperty` class, which stores all the attributes of a map feature, inherits from the `IlvNamedProperty` class and can therefore be attached to any graphic object.

The following code example attaches an `IlvFeatureAttributeProperty` object to an object of the `IlvGraphic` class:

```
IlvFeatureAttributeProperty attributes = feature.getAttributes();
graphic.setNamedProperty(attributes.copy());
```

Note that in this example, a copy of the attribute property is made. The reason for this is that map features, along with their geometry and attributes, are volatile and get lost when another map feature is read. For more information about map feature volatility, see *The IlvMapFeatureIterator interface*.

To access the attributes that have been attached to a graphic object, you can use the following code:

```
IlvFeatureAttributeProperty attributes = (IlvFeatureAttributeProperty)
  graphic.getNamedProperty(IlvFeatureAttributeProperty.NAME);
```

To save information specific to an application that cannot be saved using the predefined named properties supplied in the `ilog.views.maps` package, you can write specially named properties as explained in *Advanced Features of JViews Framework*.

Overview of renderers

A renderer is an object that is used to transform a map feature into a graphic object of the class `IlvGraphic` or one of its subclasses.

A renderer must implement the `IlvFeatureRenderer` interface, which is supplied in the `ilog.views.maps` package. To transform a given map feature into a graphic object, you use its `makeGraphic` method:

```
IlvGraphic makeGraphic(IlvMapFeature feature,
                       IlvCoordinateTransformation tr);
```

The second argument, `tr`, allows you to specify a coordinate transformation. A typical way of constructing such a transformation is to call `IlvCoordinateTransformation.CreateTransformation` with `feature.getCoordinateSystem` as its first parameter (the Source coordinate system) and the coordinate system set on the Manager (through `IlvCoordinateSystemProperty`) as second parameter (the Target coordinate system). For information about coordinate systems and coordinate transformations, see *Converting coordinates between coordinate systems*.

JViews Maps includes a set of default renderers for each one of the geometry types available in the library. These renderers can be found in the package `ilog.views.maps.rendering`. The `IlvMapPointRenderer`, for example, transforms a map feature whose geometry is a point into an object of the type `IlvMapMarker`. The library also provides a global default renderer of the type `IlvDefaultFeatureRenderer`, which you can use to translate any map feature whose geometry is one of the predefined geometries. This renderer is in the `ilog.views.maps` package.

The following code example shows how to transform a map feature whose geometry is of the type `IlvMapLineString` into green polylines with a thickness of four pixels if the scale is greater than 1/1,000,000. These polylines could be, for example, the segments of a country's border.

```
IlvMapLineRenderingStyle style = new IlvMapLineRenderingStyle(); style.
setForeground(Color.green);
style.setLineWidth(4);
style.setScale(1f/1000000f);

IlvDefaultCurveRenderer renderer = new IlvDefaultCurveRenderer(); renderer.
setLineRenderingStyle(style);

try {
  // Identity transformation.
  IlvCoordinateTransformation identity = IlvCoordinateTransformation.
CreateTransformation(null, null);
  IlvGraphic graphic = renderer.makeGraphic(feature, identity);

  // Adding the graphic object into a manager.
  manager.addObject(graphic, layerIndex, true);
} catch (IlvMapRenderException e) {
  // Might occur if the geometry is not a curve.
  System.out.println("This renderer can't translate the map feature");
  System.out.println(e.getMessage());
} catch (IlvCoordinateTransformationException te) {
  // Might occur if the coordinate transformation could not be
  // performed.
  System.out.println("This renderer could not transform the geometry");
  System.out.println(te.getMessage());
}
```

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/renderer/src/RendererSample.java`**

Creating a colored line renderer

The complete source code example for this renderer can be found in the following file:

<installdir> **/jviews-maps86/codefragments/renderer/src/ColorLineRenderer.java**.

Suppose that you want to display contour lines of different elevations with different colors. A simple solution would consist of indexing a color using the elevation value by means of a `java.awt.image.ColorModel`. More generally, it would be useful to have a renderer class that applies a color to graphic objects, such as lines, polygons, or text, by using any of the attributes associated with a map feature.

The `makeGraphic` method in the `ColorLineRenderer` class builds an `IlvPolyline` graphic object from the interface `IlvMapMultiPointInterface`. This interface is common to all geometries composed of a set of points.

```
public IlvGraphic makeGraphic(IlvMapFeature feature,
                              IlvCoordinateTransformation tr)
    throws IlvMapRenderException, IlvCoordinateTransformationException
  {
    // Check that this geometry can be processed.
    IlvMapMultiPointInterface multiPoint;
    try {
      multiPoint = (IlvMapMultiPointInterface)feature.getGeometry();
    } catch (Exception e) {
      throw new IlvMapRenderException("not a multipoint geometry");
    }

    // Check that something has to be done.
    int pointCount = multiPoint.getPointCount();
    if (pointCount == 0)
      return null;

    // Allocate polyline point array.
    IlvPoint p[] = new IlvPoint[pointCount];

    // Convert points.
    for (int i = 0; i < pointCount ; i++) {
      p[i] = new IlvPoint();
      IlvProjectionUtil.ToViews(tr,
                                multiPoint.getPoint(i),
                                p[i]);
    }
    // Create the graphic object, without duplicating the p[] array.
    IlvPolyline poly = new IlvPolyline(p, false);
   [...]
```

The map feature coordinates must be converted to the manager coordinate system. This conversion implies a change of orientation of the y-axis since cartographic data coordinate systems have the positive portion of their y-axis oriented upward, whereas the manager has it oriented downward. It might also imply a coordinate transformation by changing the source coordinate system of the data into a target coordinate system. In our example, the `ToViews(ilog.views.maps.IlvCoordinate, ilog.views.IlvPoint, ilog.views.maps.`

projection.IlvProjection, ilog.views.maps.projection.IlvProjection) method both transforms the coordinates, if necessary, and corrects the orientation of the y-axis. Note that the IlvCoordinateTransformation can be identity by setting the source (feature.getCoordinateSystem()) and target coordinate systems to null, especially if the source data is not georeferenced (that is, its projection is not known), or does not need to be transformed. For further information about coordinate systems, see *Handling spatial reference systems*.

Once the graphic object is created, the attribute value for coloring the lines using a color model is retrieved, as shown below:

```
int colorIndex = 0;
   // Get attribute list.
   IlvFeatureAttributeProperty attributeList = feature.getAttributes();
   if (attributeList != null) {
     try {
       IlvFeatureAttribute attribute = null;
       attribute = attributeList.getAttribute(myAttributeName);
       if (attribute instanceof IlvIntegerAttribute)
         colorIndex = ((IlvIntegerAttribute)attribute).getValue();
       else if (attribute instanceof IlvDoubleAttribute)
         colorIndex = (int)((IlvDoubleAttribute)attribute).getValue();
     } catch (IllegalArgumentException e) {
       // No attribute found.
       colorIndex = 0;
     }
   }
   Color color = new Color(myColorModel.getRed(colorIndex),
                           myColorModel.getGreen(colorIndex),
                           myColorModel.getBlue(colorIndex));
   // Sets the color of graphic.
   poly.setForeground(color);
```

Making a renderer persistent

There are certain situations where you might want to save a renderer. When you work in load-on-demand mode, for example, only the parameters necessary for loading the graphic objects in the layer are saved, not the objects themselves. Load-on-demand is described in *Using load-on-demand*.

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/renderer/src/ColorLineRenderer.java`**

If the graphic objects are created using a specific renderer, you must save that renderer to render the objects in the same way the next time they are loaded. The class `IlvSDOLayer`, for example, lets you specify a renderer (`setFeatureRenderer(ilog.views.maps.IlvFeatureRenderer)` that will be saved with the layer.

The `ColorLineRenderer` presented in the section *Creating a colored line renderer* derives from the `IlvFeatureRenderer` interface, which extends `IlvPersistentObject` and can thus be saved.

```
  /**
   * Writes this to specified stream.
   */
  public void write(IlvOutputStream stream)
  throws java.io.IOException
  {
    stream.write("attributeName",myAttributeName);
    if (myColorModel instanceof IlvPersistentObject)
      stream.write("colorModel",(IlvPersistentObject)myColorModel);
    else
      System.err.println("Warning : colormodel not saved");
  }
  public ColorLineRenderer(IlvInputStream stream)
  throws IlvReadFileException
  {
    myAttributeName = stream.readString("attributeName");
    try {
      myColorModel = (ColorModel)stream.readPersistentObject("colorModel");

    } catch (IlvFieldNotFoundException e) {
      // Get default colormodel
      myColorModel = IlvIntervalColorModel.MakeElevationColorModel();
    }
  }
}
```

Extending an existing renderer

Most of the time, you do not have to create a totally new renderer. You can use one of the default renderers that are supplied in the package and tailor it to your needs.

This section shows how to extend an `IlvDefaultPointRenderer` to add text of the type `IlvLabel` next to the point of the feature being rendered. It also shows the use of `IlvMapRenderingStyle` which are the classes used by the renderers in order to customize them. For instance, in this code example, the color of the labels generated by the renderer is obtained through the `IlvMapPointRenderingStyle` of the `IlvDefaultPointRenderer`.

The complete source code for the example in this section can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/renderer/src/MarkerTextRenderer.java`**.

The text is stored in an attribute whose name is provided for the class `MarkerTextRenderer`. When this text exists, it is returned with the marker generated by the superclass of the renderer; otherwise only the marker is returned.

```
public IlvGraphic makeGraphic(IlvMapFeature feature,
                              IlvCoordinateTransformation tr)
    throws IlvMapRenderException, IlvCoordinateTransformationException
  {
    // Let the super class create the marker.
    IlvMarker marker = (IlvMarker)super.makeGraphic(feature, tr);

    // Create label if needed.
    IlvLabel label = null;
    IlvFeatureAttributeProperty attributeList = feature.getAttributes();
    if (attributeList != null) {
      try {
        IlvFeatureAttribute attribute = null;
        attribute = attributeList.getAttribute(myAttributeName);
        if (attribute != null)
          label = new IlvLabel(marker.getPoint(), attribute.toString());
      } catch (IllegalArgumentException e) {
        label = null;
      }
    }

    // Case no label: return marker.
    if (label == null) {
      return marker;
    }
    // Else generate a graphic set containing the marker as the label.
    else {
      // Make this label of the same color than the marker.
      label.setForeground(getPointRenderingStyle().getMarkerColor());

      IlvGraphicSet set = new IlvGraphicSet();
      set.addObject(marker, false);
      set.addObject(label, false);
      return set;
```

```
        }
    }
```

Using CSS to customize the rendering process

Usually, renderers can be customized by means of their own Java™ API. For example, to obtain blue lines, you can use `IlvDefaultCurveRenderer` and call the `setForeground` method of its `IlvMapLineRenderingStyle`. To customize a renderer easily and quickly, you can also use the `IlvMapCSSRenderer` class, which is an `IlvDefaultFeatureRenderer` (it can render any known `IlvMapGeometry`).

Note: This feature is difficult to use with map layers having their own style. It leads to a situation in which each object style can be defined in two ways.

For more information about CSS, see *Using CSS Syntax in the Style Sheet*, in *IBM® ILOG® JViews Diagrammer, Developing with the SDK*.

Cascading style sheets

For its customization, the `IlvMapCSSRenderer` class uses *cascading style sheets*, or CSS files. The customization is done so that for the same Java code you can have different style sheets, that is, different rendering aspects.

Basically, a style sheet is a set of rules. For more details on the structure of the CSS, refer to *Using CSS Syntax in the Style Sheet*, in *IBM® ILOG® JViews Diagrammer, Developing with the SDK*.

The complete source code of this example can be found in the following files:

<installdir> **`/jviews-maps86/codefragments/renderer/src/CSSRenderer.java`**

<installdir> **`/jviews-maps86/codefragments/renderer/data/style.css`**.

Here is how to use the `IlvMapCSSRenderer` class. Imagine, for example, that you have a shapefile data file, `roads.shp`, and you need to render the roads in red. Here is the simplest way to do it by means of a CSS:

```
IlvMapCSSRenderer cssRenderer =
 new IlvMapCSSRenderer(null, //use as default renderer
        "roads",
        new String[] { "simple.css"});
 IlvMapLoader loader = new IlvMapLoader(manager);
 IlvMapFeatureIterator iterator = loader.makeFeatureIterator("roads.shp");
    loader.load(iterator, cssRenderer);
```

and the `simple.css` file would be:

```
#roadsLinesStyle {
    class : 'ilog.views.maps.rendering.IlvMapLineRenderingStyle';
    foreground : red;
}
```

```
roads {
    class : 'ilog.views.maps.rendering.IlvDefaultCurveRenderer';
    lineRenderingStyle : @=roadsLinesStyle;
}
```

Later in this section, all the content of the `corsica.css` file contained in the CSS demo (`<installdir>/JViews Maps86/samples/css/data`) is examined and explained.

Customizing the CSS renderers

The following extracts of CSS code show how the user can customize the CSS renderer.

♦ Set the debug mask to 0, so that no debug information will be printed. The maximum debug information is obtained by setting the debug mask to 65535.

```
#IlvMapCSSRenderer {
    styleSheetDebugMask : 0;
}
```

♦ Customize the rendering of the `airports.shp` shapefile which contains Point geometries. Use an `IlvLabeledPointRendererIlvLabeled`.

```
#airportsPointStyle {
    class : 'ilog.views.maps.rendering.IlvMapPointRenderingStyle';
    markerSize : 8;
    markerType : 'FilledDiamond|Square';
    markerColor : #99ffee04;
}

#airPortsTextStyle {
    class : 'ilog.views.maps.rendering.IlvMapTextRenderingStyle';
    backgroundPaint : yellow;
    framePaint : green;
    innerMargin : 2;
    maximumHeight :20;
    minimumHeight : 15;
    scale : 0.0000025;
    antialiasing : true;
}

airports {
    class : "ilog.views.maps.labelling.IlvLabeledPointRenderer";
    attributeNames : nam;
    pointRenderingStyle : @=airportsPointStyle;
    textRenderingStyle : @=airPortsTextStyle;
}
```

♦ Create an instance of `IlvLabeledPointRenderer`, customize it and use it to render the shapefile. It is equivalent to the following Java code:

```
IlvMapPointRenderingStyle pStyle = new IlvMapPointRenderingStyle();
```

```
pStyle.setMarkerSize(8);
pStyle.setMarkerType(IlvMarker.IlvMarkerFilledDiamond |
                                         IlvMarker.IlvMarkerSquare);
pStyle.setMarkerColor(new java.awt.Color(255, 238, 4, 153));
// The #99ffee04 color notation has the same syntaxe as HTML,
// the first 2 digits after the # represent the alpha channel.
IlvMapTextRenderingStyle tStyle = new IlvMapTextRenderingStyle();
tStyle.setBackgroundPaint(Color.yellow);
tStyle.setFramePaint(Color.green);
tStyle.setInnerMargin(2);
tStyle.setMaximumHeight(20);
tStyle.setMinimumHeight(15);
tStyle.setScale(0.0000025);
tStyle.setAntialiasing(true);
IlvLabeledPointRenderer labelRender =
       new IlvLabeledPointRenderer();
labelRenderer.setAttributeNames (new String[] {"nam"});
labelRenderer.setPointRenderingStyle(pStyle);
labelRenderer.setTextRenderingStyle(tStyle);
```

♦ Create an `IlvDefaultCurveRenderer` to render the `coastlines.shp` shapefile that contains Line geometries:

```
#coastlinesStyle {
    class : 'ilog.views.maps.rendering.IlvMapLineRenderingStyle';
    foreground : black;
    lineWidthZoomed : true;
    lineWidth : 2;
    lineStyle : 4.2,4.3;
    lineJoin : JOIN_BEVEL;
    endCap : Cap_Round;
}

coastlines {
    class : 'ilog.views.maps.rendering.IlvDefaultCurveRenderer';
    lineRenderingStyle : @=coastlinesStyle;
}
```

Note: **1.** In the previous code extract the `class` keyword was used to tell the renderer to instantiate the corresponding class and to use it. Here an `ilog.views.maps.rendering.IlvMapLineRenderingStyle` class is instantiated to customize an instance of the `ilog.views.maps.rendering.IlvDefaultCurveRenderer` class.

2. You can assign a `float` array to the `lineStyle` keyword, as well as some predefined values such as `Dash`, `Dot`, and so on. Moreover, values assigned to the `LineStyle`, `lineJoin` and `endCap` are not case sensitive.

♦ Create an `IlvLabeledPointRenderer` to render the `cities.shp` shapefile that contains Point geometries:

```
#citiesPointStyle {
    class : ilog.views.maps.rendering.IlvMapPointRenderingStyle ;
    markerSize : 3;
    markerType : FilledCircle;
    markerColor : blue
}

#col1 {
    class : 'java.awt.Color(red, green, blue)' ;
    red : 0;
    green : 0;
    blue : 200;
}

#col2 {
    class : 'java.awt.Color(red, green, blue, transparency)' ;
    red : 150;
    green : 200;
    blue : 255;
    transparency : 180;
}

#citiesLabel {
    class : ilog.views.maps.rendering.IlvMapTextRenderingStyle ;
    backgroundPaint : @=col2;
    labelFillColor : black;
    framePaint : @=col1;
    innerMargin : 2;
    maximumHeight :15;
    minimumHeight : 10;
    antialiased : true;
    scale : 0.0000025;
}

cities {
    class : ilog.views.maps.labelling.IlvLabeledPointRenderer;
    attributeNames : NAME,txt;
    rejectedValues : UNK;
    pointRenderingStyle : @=citiesPointStyle;
    textRenderingStyle : @=citiesLabel;
}
```

Notice here the syntax of the `#col1` node. This shows how you can instantiate any kind of objects using the CSS. You just have to provide the `class` keyword in the node scope and link it to the definition of the constructor which is given with all the necessary parameters: `class : 'java.awt.Color(red, green, blue)'`.

You can see here that an AWT Color with its three channels is constructed: red, green, and blue. To construct the Color, you have to provide the values of each parameter given in the list. For example, here it is 0 for red, 0 for green, and 200 for blue.

♦ Create an `IlvRailroadRenderer` to render the `roads.shp` shapefile that contains Line geometries. The `IlvRailroadRenderer` is a renderer provided with its source code in the

CSS demo. It is designed to render line geometries as roads or railroads. Note that, using the CSS, you can even customize your own renderer:

```
#col3 {
    class : 'java.awt.Color(red, green, blue, transparency)' ;
    red : 220;
    green : 10;
    blue : 10;
    transparency : 100;
}

#roadsAttributes {
    class : 'ilog.views.maps.graphic.IlvRailroadAttributes';
    drawingTies : false;
    background : @=col3;
    railColor : #66ff0000;
    railSpacing : 1;
    maximumRailSpacing : 1;
    scale : 0.000000025;
}

roads {
    class : 'ilog.views.maps.rendering.IlvRailroadRenderer';
    attributes : @=roadsAttributes;
}
```

♦ Create an `IlvDefaultAreaRenderer` to render the `builtareas.shp` shapefile that contains Polygon geometries:

```
#pattern1 {
   class : 'ilog.views.util.java2d.IlvPattern(type, foreground, background)
';
   type: THICK_DIAGONAL_GRID;
   foreground: gray;
   background: wheat;
}

#builtareasLineStyle {
    class : 'ilog.views.maps.rendering.IlvMapLineRenderingStyle';
    foreground : maroon;
    lineWidthZoomed : true;
    lineWidth : 2;
    lineJoin : join_Miter;
    endCap : CAP_Round;
}

#builtareasAreaStyle {
    class : 'ilog.views.maps.rendering.IlvMapAreaRenderingStyle';
    fillingObject : true;
    fillPattern : @=pattern1;
    drawingStroke : true;
    lineRenderingStyle : @=builtareasLineStyle;
}
```

```
builtareas {
    class : 'ilog.views.maps.rendering.IlvDefaultAreaRenderer';
    usingGeneralPath : true;
    areaRenderingStyle : @=builtareasAreaStyle;
}
```

You can see here how to create an instance of `IlvPattern` and use it for the rendering process. The Java code corresponding to this CSS part of code is the following:

```
IlvPattern pattern1 =
       new IlvPattern(IlvPattern.THICK_DIAGONAL_GRID,
                      Color.gray,
                      new Color(245,222,179));
IlvMapLineRenderingStyle builtareasLineStyle = new
   IlvMapLineRenderingStyle();
builtareasLineStyle.setForeground(new Color(128, 0, 0));
builtareasLineStyle.setLineWidthZoomed(true);
builtareasLineStyle.setLineWidth(2);
builtareasLineStyle.setLineJoin(BasicStroke.JOIN_MITER);
builtareasLineStyle.setEndCap(BasicStroke.CAP_ROUND);
IlvMapAreaRenderingStyle builtareasAreaStyle = new
   IlvMapAreaRenderingStyle();
builtareasAreaStyle.setFillingObject(true);
builtareasAreaStyle.setFillPattern(pattern1);
builtareasAreaStyle.setDrawingStroke(true);
builtareasAreaStyle.setLineRenderingStyle(builtareasLineStyle);

IlvDefaultAreaRenderer builtareas = new IlvDefaultAreaRenderer();
builtareas.setUsingGeneralPath(true);
builtareas.setAreaRenderingStyle(builtareasAreaStyle);
```

♦ Create an `IlvRailroadRenderer` to render the railroads`.shp` shapefile that contains Line geometries:

```
#railRoadStyle {
    class : 'ilog.views.maps.graphic.IlvRailroadAttributes';
    railSpacing : 1;
    scale : 0.00000025;
    maximumRailSpacing : 1;
    tieWidth : 3;
    maximumTieWidth : 3;
    tieSpacing : 4;
    slantingLimit : 15;
    railColor : green;
    tieColor : #ff40ff40;
}

railroads {
    class : 'ilog.views.maps.rendering.IlvRailroadRenderer';
    attributes : @=railRoadStyle;
    dummy[0] : "@#col1";
    dummy[1] : "@#col2";
    dummy[2] : "@#col3";
}
```

Another interesting aspect of this part of the CSS code is the manipulation of indexed properties, called *dummy* here. The `IlvRailroadRenderer` class defines a dummy attribute member, which is an array of `String` (`String[]`).

To customize this array by using the CSS, you need to:

1. Define the `get/set` methods according to the Java Beans specifications.

Here is the part of the code of `IlvRailroadRenderer` that declares these methods:

```
public Color getDummy(int i)
{
  return _dummy[i];
}

public Color[] getDummy()
{
  return _dummy;
}

public void setDummy(Color[] val)
{
  _dummy = val;
}

public void setDummy(int i, Color val)
{
  _dummy[i] = val;
}
```

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/renderer/src/IlvRailroadRenderer.java`**.

Note: In our example, the number of dummy colors is limited to 3.

2. Define the descriptor in the `BeanInfo` class, `IlvRailroadRendererbeanInfo`:

```
public class IlvRailroadRendererBeanInfo
  extends SimpleBeanInfo
{
  private final static Class beanClass = IlvRailroadRenderer.class;

  public PropertyDescriptor[] getPropertyDescriptors()
  {
    try {
      PropertyDescriptor[] properties = {
        new PropertyDescriptor("attributes", beanClass),
        new IndexedPropertyDescriptor("dummy", beanClass)
                        // The dummy indexed property.
      };
      return properties;
    } catch (IntrospectionException e) {
      throw new Error(e.toString());
    }
  }
}
```

The complete source code of this example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/renderer/src/IlvRailroadRendererBeanInfo.java`**.

3. Define indexed properties in your CSS file by using the @# syntax.

Note: You can define a unique CSS file for all your CSS renderers. But each time you instantiate the CSS renderer, the long CSS file is parsed, and depending on the size of the CSS file, it can be time consuming. In this case, you can cut your CSS file into smaller CSS files to avoid the performance drop.

For more information about CSS, see *Using CSS Syntax in the Style Sheet* in *IBM® ILOG® JViews Diagrammer, Developing with the SDK*.

Renderers and styling

Some predefined renderers produce IlvGraphic objects, which are instances of the IlvMapGraphic class. These renderers include IlvMapAreaRenderer, IlvMapPointRenderer, IlvMapCurveRenderer and IlvMapTextRenderer. The graphic objects produced by these renders can be styled by means of an IlvMapStyle object. The predefined data sources provided by JViews Maps use these renderers and make it possible to change the style of a layer without reloading the entire map.

The following code creates an IlvShapeDataSource and sets an IlvMapAreaRenderer to render the feature read by the data source. The fill color is set to red. A button is created whose action sets the color of the map to blue simply by changing the Paint attribute of the layer.

```
IlvShapeDataSource ds = new IlvShapeDataSource(shapeFile);
ds.setManager(view.getManager());
IlvMapAreaRenderer renderer = new IlvMapAreaRenderer(false, false);
IlvGraphicPathStyle style = new IlvGraphicPathStyle();
ds.getInsertionLayer().setStyle(style);
style.setFilling(true);
style.setPaint(Color.red);
ds.setFeatureRenderer(renderer);

JButton b = new JButton();
b.addActionListener(new ActionListener() {
public void actionPerformed(ActionEvent e) {
  IlvGraphicPathStyle style =      (  (IlvGraphicPathStyle)ds.getInsertionLayer
().getStyle();
  style.setPaint(Color.blue);
  view.repaint();
 }
});
```

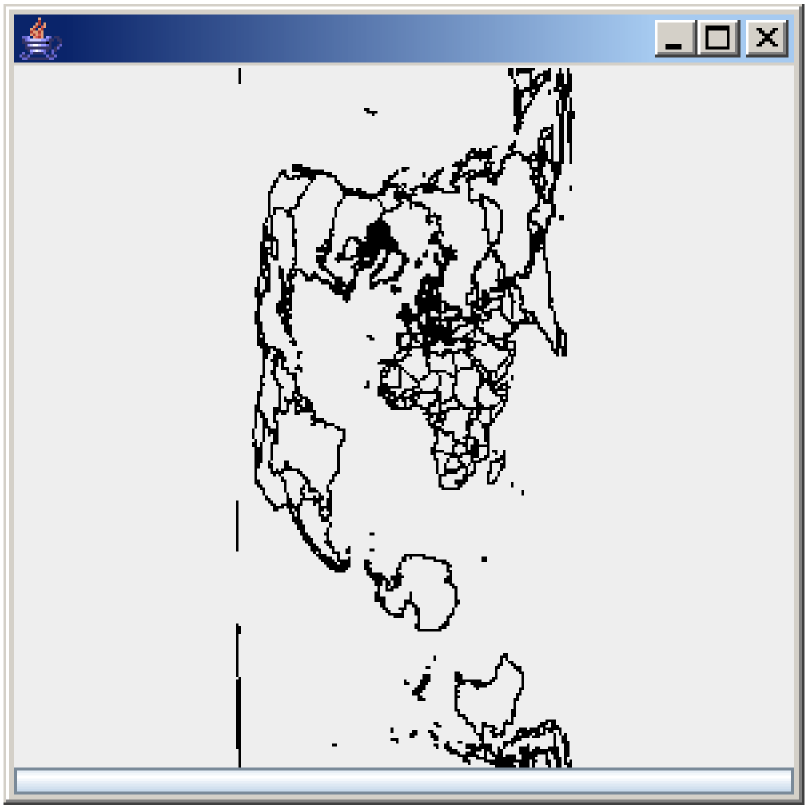
Rendering with a geodetic computation

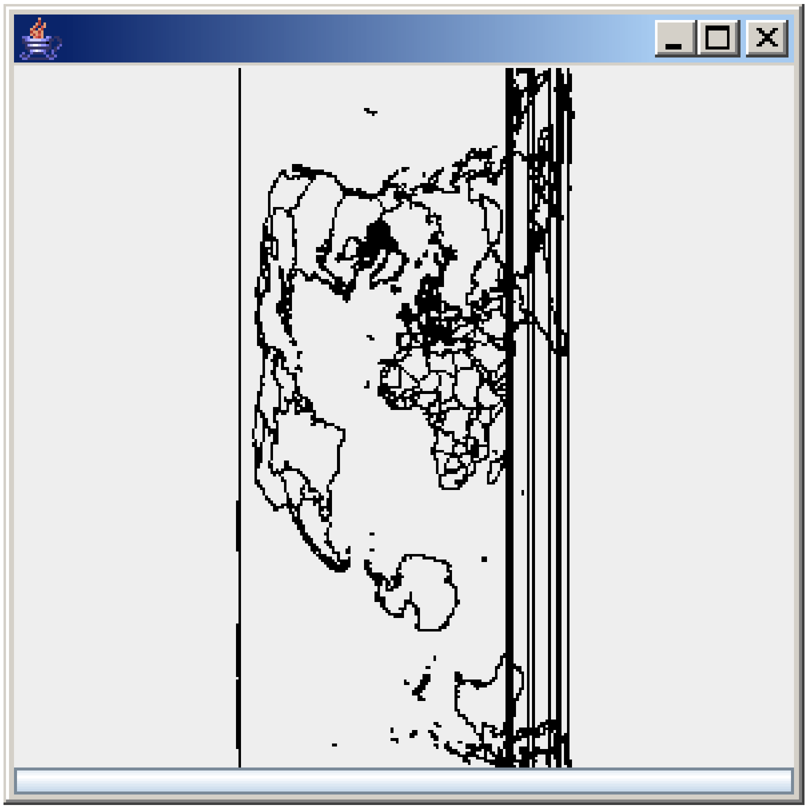
The `IlvMapAreaRenderer` and `IlvMapCurveRenderer` provide a geodetic computation option. Geodetic computation includes date line wrapping and reprojection of features that go outside of the projection limits.

To create a renderer that performs geodetic computations, just pass `true` to the specified argument:

```
IlvShapeDataSource ds = new IlvShapeDataSource(shapeFile);
ds.setManager(view.getManager());
// perform geodetic computations
IlvMapAreaRenderer renderer = new IlvMapAreaRenderer(false, true);
ds.setFeatureRenderer(renderer);
```

The figures show geodetic computation with and without date line wrapping:


Geodetic computation with date line wrapping


Geodetic computation without date line wrapping

Handling spatial reference systems

Describes various conversions.

In this section

Converting between two spatial reference systems

The conversion between two Spatial Reference Systems (SRS) is performed by a coordinate transformation, that is, a mathematical transformation or a chain of mathematical transformations.

For example, when superimposing Global Positioning System (GPS) points (expressed in longitude and latitude) on a state map (where coordinates are expressed in meters, in UTM projection), the GPS coordinates are expressed in a geographic coordinate system, while the state map coordinates are in a projected coordinate system. Once the SRS for each data source is known, the JViews Maps package is able to construct a mathematical transformation to convert from one coordinate system to another.

SRS and coordinate transformations are mainly used for two purposes.

The first one is to render the pipeline of the reader framework, and the main steps are:

1. Defining the source SRS

   The SRS of source data is often stored in the reader or in the feature of the iterator that is used. If not, the reader provides the `setSourceCoordinateSystem()` method. The SRS of source data is stored in map features returned by the iterator.

2. Defining the target SRS

   The SRS of target data is optional. In fact, the rendering mechanism only needs a coordinate transformation. Before rendering your graphic objects, you have to create the coordinate transformation once.

3. Creating the coordinate transformation

   If no coordinate transformation is needed, the rendering can be performed using an identity transformation.

4. Iterating on map features provided by the iterator, and rendering them.

The second one is to position free data, (without the renderers, you create the coordinate of the graphic object yourself), and the main steps are nearly the same:

1. Defining the source SRS.
2. Defining the target SRS.
3. Creating the coordinate transformation.
4. Transforming each coordinate manually.

The following packages define some useful SRS to be used with maps:

♦ `ilog.views.maps.srs.coordsys`

  Contains coordinate systems and related classes.

♦ `ilog.views.maps.srs.coordtrans`

  Defines the transformations between coordinate systems.

♦ `ilog.views.maps.srs.wkt`

Defines factories and utility classes to convert coordinate systems to and from Open GIS Well-Known Text (WKT) specifications.

- `ilog.views.maps.projection`

  Defines projections used in projected coordinate systems.

> **Note**: The `projection` package also contains the definition of ellipsoids and geodetic datum (horizontal datum), even when these are not specific to projections and are used by the `coordsys` package. This is to keep the backward compatibility with JViews Maps versions prior to 5.0, where only projected coordinate systems were handled through the use of projections.

Converting coordinates between coordinate systems

Once coordinate systems are defined, there must be some way to convert from one coordinate system to another. This is done using coordinate transformations, defined in the package `ilog.views.maps.srs.coordtrans`.

Overview of the coordinate transformations

As soon as you work with different coordinate systems and datums, you need to be able to convert from one to another. When working with coordinate systems, this can be performed using functions.

The coordinate transformations, as defined by the `ilog.views.maps.srs.coordtrans.IlvCoordinateTransformation` class, contain the information that allows you to manipulate them:

- ♦ The *source* coordinate system
- ♦ The *target* coordinate system
- ♦ The *Math Transform* to convert coordinates from one to another

In addition to that, all the coordinate transformations implement the following methods:

- ♦ `IlvCoordinate transform(ilog.views.maps.IlvCoordinate, ilog.views.maps.IlvCoordinate)`

  This method is the basic one. It transforms the source coordinate, storing the result in `result`, or in a newly allocated coordinate if `result` is `null`. The method returns the transformed point.

- ♦ IlvCoordinate[] `transform(ilog.views.maps.IlvCoordinate[], ilog.views.maps.IlvCoordinate[])`

  This is the vectorized version of the previous method. This allows the transformations to convert a whole batch of coordinates, possibly allowing some optimizations to be performed.

- ♦ Optionally, transformations can implement the `getInverse()` method that returns an inverse transformation.

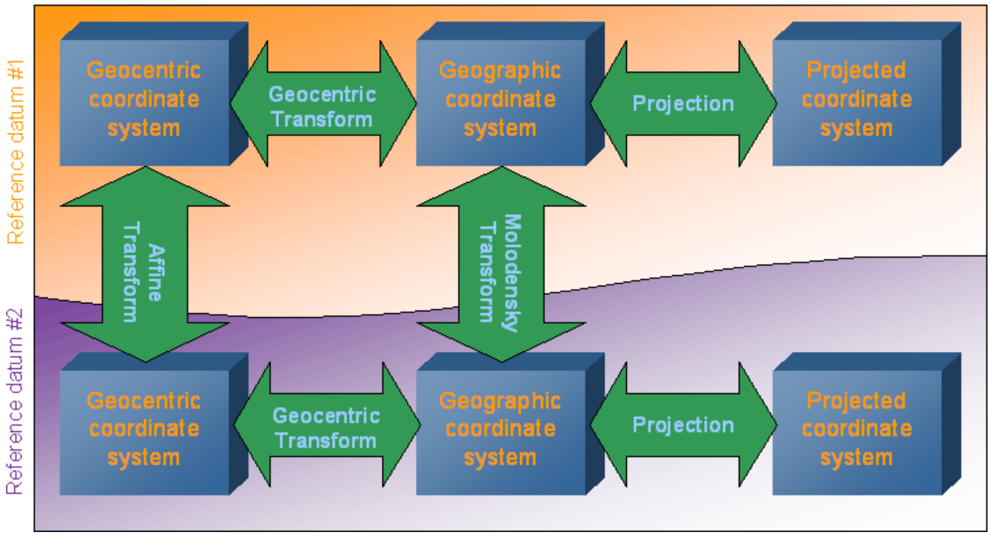
The following sections describe the predefined built in transformations of JViews Maps, and provide some examples of the transformation package.

Transformation paths

To transform the coordinates from one coordinate system to another, mathematical functions are needed. These mathematical functions can be either simple straight forward functions, or more complicated transformations. The JViews Maps package includes the most elementary transformations (or transformation steps) used for coordinate conversions, and these are available in the `ilog.views.maps.srs.coordtrans` package.

The chaining of elementary transformations from one coordinate system to another is called a *transformation path*.

The static method `CreateTransformation(ilog.views.maps.srs.coordsys.IlvCoordinateSystem, ilog.views.maps.srs.coordsys.IlvCoordinateSystem)` from the class `IlvCoordinateTransformation` can automatically create transformation paths from coordinate systems of the `ilog.views.maps.srs.coordsys` package. To find the transformation paths created, refer to the following figure:


Reference datum #1
Geocentric coordinate system
Geocentric Transform
Geographic coordinate system
Projection
Projected coordinate system
Affine Transform
Molodensky Transform
Reference datum #2
Geocentric coordinate system
Geocentric Transform
Geographic coordinate system
Projection
Projected coordinate system

Transformation paths

Example

Convert from the following projected coordinate system:

- ♦ Projection: Lambert Azimuthal Conformal Conic
- ♦ Datum: NTF (Nouvelle Triangulation de la France)
- ♦ Associated geographic coordinate system based on Clark 1880 ellipsoid, IGN modified.

to the following one:

- ♦ Projection: Mercator
- ♦ Datum: European 1960
- ♦ Associated geographic coordinate system: International.

The transformation path selected by `CreateTransformation()` will be:

1. Projection transformation from Lambert to geographic NTF/Clark1880
2. Molodensky conversion from NTF/Clark1880 to European 1960/International systems
3. Projection transformation from geographic European 1960/International to Mercator.

At each step, the relevant unit conversion is added by using the affine transforms.

Note that this path is not the only existing path. It is also possible to convert from geographic coordinates to geocentric coordinates, in which case the transformation path would be:

1. Projection transformation from Lambert to geographic NTF/Clark1880
2. Geocentric transform to geocentric NTF coordinates
3. Affine transform to convert from NTF to European 1960 coordinates
4. Geocentric transform from European 1960 to geographic coordinates
5. Projection transformation from geographic European 1960/International to Mercator.

Predefined math transformations

Abridged Molodensky Transform

The standard way to convert coordinates from one datum to another is to convert first the coordinates to geocentric coordinates, apply the datum shift and rotation parameters, and then convert them back to geographic coordinates.

As an alternative to this transform, the `IlvAbridgedMoldenskyTransform` class implements directly a transform derived directly from the Moldensky formulas. The Abridged form of these formulas are quite satisfactory for three-parameter transformations.

This transform can work on either 2-D (only latitude and longitude are modified), or 3-D (the ellipsoidal height of coordinates is also modified).

Affine Transform

Affine transforms are commonly used in coordinate transformation. An affine transform is simply defined by a 4x4 double values matrix, and are applied to coordinates by multiplying them as if they were one 1x4 matrix.

Affine transforms are mainly used in JViews Maps for unit conversions in a transformation path.

Another use of affine transforms is to use them to implement Bursa Wolf transformations. The Bursa Wolf transformation is applied to geocentric coordinates to model a seven-parameter datum change. A seven-parameter datum is defined by the *dX*, *dY*, *dZ* axis shifts, the *eX*, *eY*, *eZ* rotations around the axis, and a scale factor expressed in parts per million. The matrix to use for Bursa Wolf transformation is:

$$S = 1 + \mathrm{ppm} \times 1000000$$

$$M = \begin{bmatrix} S & (-eZ \times S) & (eY \times S) & dX \\ (eZ \times S) & S & (-eX \times S) & dY \\ (-eY \times S) & (eX \times S) & S & dZ \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Concatenated Transform

There are some cases where a straightforward mathematical function cannot be found to convert from one coordinate system to another. In those cases, some elementary transformations can be chained to build the full transformation. This is typically the case when converting from a projected coordinate system P1 to another one P2, using different datums: first you need to convert coordinates from P1 to the geographic coordinate system, then apply a datum conversion on these coordinate systems, and then convert them to the final coordinate system P2.

The `IlvConcatenatedTransform` class allows multiple transformation steps to be chained and used as a unique transformation.

Geocentric Transform

The function used to convert from geocentric to ellipsoid and the function used to convert from ellipsoid to geocentric are grouped together in the `IlvGeocentricTransform` class. Actually, the transformation is performed by specialized versions of this class:

♦ `IlvGeocentricTransform.GeocentricEllipsoidal`

  converts geocentric coordinates (x, y, z) to ellipsoidal (or geographic) coordinates (lon, lat, H)

♦ `IlvGeocentricTransform.EllipsoidalGeocentric`

  converts ellipsoidal coordinates (lon, lat, H) to geocentric coordinates (x, y, z)

Projection Transform

The `IlvProjectionTransform` class implements a transformation in which an `IlvProjection` is used to convert coordinates from geographic coordinates to a projected coordinate system, and vice versa.

This transform encapsulates a projection, and uses the `forward(ilog.views.maps.IlvCoordinate)` method if the transformation is a forward transformation, or the `inverse(ilog.views.maps.IlvCoordinate)` method if the transformation is an inverse transformation.

Transforming data

This section is based on a simple example that illustrates the basic operations required to define the coordinate systems, and shows how to use the coordinate transformations back and forth.

The complete source code of the example on which this section is based can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/srs/src/Sample1.java`**.

Example overview

The `Sample1.java` file contains a very simple program that shows how to convert coordinates from geographic coordinates to a projected coordinate system, using a Mercator projection.

This class has only a static `main()` method, in which the coordinate systems are instantiated and the coordinate is transformed.

Choosing a source and a destination coordinate system

As our example uses coordinate systems and transformation, the relevant packages must be imported, as well as the projection package for the Mercator projection definition:

```
import ilog.views.maps.*;
import ilog.views.maps.srs.coordsys.*;
import ilog.views.maps.srs.coordtrans.*;
import ilog.views.maps.projection.*;
```

The first important step is to define the source and the target coordinate systems.

For the source coordinate system, some latitude and longitude coordinates expressed in degrees are needed. This is the kind of coordinate defined in a geographic coordinate system. In this example, the WGS84 coordinate is used. The WGS84 geographic coordinate system defines ellipsoidal coordinates over the standard WGS84 ellipsoid.

```
IlvCoordinateSystem sourceCS = IlvGeographicCoordinateSystem.WGS84;
```

Now these coordinates must be changed to Mercator coordinates. Then you can create a new projected coordinate system using the Mercator projection to express coordinates. Note that the Mercator projection should use exactly the same geodetic parameters as the WGS84 geographic coordinate system. The latter is passed as the geographic coordinate system of the projected coordinate system.

```
IlvCoordinateSystem targetCS =
   new IlvProjectedCoordinateSystem("Mercator",
                                    IlvGeographicCoordinateSystem.WGS84,
                             new IlvMercatorProjection(),
                             IlvLinearUnit.METER,
                             "X",   // The X axis name
                             "Y");  // The Y axis name
```

Transforming coordinates

In order to transform the coordinates, you need an `IlvCoordinateTransformation`. This is performed by calling the automatic transformation creation method:

```
IlvCoordinateTransformation CT =
      IlvCoordinateTransformation.CreateTransformation(sourceCS, targetCS);
```

To convert coordinates, you just have to store the coordinates in an `IlvCoordinate`, and then call the `transform()` method of the coordinate transformation.

```
// The coordinate to convert : 45W, 30N
IlvCoordinate coord = new IlvCoordinate(-45D, 30D);

try {
   coord = CT.transform(coord,
                        coord); // put the result in coord
}
catch (IlvCoordinateTransformationException e) {
   System.out.println("Transformation exception for this data");
}
```

The `transform()` method takes two parameters: the first one is the source coordinate to transform, the second one is an `IlvCoordinate` to hold the result of the transformation. When this second parameter is `null`, a new `IlvCoordinate` is allocated and used. The method returns the result coordinate.

Of course, as in this example, it is possible to use the same `IlvCoordinate` as source and destination.

Note that the `IlvCoordinateTransformation.transform()` method may throw different kinds of exception if the transformation process leads to mathematical errors or overflows. Most of the time, when those methods are thrown, the transformation is not defined at the specified point.

Displaying the result

The result of the transformation is now stored in the `coord` variable. In the following code example, the result is expressed in meters, which is the default measurement unit.

```
System.out.println("The Mercator coordinates of 45W 30N is ");
System.out.println("x = " + (int) coord.x + " m");
System.out.println("y = " + (int) coord.y + " m");
```

Note: It does not make much sense to interpret these values as distances since the center of the projection is far from the projected point, and projections do not maintain distances on a large scale

Getting the inverse transformation

At this point, the methodology to convert from geographic coordinates to Mercator coordinates is available.

To convert coordinates from Mercator back to geographic, use the `getInverse()` method, which returns the inverse transformation, if any.

```
// Get the inverse transformation.
IlvCoordinateTransformation invCT = CT.getInverse();

// Transform the point.
try {
  coord = invCT.transform(coord,
                        coord); // put the result in coord
}
catch (IlvCoordinateTransformationException e) {
  System.out.println("TransformationException exception for this data");
}
```

Printing geographic coordinates

To print geographic coordinates, you can use the `toDMS(double, boolean)` conversion method from the `IlvAngularUnit` class. This method converts an angle specified by a double value in a unit to DMS encoding.

For example, `IlvAngularUnit.DEGREE.toDMS()` converts an angle specified in degrees to DMS.

```
System.out.println("The inverse projection is "
                + IlvAngularUnit.DEGREE.toDMS(coord.x,false)
                + " "
                + IlvAngularUnit.DEGREE.toDMS(coord.y,true));
```

Adding Graphic Objects on Top of an Imported Map

This section is based on an example that loads a map of the USA projected with a Lambert Azimuthal Equal Area projection into a manager, and adds cities on top of the map. The geographic coordinates indicated by the mouse pointer as well as the name of the cities pointed to are displayed in text fields at the bottom of the window. This section also gives information about the instantiating and the parameterization of a projected coordinate system, describes how to convert data from geographic coordinates to this projected coordinate system, and how to use a view interactor.

The complete source code for this example can be found in the following file:

Note: The purpose of this example is to give a tutorial on how to use coordinate transformations. It does not use *JViews Maps* beans such as `IlvJMouseCoordinateViewer`, to integrate with a data source or for map layer management.

Initializing coordinate systems

Since you need to convert from geographic coordinates to projected coordinates and vice versa, you need to keep an instance of these transformations. This is performed by means of the `createTransformations()` method.

First, initialize an instance of the projection used in the coordinate system of the imported map. In this example, the `usa.ivl` file is in Lambert Azimuthal Equal Area projection.

The projection parameters are:

Name of the Projection	Lambert Azimuthal Equal Area
Central Meridian	100DW
Central parallel	40DN
Measurement unit	Meters (the default value)
Offset	0 (the default value)

For more information on projection parameters see the section *Projection parameters*.

```
private void createTransformations()
{
  // Create the projection.
  IlvProjection projection = new IlvLambertAzimuthalEqualAreaProjection();
  projection.setEllipsoid(IlvEllipsoid.SPHERE);
  try {
      double centralMeridian = IlvAngularUnit.RADIAN.fromDMS("100DW");
      double centralParallel = IlvAngularUnit.RADIAN.fromDMS("40DN");
      projection.setCentralMeridian(centralMeridian);
      projection.setCentralParallel(centralParallel);
  } catch (IllegalArgumentException e) {
    System.out.println("wrong string passed to "
                       + "IlvAngularUnit.RADIAN.fromDMS");
    System.out.println("unable to create the projection for the file"
                     + "usa.ivl");
    System.exit(0);
  }
```

Note that projection parameters are always specified using kernel units. In this case, the central meridian and parallel have to be specified in radians. Use the method `IlvAngularUnit.RADIAN.fromDMS()` to achieve this goal.

Once the Lambert Azimuthal projection has been initialized, you only have to create the corresponding projected coordinate system:

```
// Create the projected coordinate system.
IlvProjectedCoordinateSystem projectedCS =
  new IlvProjectedCoordinateSystem("Lambert Azimutal Equal Area",
                                   projection);
```

You also create a geographic coordinate system whose ellipsoid is a simple sphere:

```
// Create the geographic coordinate system.
IlvGeographicCoordinateSystem geoCS =
   new IlvGeographicCoordinateSystem(IlvHorizontalShiftDatum.SPHERE_WGS84,
                               IlvMeridian.GREENWICH);
```

Finally, create the coordinate transformation and store also the inverse transformation for future quick reference:

```
// A coordinate transform.
geo2projCT =
 IlvCoordinateTransformation.CreateTransformation(geoCS,projectedCS);
// The inverse transform.
proj2geoCT = geo2projCT.getInverse();
```

Adding cities

The `addCitites()` method adds a number of cities on top of the imported map of the United States:

```
private void addCities()
{
  addCity("Washington", "39D11'N", "76D51W");
  addCity("New York", "40D59'N", "73D39'W");
  addCity("Miami", "25D58'N", "80D02'W");
  addCity("San Francisco", "37D44'N", "122D20'W");
  addCity("Seattle", "47D51'N", "122D01'W");
  addCity("Denvers", "39D50'N", "104D53'W");
}
```

The coordinate conversion is performed in the `addCity()` method.

First, this method converts the DMS coordinates specified as string to degrees, in order to have the longitude and latitude coordinates of each point:

```
private void addCity(String cityName, String lat, String lon)
{
  try {
    double latitude = IlvAngularUnit.DEGREE.fromDMS(lat);
    double longitude = IlvAngularUnit.DEGREE.fromDMS(lon);
    IlvCoordinate coordinate = new IlvCoordinate(longitude,
                                                 latitude);
```

Then, it computes the projected coordinates by applying the forward coordinate transformation. If the coordinates cannot be transformed (for example, because the tolerance conditions have been exceeded), the transformation may throw an exception. This is why the code is contained inside a try/catch block.

The `IlvProjectionUtil.invertY()` method is then called to invert the y-coordinate: in the JViews Maps manager coordinate system, the y-axis is oriented downwards, whereas, in the projection coordinate system, it is oriented upwards.

```
    geo2projCT.transform(coordinate,coordinate);
    IlvPoint p = new IlvPoint((float)coordinate.x,
                              (float)coordinate.y);
    IlvProjectionUtil.invertY(p);
```

Note also that a new `IlvPoint` is allocated here. It will be used to create graphic objects (an `IlvMarker`) to represent the city. This graphic object is added to Layer #1 of the manager (Layer #0 contains the boundaries of the USA).

```
  IlvMarker marker = new IlvMarker(p, IlvMarker.IlvMarkerFilledDiamond);
      marker.setSize(4);
      marker.setForeground(Color.red);
      manager.addObject(marker, 1, false);
      marker.setName(cityName);
  } catch (IlvCoordinateTransformationException e) {
      e.printStackTrace();
  }
```

Setting the view interactor

In order to track mouse position, a view interactor is used. This is performed by means of the `setViewInteractor()` method.

First, create a selection interactor that will be used to select cities on the map. This interactor is configured so that multiple selections are not allowed, and the map cannot be modified, which means that the user will not be able to move cities around the map. The interactor is associated with the view. Note also that the elements making up Layer #0 (that is, borders) cannot be selected either.

```
private void setViewInteractor()
{
  IlvSelectInteractor interactor = new IlvSelectInteractor();
  interactor.setDragAllowed(false);
  interactor.setEditionAllowed(false);
  interactor.setMoveAllowed(false);
  interactor.setMultipleSelectionMode(false);
  manager.setSelectable(0, false);
  mgrview.pushInteractor(interactor);
```

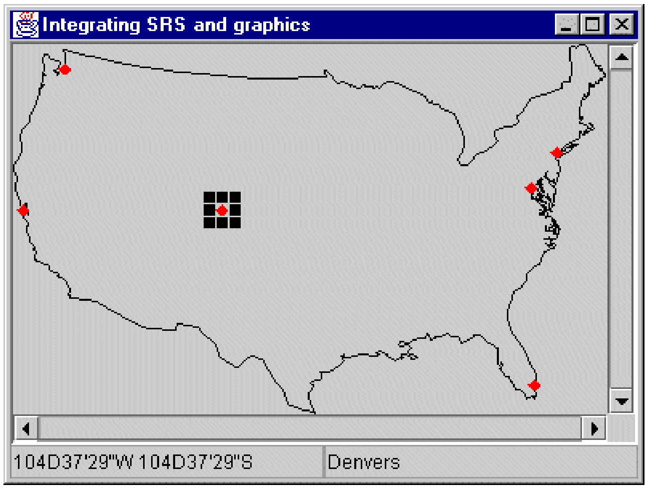
Then, add a listener to the interactor that will display the longitude and the latitude indicated by the mouse in the appropriate text field. To compute the latitude and the longitude, apply the inverse transformation computed previously. The result returned by this method is formatted with the `IlvAngularUnit.DEGREE.toDMS()` method.

```
  // Display the position of the mouse.
  interactor.addMouseMotionListener(new MouseMotionAdapter() {
    public void mouseMoved(MouseEvent e) {
      // Get the point in manager coordinates.
      IlvTransformer t = mgrview.getTransformer();
      IlvPoint p = new IlvPoint(e.getX(), e.getY());
      t.inverse(p);
      IlvProjectionUtil.invertY(p);
      // Display the mouse position.
      try {
        IlvCoordinate c = new IlvCoordinate(p.x,p.y);
        proj2geoCT.transform(c,c);
        llField.setText(IlvAngularUnit.DEGREE.toDMS(c.x,false)
                        + " " +
                       IlvAngularUnit.DEGREE.toDMS(c.x,true));
      } catch (IlvCoordinateTransformationException ex) {
         System.out.println("Unable to inverse this point " +
                            ex.getMessage());
       }
    }
  });
```

Finally, add a listener to the manager. This listener will display the name of the selected city in the appropriate text field.

```
manager.addManagerSelectionListener(new ManagerSelectionListener(){
  public void selectionChanged(ManagerSelectionChangedEvent e) {
    IlvGraphic g = e.getGraphic();
    if (g == null)
      return;
    String name = g.getName();
    if (name != null)
       cityField.setText(name);
    }
  });
```

After you have compiled your sample, the map shown in *Setting a view interactor* should appear on the screen:



Setting a view interactor

Managing units

The management of units is performed by using three classes included in the `ilog.views.maps` package:

- `IlvUnit`
- `IlvAngularUnit` to measure angles.
- `IlvLinearUnit` to measure lengths.

These classes deprecate the former `ilog.views.maps.IlvUnitConverter`, which did not make a clear distinction between units to measure angles and distances.

Units are simply reference systems to measure physical quantity. By convention, a kernel unit is defined for each type of unit. For angles, the kernel units are radians, and for lengths, the kernel units are meters. This allows an easy conversion between any derived units.

Units are attached to each axis of a coordinate system. This means that inside a coordinate system, you can have each ordinate expressed in different coordinates. For example, with geographic coordinate, you can have the x- and y-ordinates expressed in degrees, while the z-ordinate (corresponding for example to the ellipsoid height) is expressed in meters. Usually, in a coordinate system, the same unit is used to measure all physical quantities that are alike (for example, degrees for all angles).

The complete source code of the example can be found in the following file:

<installdir> **`/jviews-maps86/codefragments/srs/src/Sample3.java`**.

Predefined units

The package contains a list of predefined units, to measure both lengths and angles. A predefined unit is referenced by its abbreviation. To access a predefined unit converter, you can use either the static instance of the relevant class, or the static method `GetRegisteredUnit(java.lang.String)`.

See the following tables for a list of the predefined angular and linear units supplied with JViews Maps.

Predefined angular units

Abbreviation	Full Name	ToKernel
rad	Radian	1.0
deg	Degree	PI / 180
grad	Grad	PI / 200

Predefined linear units

Abbreviation	Full Name	ToKernel
km	Kilometer	1000.0
m	Meter	1.0
dm	Decimeter	0.1
cm	Centimeter	0.01
mm	Millimeter	0.0010
kmi	International Nautical Mile	1852.0
in	International Inch	0.0254
ft	International Foot	0.3048
yd	International Yard	0.9144
mi	International Statute Mile	1609.344
fath	International Fathom	1.8288
ch	International Chain	20.1168
link	International Link	0.201168
us-in	U.S. Surveyor’s Inch	0.025400050800101603
us-ft	U.S. Surveyor’s Foot	0.304800609601219
us-yd	U.S. Surveyor’s Yard	0.914401828803658
us-ch	U.S. Surveyor’s Chain	20.11684023368047
us-mi	U.S. Surveyor’s Statute Mile	1609.347218694437
ind-yd	Indian Yard	0.91439523
ind-ft	Indian Foot	0.30479841
ind-ch	Indian Chain	20.11669506

Defining units

The `IlvUnit` class is the superclass of all the units.The JViews Maps package provides two specialized versions of this class:

♦ `IlvLinearUnit`

For units measuring length.

- ♦ `IlvAngularUnit`

  For units measuring angles.

To define a new unit, you can create a new instance of these two classes specifying three parameters:

- ♦ The factor of conversion to kernel units (the number of kernel units necessary to define one unit).
- ♦ The abbreviation for your unit.
- ♦ The full name of the unit.

For example, the following code creates a new instance of `IlvLinearUnit` to convert meters to feet (assuming than 1 ft = 0.3048 meters):

```
IlvLinearUnit unit = new IlvLinearUnit(0.3048,
                                       "ft",
                                       "International foot");
```

Using units

Units can be used for simple conversion between values, or in a coordinate system. Various standard units are defined in the built-in unit classes of the JViews Maps package. These units can be retrieved by using either predefined static members of the classes `IlvLinearUnit` and `IlvAngularUnit` (these are the most used units), or the `GetRegisteredUnit(java.lang.String)` method.

Simple unit conversion

For example, to convert meters to feet you can use either the static definition of feet as follows:

```
IlvLinearUnit unit = IlvLinearUnit.FT;
double meters = 100D;
double feet = unit.fromMeters(meters);
System.out.println("100 m = " + feet + " ft");
// The following is also valid, as the kernel unit for lengths
// is the meter.
double other_feet = unit.fromKernel(meters);
System.out.println("100 m = " + feet + " ft");
```

or the registered version:

```
IlvUnit unit = IlvUnit.GetRegisteredUnit("ft");
double meters = 100D;
double feet = unit.fromKernel(meters);
System.out.println("100 m = " + feet + " ft");
```

For standard conversion of units, the `IlvUnit` class defines the following methods:

- ♦ The `toKernel(double)` method converts from the unit to the kernel unit.
- ♦ The `fromKernel(double)` method converts from kernel unit to the specific unit.

In addition, the `IlvLinearUnit` class (respectively `IlvAngularUnit`) defines the `toMeters (double)` and `fromMeters(double)` methods (respectively `toRadians` and `fromRadians` methods) for applications that need explicit method calls.

Units in coordinate systems

Units are part of the definition of coordinate systems, which means that all the coordinate systems have to be constructed with units for axis. Once the unit is defined for a coordinate system, the transformation factory is able to automatically add subsequent affine transforms to convert coordinates in the transformation path.

Pregenerating tiled images for a thin client

The `IlvManagerTiler` class is a utility class that renders a map, saved to a specified location on the disk at different scale levels, as tiled images. These images can be used to fill the cache of a thin-client server application. Offline tile generation speeds up the initial response time of the server, as it does not need to create and cache the images when clients request them. It increases the server scalability for the same reason.

The first step to generate tiled images for a map is to ensure that the Thin Client Parameters have been set in the map, see *Setting Thin Client Parameters* in JViews Maps *Using the Map Builder*. What the map builder does is that is to add two `IlvNamedProperty` instances to the map. The names of these properties are defined in the IlvMapTileGen `IlvMapTileGeneratorConstants` class as constants:

- ♦ TILE_SIZE_PROPERTY is the name of the property that holds the size of the tiles that will be generated. The size is measure in pixels. An example value is 256 pixels.
- ♦ SCALE_LEVELS_PROPERTY PROPERTY is the name of the property that holds an array of `double` values representing the different scales at which the map should be rendered using tiles.

If these parameters are not set in the map, you must specify them explicitly in the `createTiles()` method call. Then, you must create an `IlvManagerTiler` instance and call the `createTiles(java.lang.String, ilog.views.IlvRect, java.io.File, long)` method with the appropriate parameters using the following code:

```
IlvManagerTiler tiler = new IlvManagerTiler();
long maximumDiskSpace = 10000000; // 10 Mbytes
String mapFilename = "MyMap.ivl";
File outputFolder = new File("output");
tiler.createTiles(mapFilename,null, outputFolder, maximumDiskSpace);
// Null is passed as the region so as not to limit the tiling process.
```

Remember that generating tiles for a map covering a large region, with several scale levels, and without specifying an area of interest, can easily lead to millions of tiles being created. This may require several Gigabytes of memory and in some cases may take a long time to complete.

To avoid this:

- ♦ Specify a reasonable disk space limit. The process stops as soon as the limit is reached.

  and/or

- ♦ Target regions that are likely to be requested by thin clients and only generate images for these regions.

A graphical tool to interactively generate tiles is provided as a code sample in: ***<installdir>*** **`/jviews-maps86/samples/tilegeneration/src/TileGenerator.java`**.

Note: Only map layers with the `thin client background` property set to `true` in their style will be rendered as tiled images. Dynamic layers such as labeling layers should not be marked as background layers, as they are dynamically generated by the servlet.

Integration

Describes how to integrate maps and symbols into an application.

In this section

Overview of integration

To integrate maps and symbols into an application, you need to:

♦ Edit your map using the JViews Map Builder.

♦ Edit your symbol design using Designer for JViews Diagrammer.

♦ Integrate background maps from the JViews Map Builder into the Designer for JViews Diagrammer.

♦ Integrate symbol styles from the Designer for JViews Diagrammer into the JViews Map Builder.

♦ Integrate a JViews Diagrammer project into an application.

You can also use the API and Beans to build an application based on a JViews Diagrammer model and styling, and then add any additional mapping capabilities such as scale display, reprojection, dynamic importation of maps and so on.

Integrating with JViews Diagrammer

Describes the tools available for integrating maps and symbols into applications that use both JViews Maps and JViews Diagrammer.

In this section

Overview

To get the best from the JViews Maps and JViews Diagrammer technologies, you can use two tools:

- The Map Builder, to create dynamic maps (See Using the Map Builder).
- The Designer for JViews Diagrammer, to connect with data and style symbols (See Using the Designer).

You can use either of these tools to display maps and symbols, but you can only design Maps with the Map Builder, and Symbols with the Designer for JViews Diagrammer.

Using symbols and maps in the Designer for JViews Diagrammer

The Designer for JViews Diagrammer provides features to style, import, and export the symbol data model and the rendering mechanism. It also includes an option that gives access to the same kind of graphic objects as the default Basic Symbols in the Map Builder. See "Predefined Renderers" in Developing with the JViews Diagrammer SDK.

The Designer for JViews Diagrammer includes an option that can import any `IVL` file saved by the Map Builder and use it as a background image of the current diagram, (see *The Map Renderer* in Developing with the JViews Diagrammer SDK). You can activate this option at any time and provide it with the name of the `IVL` file.

The Designer for JViews Diagrammer Wizard is a powerful tool that allows you to use maps by selecting a Georeferenced Diagram option when you create a new diagram application.

Integrating a JViews Diagrammer project into an application

A project is an association of a *style sheet* and a *data source* which supplies data. It groups the inputs for a diagram. A project is saved as an XML file with the extension `.idpr` (JViews Diagrammer Project File).

Loading a project file is the recommended way to load a diagram in Java™ because it is the quickest way. The following code sample shows how to load a project into a diagram component using the method `setProject(ilog.views.diagrammer.project.IlvDiagrammerProject, boolean)`.

```
IlvDiagrammer diagrammer = new IlvDiagrammer();
diagrammer.setProject(new IlvDiagrammerProject(
  new URL("file:myproject.idpr"));
//Display the diagram.
```

The project is represented by the `IlvDiagrammerProject` class, which is in the IBM® ILOG® JViews Diagrammer package `ilog.views.diagrammer.project`. When a new project is created, the style sheet and data source are both null.

The following example shows how to create a new project file, set the style sheet and data source, and save the file.

```
IlvDiagrammerProject project = new IlvDiagrammerProject();
project.setStyleSheet(new URL("file:example.css"));
IlvXMLDataSource dataSource = new IlvXMLDataSource();
dataSource.setDataURL(new URL("file:example.xml"));
project.setDataSource(dataSource);
project.write(new URL("file:example.idpr"));
```

Using symbols through the API

Describes symbols and explains how to use them.

In this section

Overview of symbols

Symbols are based on JViews Diagrammer technology.

Symbols are nodes of a JViews Diagrammer data model. They are rendered into a graphical view with styling information contained in a *Cascading Style Sheet*.

See also “Basic Concepts” in Introducing JViews Diagrammer.

Storing symbols

You can use a default SDM Model to store your symbols such as:

```
IlvSDMEngine engine = new IlvSDMEngine();
engine.setGrapher((IlvGrapher)view.getManager());
engine.setReferenceView(view);
```

Integrating symbols into an application

The simplest way to integrate symbols into a map application is to provide a Cascading Style Sheet (see “Styling” in Introducing JViews Diagrammer) that describes the symbol rendering, and then use the API to populate the SDM model, for example:

```
// set the rendering style sheet
engine.setStyleSheets(new String[]{"myfile.css"});
```

You can also use any technique described in “User Interactions” in Introducing JViews Diagrammer to integrate your own symbol data model, possibly through an XML stream or data base connection.

Populating the SDM model

You can populate the SDM model using the API, for example:

```
Object symbolNode = engine.getModel().createNode("symbol");
engine.getModel().addObject(symbolNode, null, null);
...
engine.getModel().setObjectProperty(symbolNode,
 "longitude", new Double(Math.toRadians(44)));
engine.getModel().setObjectProperty(symbolNode,
 "latitude", new Double(Math.toRadians(-105)));
...
```

Creating symbol groups

You can also create a symbol group, and add the newly created node to the group:

```
Object symbolGroup = engine.getModel().createNode("group");
Object symbolNode = engine.getModel().createNode("symbol");
engine.getModel().addObject(symbolGroup, null, null);//Add the group itself.
engine.getModel().addObject(symbolNode, symbolGroup, null);//Add the symbol
to the group
```

Using JViews Maps in SWT applications

The Standard Widget Toolkit (SWT) is the windowing toolkit of the Eclipse® development environment and the Eclipse Rich Client Platform (RCP).

An Eclipse RCP application with JViews Maps is implemented in the same way as a Swing application. Additional considerations are explained in *Using JViews Diagrammer in SWT Applications* in Developing with the JViews Diagrammer SDK.

Installing the JViews runtime plugin

IBM® ILOG® JViews Maps provides jar files in the form of a pre-packaged Eclipse plugin. The name of this package is `ilog.views.eclipse.maps.runtime`.

In order to install the Eclipse plugins, you need to install from the local site as shown below.

For Eclipse 3.3:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates/Find And Install**.
3. In the `Install/Update` dialog box, click **Search for new features to install**.
4. Define a New Local Site with the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`.
5. Select the features you want to install.

For Eclipse 3.4:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates** and select the **Available Software** tab.
3. Add a new local site: Click **Add Site**, then **Local** and specify the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`
4. Select the features you want to install, and press the **Install** button.

In your applications, you need the `ilog.views.eclipse.maps.runtime` plugin and its dependencies:

- `ilog.views.eclipse.maps.runtime`
- `ilog.views.eclipse.diagrammer.runtime` (optional)
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

Map data

The following provides as list of suggested free sources for downloading map data.

DTED0

Coverage	Web link
Worldwide elevation	*http://geoengine.nima.mil/muse-cgi-bin/rast_roam.cgi*

ESRI shape

Coverage	Web link	Alternative web link
Worldwide map of countries	*http://en.wikipedia.org/wiki/Global_Administrative_Unit_Layers_(GAUL)*	*http://www.bluemarblegeo.com/products/worldmapdata.php?op=download* (a lower resolution map)
US time zones	*http://www.nationalatlas.gov/mld/timeznp.html*	
US states	*http://www.nationalatlas.gov/mld/statesp.html*	*http://www.census.gov/geo/www/cob/st2000.html#shp*
US counties	*http://www.nationalatlas.gov/mld/countyp.html*	
US ZIP codes	*http://www.census.gov/geo/www/cob/z32000.html* for 3 digits	*http://www.census.gov/geo/www/cob/z52000.html* for 5 digits

ESRI shape or TIGER/Line

Coverage	Web link
USA	*http://www.census.gov/geo/www/tiger/*

GeoTIFF, JPG or PNG

Coverage	Web link	
Worldwide	*http://www.unearthedoutdoors.net/global_data/true_marble/download*	
Worldwide satellite map from NASA	*http://earthobservatory.nasa.gov/Features/BlueMarble/* with a download mirror at	*http://neo.sci.gsfc.nasa.gov/Search.html*

Coverage	Web link	
	http://mirrors.arsc.edu/nasa/world_500m/	(a lower resolution map)

GTOPO30

Coverage	Web link
Worldwide elevation	*http://edc.usgs.gov/products/elevation/gtopo30/gtopo30.html*

VMAP0

Coverage	Web link
Worldwide	*http://geoengine.nga.mil/geospatial/SW_TOOLS/NIMAMUSE/webinter/vmap0_legend.html*

Other data sources

- *http://www.nationalatlas.gov/atlasftp.html*
- *http://www.census.gov/geo/www/cob/bdy_files.html*
- *http://data.geocomm.com/catalog/US/group21.html*

Index