IBM ILOG JViews Diagrammer V8.6

Developing with the JViews Diagrammer SDK

Table of contents

Introducing the SDK **9**

The diagram component **11**

What is a diagram component 12

The Diagrammer class 13

The data source 15

The style sheets 16

The project 18

Managing the diagram 19

Styling and Data Mapping (SDM) **21**

Overview 22

The SDM engine 24

The SDM data model 26

Renderers 27

The grapher 28

Interactors 29

Composite graphics and symbols **30**

Graph layout **31**

Maps **32**

Developing applications **33**

User interface components 34

Customizing the user interface components 36

Deploying applications...37
Overview...38
Java AWT/Swing application...39
Java applets...40
Java Web Start...41
Web deployment...42
Eclipse/RCP...43
Using the Graphics Framework directly...45
Overview...46
Accessing and creating basic graphic objects...47
Storing graphic objects in layers...48
Organizing graphic objects into graphs...49
Composite graphics in Java code...50
Using and writing data models...51
Overview...53
Deciding your data model strategy...54
Implementing the behavior of data model objects...58
Connecting data sources to the diagram component...60
JavaBeans example...63
The Molecule example...64
The Atom, Bond, and Molecule classes...65
The Molecule model...67
The Phenol Molecule data source...69
Loading the Molecule into the diagram component...71
NonJavaBeans example: abstract model variant...73
The Tree Model example...74
The Swing JTree...75
The TreeSDMModel class...76
The TreeSDMNode class...77
The TreeSDMLink class...80
Loading the data model and style sheet into the diagram component...82
NonJavaBeans example: basic model variant...85
The second Tree Model example...86
The TreeSDMModel2 class...87
The TreeLink class...90
Loading the data model...91
Using a custom data model in the Designer...92
Handling XML data files in Java...93
Reading data...94

Writing data....95
Reading and writing custom XML formats....96
Content on demand....99
About content on demand....100
Concepts....101
Classes involved....104
Using content on demand....106
Content on demand and tiling systems....108
Using CSS syntax in the style sheet....109
Overview....110
The origins of CSS....111
The CSS syntax....113
Style rule....114
Selector....115
Declaration....117
Priority....118
Cascading....119
Inheritance....120
Applying CSS to Java objects....121
Overview....122
The CSS engine....123
The data model....124
CSS recursion....128
Expressions....130
Divergences from CSS2....135
Customizing general nodes in the style sheet....139
Overview....140
Controlling the node’s shape....141
Controlling the node’s skin....143
Controlling the node’s border....145
Controlling the node’s label....146
Controlling the node’s icon....148
Automatic resizing....149
Decorations....150
Customizing general links in the style sheet....151
Controlling the link’s look....152
Obtaining color effects....157
Controlling link decorations....158
Controlling arrows....161
Controlling extra effects....164
Summary of link properties....167

Using and adding renderers....171

About renderers....172

Using renderers in the style sheet....173

Enabling a renderer....174
Customizing a renderer....175
Using rendering properties on objects....176

Predefined renderers....177

Class summary....179
The Coloring renderer....180
The Decoration renderer....182
The Blinking renderer....184
The GrapherPin renderer....187
The GraphLayout renderer....189
The DrillDown renderer....193
The HalfZooming renderer....195
The InfoBalloon renderer....197
The Composite renderer....198
The Interactor renderer....200
The LabelLayout renderer....201
The Legend renderer....202
The LinkLayout renderer....205
The Map renderer....206
The StyleSheet renderer....208
The Map StyleSheet renderer....210
The SubGraph renderer....212
The SwimLanes renderer....214

Adding your own renderer....219

Writing a renderer class....220
Registering a renderer....221
Loading and customizing a renderer....222

The Flag renderer example....223

Overview....224
Header part....225
Bean properties....227
Private methods....228
Overloading methods of the Filter class....230
Integrating the Flag renderer....232
Possible enhancements....234

Configuring renderers in Java code....235

Overview....236
Accessing a renderer....237

Modifying a renderer....238
Setting new renderers....239
Support for renderers in the Designer....240

Using and writing interactors....241

Predefined interactors....242
Subclassing view interactors....244
Writing an object interactor....245
Writing a subclass of IlvObjectInteractor....246
Enabling a custom interactor....247
Connecting interactors to diagrams using listeners....248

Managing dynamic symbols....249

Introducing symbols....251
What is a symbol?....252
The advantages of symbols....253
Using symbols....255
Basic concepts....256
Loading palettes....258
Saving palettes....260
Advanced management of symbols and palettes....261
Palettes....263
Categories....264
Palette symbols....265
Palette parameters....266
Palette events....268
Palette manager....269
Palette manager events....270
Organizing symbols in categories....271
How to use a symbol in CSS....273
Using composite graphics....275
What is a composite graphic object?....276
Building composite nodes in CSS....277
Building composite graphics in Java....284

Printing....285

Using JViews products in Eclipse RCP applications....286

Performance enhancements....291

Global performance improvements....292
Optimizing the performance of JViews Framework....293

Introduction....295
Configuring the Java Virtual Machine....296
Minimizing JAR files....297
Choosing the best graphic object....298
Drawing performance in the view....302
Events and listeners....305
Interactors....308
Antialiasing....309
Transparency....310
Printing....311
Optimizing the performance of symbols....312
Optimizing the performance of diagrams....313
The Overview pane....315
Grids in a diagram....316
Rendering Done mode....317
Composite renderer....318
Load on demand....319
Content on demand....320
Adjusting modes....321
Detail level....322
Better SDM/Java transition....323
Optimizing the performance of dashboards....325
The Overview pane....326
Binary dashboard format....327
Optimizing the performance of Graph Layout....329
Introduction....330
Use layout only when needed....331
Layout of huge graphs....332
Speed of layout algorithms....333
Index....335

Introducing the SDK

Introduces the classes available in the SDK of IBM® ILOG® JViews Diagrammer, and the features they offer for developing your application. The IBM® ILOG® JViews Diagrammer Designer gives you a noncode route through the first phase of development, but you will need to write code to complete and deploy your application. You can choose a code route for all phases.

In this section

The diagram component
Describes the basic functions of the diagram component.

Styling and Data Mapping (SDM)
Describes the SDM package and its subpackages, which are used to define styling and data mapping for the diagram component, but which can also be used to extend its basic functionality.

Composite graphics and symbols
Introduces the Composite package and its subpackages, which provide composite nodes that can be in layers.

Graph layout
Introduces the Graph Layout package and its subpackages, which are used for graph, link and label layout.

Maps
Describes the maps feature which is available if you have purchased the IBM® ILOG® JViews Maps product.

Developing applications
Describes the user interface components provided in IBM® ILOG® JViews Diagrammer and how they may be customized.

Deploying applications
Contains information on deploying your IBM® ILOG® JViews Diagrammer applications.

Using the Graphics Framework directly
Provides pointers to using the lower-level API of JViews Framework in JViews Diagrammer.

The diagram component

Describes the basic functions of the diagram component.

In this section

What is a diagram component
Provides an overview of the diagram component, a panel containing graphic objects that represent your business objects. The diagram component is organized as a graph of nodes and links, and is a JavaBean™ for easy insertion into a graphic editor or monitoring application.

The Diagrammer class
Explains the `IlvDiagrammer` class and its component classes.

The data source
Provides an overview of the two different types of data source: XML data source and JDBC data source.

The style sheets
Explains the style sheet, a `.css` file that defines the way the objects of your data model will be translated to graphic objects.

The project
Describes a project, which is the association of a style sheet with the data source that supplies it data.

Managing the diagram
Describes the different methods of interacting with the diagram, controlling the view, laying out the objects, and editing or printing it.

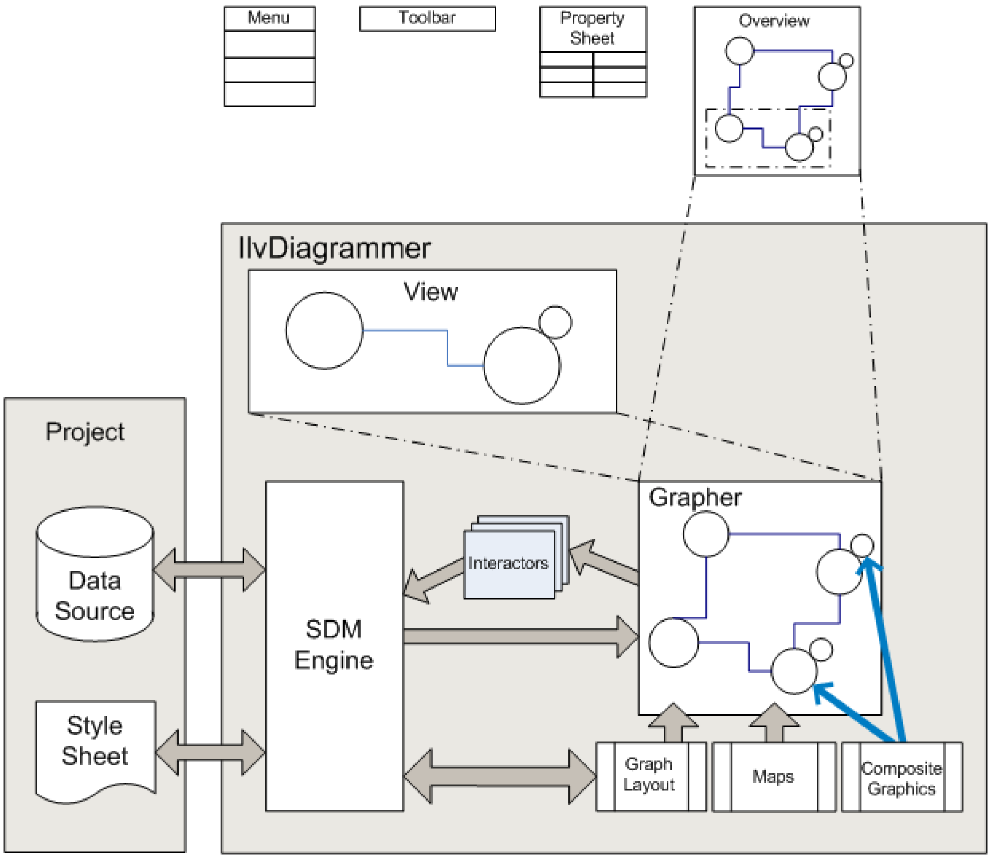
What is a diagram component

The diagram component is a panel containing graphic objects that represent your business objects, organized as a graph of nodes and links. It is a JavaBean™ for easy insertion into a graphic editor or monitoring application.

The diagram component is designed to provide all the basic functionality that most applications will need to display and edit diagrams. Its basic functions are described in this section. Some applications, however, will need to extend the basic functionality, or to use advanced functionality not directly accessible in the diagram component. These advanced features are described in the later sections.

The Diagrammer class

The diagram component is represented by the `IlvDiagrammer` class, which is in the main IBM® ILOG® JViews Diagrammer package, `ilog.views.diagrammer`, see figure *The IlvDiagrammer class relationships*.

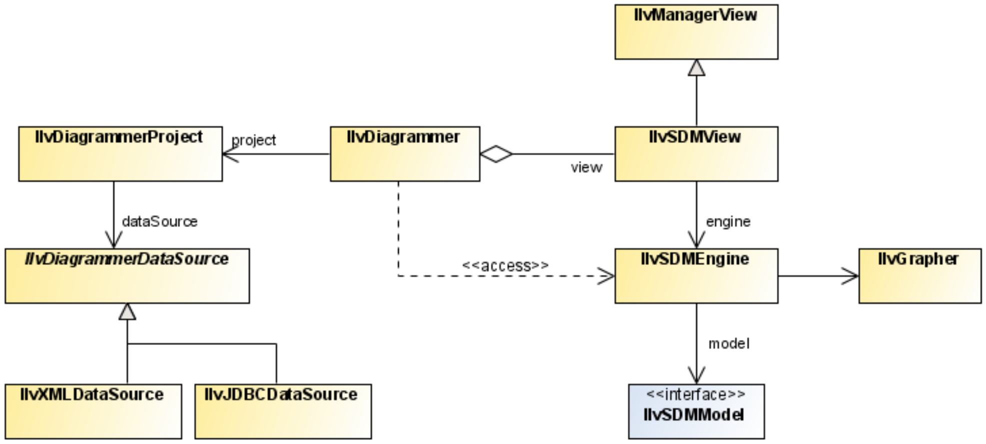


The diagram component

The `IlvDiagrammer` class is a façade that gives easy access to the following underlying facilities:

- The SDM (Stylable Data Mapper) engine in the package `ilog.views.sdm`, which transforms your application data into graphic objects on your screen, based on styling information contained in one or more style sheets.
- A project, which gives access to the first style sheet and the data source for loading your data.
- A grapher, which is a low-level object whose role is to manage the nodes and the links and arrange them as a graph.

- An SDM View (also in the SDM package), which displays the resulting diagram on the screen.
- The Graph Layout package `ilog.views.graphlayout`, which contains advanced algorithms used to place the nodes and to route the links of the graph. Several graph layout algorithms are available, each adapted to a particular type of graph or to a particular application domain. (Graph layout features are available only if you purchased a full JViews Diagrammer license.)
- The Composite Graphics package `ilog.views.graphic.composite`, which lets you define complex graphic objects from basic elements like shapes, text, and icons.
- The Application package `ilog.views.diagrammer.application`, which contains Swing objects such as actions and toolbars that make it very easy to build a complete Swing GUI around a diagram component. A prebuilt application is also available in the Application package.



The IlvDiagrammer class relationships

The data source

The role of the data source is to load the data to display in the diagram, and possibly write back the data if it has been modified.

There are the following predefined types of data source: XML data sources, which read data from an XML file and JDBC data sources, which read data from a database through the JDBC API.

The data source is represented by one of the following classes in the package `ilog.views.diagrammer.datasource`:

- `IlvXMLDataSource`
- `IlvJDBCDataSource`

These classes are subclasses of `IlvDiagrammerDataSource IlvDiagrammerDataSource`.

An XML file (`IlvXMLDataSource`) can be in the required format, which is called *diagram format*, or in a custom format, in which case XSLT files must be supplied for reading and writing.

A database (`IlvJDBCDataSource`) must be JDBC-compliant. SQL is used to read and write data.

Extra types of data source can be defined to read data from other sources.

To set the data source of a diagram component, you can use the `setDataSource(ilog.views.diagrammer.datasource.IlvDiagrammerDataSource)` method. The following code example shows how to use this method.

Setting an XML data source for the diagram component

```
IlvXMLDataSource dataSource = new IlvXMLDataSource();
dataSource.setDataURL(new URL("file:example.xml"));
diagrammer.setDataSource(dataSource);
```

Alternatively, you can set the data source and the style sheet at the same time by loading a project file (see *The project*).

The style sheets

A style sheet is a `.css` file. It is referenced by a URL. The style sheet can load other style sheets using the `import` statement. A style sheet controls the mapping of data to a graphic representation through style rules conforming to the CSS level 2 syntax. It defines the way the objects of your data model will be translated to graphic objects.

The SDM engine accepts *cascading style sheets* (CSS). Cascading style sheets can be used to define several levels of customization. For example, different style sheets could be defined for a whole company, a group, and an individual user.

You can either write a style sheet by hand or create it using the IBM® ILOG® JViews Diagrammer Designer. For example, if your final application loads a custom data model, you can still use the Designer to define the style of your diagram. All you need to do is to create a sample XML file containing objects with the same properties as your real data. Then, use the Designer to define the graphic style of your nodes and links. When you save your project, the Designer will create three files: a project file (`.idpr`), an XML file containing your sample data (`.xml`), and a CSS file containing all the styling information (`.css`). In your final application, you will only use the CSS file as explained next, and load your real data instead of the sample data used in the Designer.

Style sheets are usually suitable only for a particular data model. For example, if your data model describes a workflow process, it will contain such objects as activities, transitions between activities, and participants that carry on the activities. A style sheet for such a data model will have rules that match these particular object types and their properties to define the graphic look of the objects. So, a workflow style sheet will probably not work for another data model, because the objects will not have the same types and properties.

To assign a style sheet to the diagram component in Java™ , call the method `setStyleSheet (java.net.URL)`. This method takes an array of `String` with each element representing one style sheet. The array simulates the cascading of style sheets, where the last style sheets have a higher priority than the previous ones. The following code example shows an example with only one style sheet.

Setting the style sheet of a diagram component

```
String[] styleSheet = new String[] { "file:example.css" };
engine.setStyleSheets(styleSheet);
```

Alternatively, you can set the data source and the style sheet at the same time by loading a project file (see *The project*).

Style sheets can be cascaded, that is, concatenated with later style sheets having a higher priority than earlier ones. To cascade style sheets, use the `addStyleSheet(java.net.URL)` method, see the following code example.

Adding a cascaded style sheet to a diagram component

```
Diagrammer.addStyleSheet(new URL("file:cascaded.css"));
```

For details of the syntax of the style sheet and how to use it to customize nodes and links, see *Using CSS syntax in the style sheet*.

As well as the diagram component itself, you can customize its associated Swing user interface components through the style sheet. See *User interface components* for more details.

The project

The 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` (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 example shows how to load a project into a diagram component using the method `setProject(ilog.views.diagrammer.project.IlvDiagrammerProject, boolean)`.

Loading a project file into a diagram component

```
IlvDiagrammer diagrammer = new IlvDiagrammer();
diagrammer.setProject(new IlvDiagrammerProject(
  new URL("file:myproject.idpr")), true);
//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 code example shows how to create a new project file, set the style sheet and data source, and save the file.

Creating a project 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"));
```

Managing the diagram

You can interact with the diagram, control the view, edit the diagram, apply a layout to the objects, print the diagram by using the appropriate methods as described below.

Interacting with the diagram

The following methods determine which interactions are available to users of the diagram component, that is, what happens when a user clicks:

- `setZoomMode(boolean)`: the view can be zoomed in or out
- `setPanMode(boolean)`: the view can be moved in any direction
- `setSelectMode(boolean)`: objects can be selected/deselected
- `setEditLabelMode(boolean)`: the label of a new object can be edited when it is created

Controlling the view

For scrolling, you can use the `setScrollable(boolean)` method to control the visibility of the diagram scrollbars.

For zooming, you can use the following methods:

- The `zoomIn()` and `zoomOut()` methods to zoom the diagram in or out.
- The `fitToContents()` method to make the whole diagram visible.
- The `resetZoom()` method to reset the zoom factor to 1:1.

Editing the diagram

The `IlvDiagrammer` class provides full support for editing, that is, you have all the methods necessary to create an editor based on the diagram component.:

- For selecting objects, the `setSelected(java.lang.Object, boolean)`, `isSelected(java.lang.Object)`, `selectAll()`, `deselectAll()` methods control the set of selected objects.
- For editing objects, the `cut()`, `copy()`, `paste()`, `delete()`, `duplicate()`, `addObject(java.lang.Object, java.lang.Object)`, `removeObject(java.lang.Object)`, `group()`, `ungroup()`, methods and others let you edit the nodes and links.
- For editing object properties, the `setObjectProperty(java.lang.Object, java.lang.String, java.lang.Object)` and `getObjectProperty(java.lang.Object, java.lang.String)` methods let you edit property values.

Laying out the objects

The layoutAllNodes() method applies the graph layout algorithm configured in the style sheet to all the nodes of the diagram. The layoutSelectedNodes() method applies the layout only to the selected nodes.

The layoutLinks() method applies the link layout algorithm configured in the style sheet to all the links of the diagram

The setAutomaticLinkLayout(boolean) and setAutomaticNodeLayout(boolean) methods cause the link (or node) layout to be applied automatically whenever the diagram is modified.

Printing the diagram

The methods print(boolean, boolean, ilog.views.diagrammer.IlvDiagrammer.PrinterExceptionHandler),printToBitmap(java.io.File),pageSetup(),printPreview(), setPrintArea() let you print all or part of the diagram.

Advanced methods

The methods of the IlvDiagrammer class let you achieve the most common tasks, but you will often need to extend the diagram component, or access advanced functionality not available directly in the IlvDiagrammer class.

For this reason, methods such as getEngine(), getView(), getSelectInteractor(), getPrintingController() give you access to the low-level objects that implement advanced features.

Styling and Data Mapping (SDM)

Describes the SDM package and its subpackages, which are used to define styling and data mapping for the diagram component, but which can also be used to extend its basic functionality.

In this section

Overview
Presents an overview of the SDM package and its subpackages.

The SDM engine
Describes the SDM engine, which controls the data-to-graphics mapping.

The SDM data model
Describes the SDM data model, which is the interface that tells the SDM engine how to get the data to be displayed.

Renderers
Briefly describes renderers, which apply the styles specified in the style sheets.

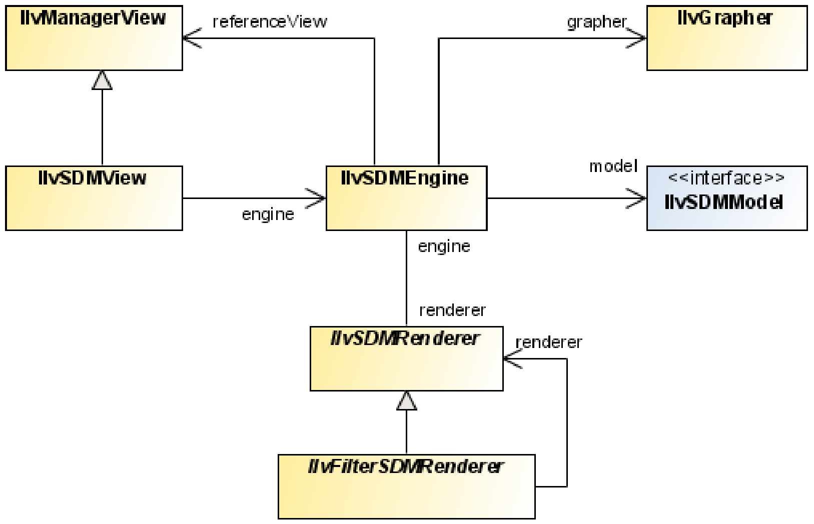
The grapher
Defines graphers, which can store graphic objects including general nodes, composite nodes, and general links. They provide the infrastructure that is minimally necessary to draw a graph.

Interactors
Describes interactors, which support the interaction between an end user and a diagram.

Overview

The SDM package `ilog.views.sdm` and its subpackages contain the styling and data mapping (SDM) classes. These classes are usually used internally by the diagram component, but you may need to use these classes directly to extend its basic functionality.

The main entry point to the package is the class `IlvSDMEngine`. See the following diagram.



The SDM class relationships

The SDM subpackages are:

- `ilog.views.sdm.event`: The event-related classes of SDM.
- `ilog.views.sdm.graphic`: The specialized SDM graphic objects such as graphic factories and the general node.
- `ilog.views.sdm.interactor`: The interactor-related classes.
- `ilog.views.sdm.model`: The predefined classes for data models.
- `ilog.views.sdm.renderer` and `ilog.views.sdm.renderer.animation`: The classes that define the renderers available for styling. Other renderer subpackages are:
  - `ilog.views.sdm.renderer.graphlayout`: The classes that configure the Graph Layout capability, if you have purchased a full JViews Diagrammer license.
  - `ilog.views.sdm.renderer.maps`: The classes that provide access to the Maps product, if you have purchased it.
- `ilog.views.sdm.servlet`: A specialized servlet that gives easy access to styling and data mapping in thin-client mode.
- `ilog.views.sdm.swing`: The JFC/Swing-based components for building a full GUI.

- `ilog.views.sdm.util`: Utility classes.

The SDM engine

The SDM engine controls the data-to-graphics mapping.

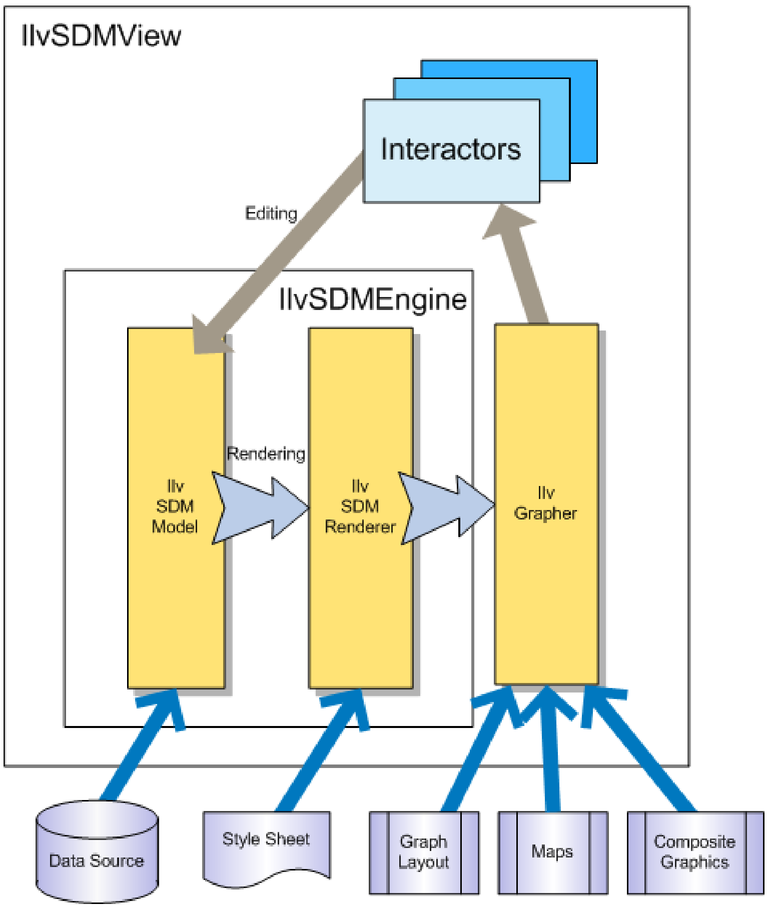
There are four key elements in the data-to-graphics mapping process:

♦ A data model that interfaces to the data to display or edit. This data model is completely independent of the GUI, and refers only to the business objects of your application.

♦ Renderers that style the diagram as a whole and the graphic objects in it.

♦ A grapher in which the graphic objects representing the data model are created.

♦ Interactors that permit user actions on graphic objects.



SDM data-to-graphics mapping

As shown in the above figure, the mapping between the data model and the graphical representation is bidirectional:

1. Data model to graphics: the *rendering* process is controlled by the style sheet, which lets you tell the SDM engine how you want each particular kind of data object to be displayed in the grapher. The rendering process is performed by specialized renderers.

    ♦ When the data model is loaded, the SDM engine explores it and creates graphic objects representing the nodes and links defined by the data model in the grapher.

    ♦ When the state of an object in the data model changes, the SDM engine updates the graphic object representing the modified data object. The object state may change due to an external application event or after a direct edit of an object property by the user.

2. Graphics to data model: the *editing* process relies on built-in editing facilities that act directly on the underlying data model. The actions in an editing application are implemented by interactors. For example:

    ♦ When the user moves a graphic object (for example, in an editor), the SDM engine updates the geometric properties of the object in the data model.

    ♦ When the user expands or collapses a node (for example, in a navigation application), the SDM engine updates the expand/collapse status of the object in the data model.

To access the SDM engine associated with a diagram component, use the `getEngine()` method. Similarly, to access the SDM view associated with a diagram component, use the `getView()` method. The following code example shows the use of these methods.

Getting the SDM engine and the SDM view

```
IlvSDMEngine engine = diagrammer.getEngine();
IlvSDMView view = diagrammer.getView();
```

You can also choose not to use the `IlvDiagrammer` class at all. Instead, you can create an SDM engine (`IlvSDMEngine` instance) and an SDM view (`IlvSDMView` instance) yourself.

The SDM data model

The SDM data model is the interface that tells the SDM engine how to get the data to be displayed. The SDM data model is an abstract description of a set of nodes and links between nodes. Nodes and links have a user-defined type (also called the “tag”), and a set of named properties.

A default data model is supplied as class `IlvDefaultSDMModel` and you can also develop your own data model.

The classes provided for data models are in the package `ilog.views.sdm.model`.

For a discussion of the available data model classes and advice on writing your own, see *Using and writing data models*.

Renderers

Renderers apply the styles specified in the style sheets. They have global properties for styling the diagram and per-object properties for styling the objects in the diagram.

The predefined renderers for styling graphic objects are described in *Using and adding renderers*.

That section also provides an example of adding your own renderer.

The grapher

Graphers can store graphic objects including general nodes, composite nodes, and general links. They provide the infrastructure that is minimally necessary to draw a graph.

A grapher (`IlvGrapher` instance) is a JViews Framework object created by the `IlvDiagrammer` instance. You may need to access the grapher to modify the graphic objects of the diagram directly. The following code example shows how to access the grapher.

Accessing the grapher of a diagram component

```
IlvGrapher = diagrammer.getEngine().getGrapher();
```

For more details on graphers, see Graphers in *The Essential JViews Framework*.

Interactors

Interactors support the interaction between an end user and a diagram. Common requirements are for zoom, pan, select, and object creation functions.

The predefined interactors are described in *Using and writing interactors*.

That section also provides an example of adding your own interactor.

Composite graphics and symbols

The Composite package (`ilog.views.graphic.composite`) and its subpackages in JViews Framework contain the Composite Graphics classes.

The main entry point to the package is the class `IlvCompositeGraphic`.

In IBM® ILOG® JViews Diagrammer, there is a specialized subclass, `IlvSDMCompositeNode`, which provides composite nodes that can be in layers.

The Composite Graphics subpackages include:

- `ilog.views.graphic.composite.layout`: The layout managers for child elements of composite graphics. There are three possible Layout Manager classes:
    - `IlvAttachmentLayout`
    - `IlvCenteredLayout`
    - `IlvStackerLayout`

For a description of composite graphics in IBM® ILOG® JViews Diagrammer and an example, see *Using composite graphics*.

A symbol is an abstraction of composite graphic that can be used only through style sheets, see *Managing dynamic symbols*.

Graph layout

The Graph Layout package `ilog.views.graphlayout` and its subpackages contain the classes for graph layouts, and the related classes for link layout and label layout.

Each subpackage represents a layout algorithm or the property editor for the Bean properties of a layout algorithm.

For a list of the layouts available in IBM® ILOG® JViews Diagrammer and various examples, see Layout algorithms.

Maps

IBM® ILOG® JViews Diagrammer allows you to load an IVL file and this can be a map. If the IVL file contains projection data, IBM® ILOG® JViews Diagrammer can then place the nodes according to their latitude and longitude properties.

The package `ilog.views.sdm.renderer.maps` gives access to IBM® ILOG® JViews Maps for developing your own maps, if you have purchased this product.

The rendering of maps in a diagram is done by the Map renderer, `IlvMapRenderer` .

Optionally, the zooming policy of the nodes displayed on a map can be controlled by the HalfZooming renderer, `IlvHalfZoomingRenderer`. which allows you to specify limits in zooming in or out.

If you use a map, your data model should contain latitude and longitude attributes. These attributes can have any names because you can map them to the names `latitude` and `longitude`. For details of the formats for latitude and longitude, see *The Map renderer*.

Refer to the JViews Maps documentation for more information on the integration of JViews Diagrammer with JViews Maps.

Developing applications

Describes the user interface components provided in IBM® ILOG® JViews Diagrammer and how they may be customized.

In this section

User interface components
Describes the ready-made user interface components of IBM® ILOG® JViews Diagrammer that help you develop your applications.

Customizing the user interface components
Explains how the user interface components can be customized.

User interface components

IBM® ILOG® JViews Diagrammer comes with a set of classes designed to ease the development of Swing GUIs containing one or more `IlvDiagrammer` objects. These classes are contained in the package `ilog.views.diagrammer.application`.

Actions

The class `IlvDiagrammerAction` is a base class for Swing Actions that act on an `IlvDiagrammer` object. A set of predefined actions that call the main `IlvDiagrammer` methods is provided.

Toolbars

The class `IlvDiagrammerToolBar` is a Swing JToolBar that accepts a set of `IlvDiagrammerAction` instances. The subclasses `IlvDiagrammerViewBar`, `IlvDiagrammerEditBar` have predefined actions to control and edit the diagram.

The class `IlvDiagrammerPaletteBar` is designed to be used for editing diagrams. You can populate it with actions that create new nodes and links according to a set of sample objects.

Menus

The class `IlvDiagrammerMenu` is a Swing JMenu that accepts a set of `IlvDiagrammerAction` instances. The subclasses `IlvDiagrammerFileMenu`, `IlvDiagrammerEditMenu`, `IlvDiagrammerOptionsMenu` and `IlvDiagrammerHelpMenu` have predefined actions to control and edit the diagram, and to set options and give access to Help for the application. The class `IlvDiagrammerMenuBar` is a complete predefined menu bar containing all the predefined menus.

Overview

The class `IlvDiagrammerOverview` displays a reduced view of a diagram. You can use it with the zoom facility of a diagram component to control which part of a large diagram is visible.

Tree

The class `IlvDiagrammerTree` is a Swing JTree that displays the nodes and links contained in the diagram’s data model. The tree is an alternative way to view and select the objects in the diagram.

Property sheet

The class `IlvDiagrammerPropertySheet` is a Swing JTable that displays the properties of the selected object of the diagram. Use the property sheet to view and edit the properties of the objects in the data model.

Table

The class `IlvDiagrammerTable` is a Swing JTable that displays the properties of all the objects in the diagram. Use it to have a global view of the data model.

The Application class

The class `IlvDiagrammerApplication` is a complete Swing application that is built using the components in the Application package. This class lets you view (and optionally edit) one or more diagrams. The following code example shows how to launch the application by invoking the `java` command on this class.

Launching a JViews Diagrammer application

```
java ilog.views.diagrammer.application.IlvDiagrammerApplication
```

The CLASSPATH environment variable must contain the JAR files for the IBM® ILOG® JViews Diagrammer product and the JViews Framework package.

A JViews Diagrammer application can also be launched as an applet: specify `IlvDiagrammerApplication` as the applet class in the HTML applet tag.

Customizing the user interface components

You can customize all the JViews Diagrammer user interface components through the style sheet. To enable styling of the user interface components, set the `IlvDiagrammer` property `styleApplicationComponents` to `true`.

The following code example shows how to customize a tree, a property sheet, and a table.

Customizing user interface components

```
Diagrammer {
  styleApplicationComponents : "true"; // enable styling of
                                          // components.
}
DiagrammerTree {
  rootLabel : "My Graph"; // set label of root item in tree
}
node:DiagrammerTree {
  text : "@name"; // use "name" property as label of items in tree
}
node:DiagrammerPropertySheet:type {
  editable : "false"; // set "type"  property for all nodes read-only
}
node.activity:DiagrammerTable {
  background : "yellow"; // set yellow background for "activity" nodes
                            // in table
}
```

Deploying applications

Contains information on deploying your IBM® ILOG® JViews Diagrammer applications.

In this section

Overview
Provides an overview of how to deploy your IBM® ILOG® JViews Diagrammer applications.

Java AWT/Swing application
Describes deploying your applications using Java™ AWT/Swing.

Java applets
Describes how to deploy your applications as Java™ applets.

Java Web Start
Describes deploying your applications using Java™ Web Start.

Web deployment
Describes how to deploy your applications for the Web.

Eclipse/RCP
Describes deploying your applications using Eclipse™ /RCP.

Overview

JViews Diagrammer offers a set of components, classes and APIs making it possible to build all sorts of graphical displays. The core Graphics Framework provides interactive vector-graphics primitives that will have their own life cycle and can be displayed in a variety of containers, including Swing or servlets. With Cascading Style Sheets (CSS) and SDM, the creation of such graphic objects is automated and sophisticated displays are built more easily. Most of the work needed to create specific visual representations is the same, whether you decide to deploy your application with rich or thin-client technologies. What is really important is to make sure that:

♦ You can control the integration of your visual components with the right containers.

♦ You have the kind of interactions you need for the application.

♦ You have the ability to package and deploy your work efficiently.

This section summarizes the main characteristics and tips of each deployment strategy.

Java AWT/Swing application

This is the most traditional way to deploy JViews applications. All graphical styles and interactions are available, and there are very few limitations. Traditionally, applications are deployed using the `IlvDiagrammer` class (derived from a `JComponent`) or classes of the `ilog.views.diagrammer.application` package. Note that the `IlvDiagrammerApplication` class (derived from `JApplet`) is a top-level container that allows you to create either applications, applets, or Java™ Web Start applications.

See Writing an application in *Using the Designer*.

Java applets

Basically, an applet is a Java™ application (a rich client) running inside a Web browser. The main noticeable differences between applets and applications are the following:

- ♦ Applets do not have the same kind of top-level container (like a `JFrame`, for example). Your Java code is not responsible for creating the main window, and your applets have to live within the limits of the provided `ContentPane`.
- ♦ The code of the applet is usually invoked by a Web page downloaded from a remote Web site and executed on demand.
- ♦ Unless your application is signed or your Java security policy has been extended, your applets have limited access to local resources, such as files, printers, and so on. This requires particular attention for dealing with security exceptions.
- ♦ Most of the time applet classes and resources need to be packaged as JAR files and need to have a very clean access to their data, such as images or resource files.

Like other Java programs, a JViews application needs some particular attention to packaging and use as an applet. If possible, we recommend you use the `IlvDiagrammerApplication` class, which is already able to manage applets or applications. Most samples available in the JViews Diagrammer distribution are packaged so they can work transparently as applets or applications. If you decide to use lower-level containers, such as `IlvDiagrammer`, you will need to take more care in the integration with the top-level containers.

Creating an applet

The `IlvDiagrammer` instance and its associated beans can be used in applets in exactly the same way as in Swing applications. The only limitations are the restricted permissions in applets, for example, it is not possible to save data files from an applet.

The `IlvDiagrammerApplication` class extends the Swing `JApplet` class, so it can be used directly in an HTML applet tag. You can pass the command-line parameters as applet parameters: the names of the parameters are the same, without the "-" prefix.

For an example of an applet application, see the Applet sample, which is available in the directory **<installdir>/jviews-diagrammer86/codefragments/applet**.

Java Web Start

A Java™ Web Start (JWS) application can be seen as an applet running outside a browser. It allows you to have rich applications automatically downloaded, with great user experience. Except for careful packaging and for the same security issues mentioned for applets, there are no particular limitations for creating JWS applications with JViews Diagrammer.

Web deployment

Creating applications for the Web is substantially different from the rich-client approach. It implies that most of the work is done on a remote server which sends ready-to-use images and scripts to a Web client (understand an Internet browser). There are many variants but the main idea is to receive and process HTTP requests on a server and, in return, to produce the right image to be sent back to the browser. Even if there are many architectural differences, the following aspects can be shared between rich and thin clients:

- Styling: responsible for transforming data into graphics, the styling rules can be the same for rich and thin-client applications. For example, CSS files, symbols and Designer projects can be used for both scenarios.
- Data model and connection with a data source: are the same and can benefit from the same code for both deployment strategies.
- Graphical content: usually displayed in the background of a panel or as parts of a dynamic symbol; graphical objects can be reused in both cases.

The main differences reside in the way the application behaves and how it is packaged. Thin clients are usually driven by transactions and only partially support rich interactions. JViews Diagrammer thin clients are based on the JavaServer™ Faces (JSF) and DHTML technologies. Web pages containing visual components are created as (JSP) pages. The JSF-based components (also known as JViews Diagrammer Faces) provide the following services:

- Display graphical content managed by Diagrammer classes and APIs. Data connection and styling are preserved.
- Provide interactive views and overviews.
- Manage local interactions, such as pan, zoom, selection, popup menus.
- Minimize roundtrips when possible.
- Extensions can be written for extending both server-side and client-side components.

The use of these JSF and DHTML components is described in Building Web applications.

Eclipse/RCP

It is possible to deploy graphic components created with JViews Diagrammer as Eclipse™ or RCP (Eclipse Rich Client Platform) plugins. As with the thin client approach, graphical contents, styling, and connections to data sources can be reused under Eclipse. The main challenge is to integrate graphical panels and manage interactions with the Eclipse framework. Since Eclipse 3.0, making Swing-based code work within an Eclipse application is possible with the SWT-AWT bridge. Located in the `org.eclipse.swt.awt` package, this bridge offers a very simple interface between Eclipse and Swing widgets. Once a Swing container has been created inside an SWT widget, all AWT/Java 2D™ primitives can be invoked. Note that this bridge only works with JDK 1.5 and higher. In the `samples` directory, you will find examples of such integration. The source code is provided, so you can make the integration package evolve with your own requirements.

JViews Diagrammer also provides its own bridge, `IlvSwingControl`, to display diagrams and dashboards inside Eclipse or RCP. For details, see *Using JViews products in Eclipse RCP applications*.

Using the Graphics Framework directly

Provides pointers to using the lower-level API of JViews Framework in JViews Diagrammer.

In this section

Overview
Provides an overview of the lower level features that you can access through the API of JViews Framework.

Accessing and creating basic graphic objects
Provides information on the basic graphic objects provided in the JViews Framework, and how to access and customize them.

Storing graphic objects in layers
Describes the class that stores graphic objects in layers.

Organizing graphic objects into graphs
Describes the class that organizes graphic objects into graphs.

Composite graphics in Java code
Describes the class for composite graphics.

Overview

Instead of, or as well as, using the high-level JViews Diagrammer API, you can make use of the lower-level API of JViews Framework.

A lower-level approach is appropriate only if you need to access low-level features directly or to create objects as subclasses of existing ones. In general, you should not need to use lower-level JavaBeans™ for GUI components or interactors; instead you are recommended to use the supplied JViews Diagrammer JavaBeans, see JViews Diagrammer classes available as beans in *Using the Designer*.

This section points you to the documentation available on each of the following lower-level features:

♦ Basic graphic objects (rectangles, arcs, ellipses, and so on)

♦ Managers and their layers, for storing graphic objects and determining the display priority (which objects appear in front of others)

♦ Graphers, for organizing graphic objects into graphs of nodes and links

♦ Composite graphics, for building more complex graphical representations than those available with basic graphic objects

If you would like to read an introduction to JViews Framework in general or make use of a tutorial, see Introducing IBM® ILOG® JViews Framework in *The Essential JViews Framework* and Getting started with JViews Framework in *The Essential JViews Framework* respectively.

More advanced lower-level features, such as nested managers and graphers, and link shapes and crossings, are described in Advanced Features of JViews Framework.

Accessing and creating basic graphic objects

The classes that represent basic graphic objects are listed in Graphic objects in *The Essential JViews Framework*, which also gives information on how to use the supplied objects.

You can customize the supplied graphic objects in terms of colors, dimensions, and other properties. If you need to assemble an object constructed from several basic objects, consider using composite graphics instead, see *Composite graphics in Java code*.

You can also use compiled symbols generated by the Symbol Compiler from the palette symbols designed with the Symbol Editor. For details, see Using compiled symbols at the Graphic Framework level.

If you require a new graphic object that can be implemented by specializing a supplied graphic object, or a completely new graphic object, you can create the required object by subclassing. For an example, see Creating a new graphic object class in *The Essential JViews Framework*.

Storing graphic objects in layers

The class that stores graphic objects in layers is `IlvManager`. It is compliant with the JavaBeans™ standard.

For information about the use of this class and related classes and interfaces, see Managers in *The Essential JViews Framework*.

Organizing graphic objects into graphs

The class that organizes nodes and links into graphs is `IlvGrapher`. It is compliant with the JavaBeans™ standard.

For information about the use of this class and the classes and interfaces related to nodes and links, see Graphers in *The Essential JViews Framework*.

Composite graphics in Java code

The base class for composite graphics is `IlvCompositeGraphic`.

The composite graphics facility allows you to combine basic graphic objects or existing composite graphics as elements of a composite graphic. At a higher level, you can do this in CSS, as described in *Using composite graphics*; at a lower level, you can do the same in Java™ code, see Composite Graphics in *The Essential JViews Framework*.

Using and writing data models

Explains the data model classes available and how to write your own data model depending on whether you want to load existing data or not and, if so, from which kind of data source.

In this section

Overview
Provides an overview of the concept of data model.

Deciding your data model strategy
Describes how to decide what your data model strategy should be, depending on the characteristics of your existing business data and your requirements.

Implementing the behavior of data model objects
Presents the methods used to implement the behavior of data model objects.

Connecting data sources to the diagram component
Lists the ways to connect data sources to the diagram component.

JavaBeans example
Shows how to display a set of JavaBeans™ using IBM® ILOG® JViews Diagrammer.

NonJavaBeans example: abstract model variant
Describes how to display a set of Java™ objects that are not JavaBeans™ by subclassing the abstract class `IlvAbstractSDMModel`.

NonJavaBeans example: basic model variant
Describes how to display a set of Java™ objects that are not JavaBeans™ by subclassing the basic class `IlvBasicSDMModel`.

Using a custom data model in the Designer
Describes how to use a custom data model in the Designer, and what limitations apply if you do.

Handling XML data files in Java
Explains how the class `IlvSDMEngine` allows you to read, write, and transform XML data.

Content on demand
Describes the content on demand feature.

Overview

The term data model in IBM® ILOG® JViews Diagrammer refers to an interface that defines methods to access a hierarchy of objects with properties. This does not necessarily imply that business object types are represented by Java™ classes. If you have Java classes that represent your business object types, you can implement the data model interface on top of these; if you do not already have such Java classes, you can implement the data model interface without them.

IBM® ILOG® JViews Diagrammer uses the data model for information about all the business objects and creates graphic objects according to the model object states. Note that one model object is represented by at most one graphic object.

Implementing a data model to interface with existing business objects does not mean that you have to duplicate your data. You can implement an appropriate data model directly on top of your existing data objects. To do this, you must write a class that implements the `IlvSDMModel` interface. You can use the existing classes in the package `ilog.views.sdm.model` as base classes.

Deciding your data model strategy

First you should decide which classes in the `model` package are appropriate, depending on the characteristics of your existing business data and your requirements.

There are five levels of data model implementation, each appropriate for certain situations:

1. No use of existing data.

   There are three possible situations:

   ♦ You do not have existing data that you want to display and you will create your diagram(s) from scratch.

   ♦ You have existing data, but you want to duplicate it before displaying it, and it does not matter how the data will be represented in memory.

   ♦ You have exported your data to an XML file, and you want to read in the XML data.

   In these cases, use the default SDM model implementation, `IlvDefaultSDMModel`.

   The nodes and the links of the default model are instances of the classes `IlvDefaultSDMNode` and `IlvDefaultSDMLink`. These objects have an arbitrary set of user-defined properties, so the default model can represent all kinds of objects.

2. Display existing data in a database.

   Your data is stored in a database, and you want to display it but not modify it directly in the diagram component.

   The `model` package provides a specific SDM model for this case: `IlvJDBCSDMModel`. This model uses the JDBC™ API to access the database. All you need to do is configure the model with the database URL, the names of the tables containing the nodes and links, and a few other parameters.

3. Existing JavaBeans™ .

   You have existing data stored in classes which conform to the JavaBeans standard: there are accessor methods (`get` and `set`) for each Bean property.

   In this case, use the class `IlvJavaBeanSDMModel`. This model uses your JavaBeans as the nodes and links of a graph, so there is no duplication or additional memory consumption. The properties of nodes and links are read and written using the JavaBeans API. All you have to do is supply the set of JavaBeans that will make up your graph. For an example, see *JavaBeans example*.

4. Existing Java™ classes that are not JavaBeans or existing data but no Java classes.

   For example, you already have an implementation of a Swing JTree model that represents your data and you want to display the contents of the tree without duplicating data.

   One of the following applies:

   ♦ You have existing data represented by Java classes that are not JavaBeans.

   ♦ You want to display data that is not already represented in memory by Java classes.

   In such cases, there are two possible approaches:

- Use the class `IlvAbstractSDMModel`, and wrap your objects into new objects that implement the `IlvSDMNode` and `IlvSDMLink` interfaces. This requires an extra object allocation because of the indirection, but allows you to make use of the predefined interfaces for nodes and links. For an example, see *NonJavaBeans example: abstract model variant*.
- Subclass the basic model, `IlvBasicSDMModel`, and implement the methods that list the objects and retrieve the properties of each object. This requires more work but is more open, and may be preferable if all objects already exist. For an example, see *NonJavaBeans example: basic model variant*.

The following table summarizes the strategies described in this section.

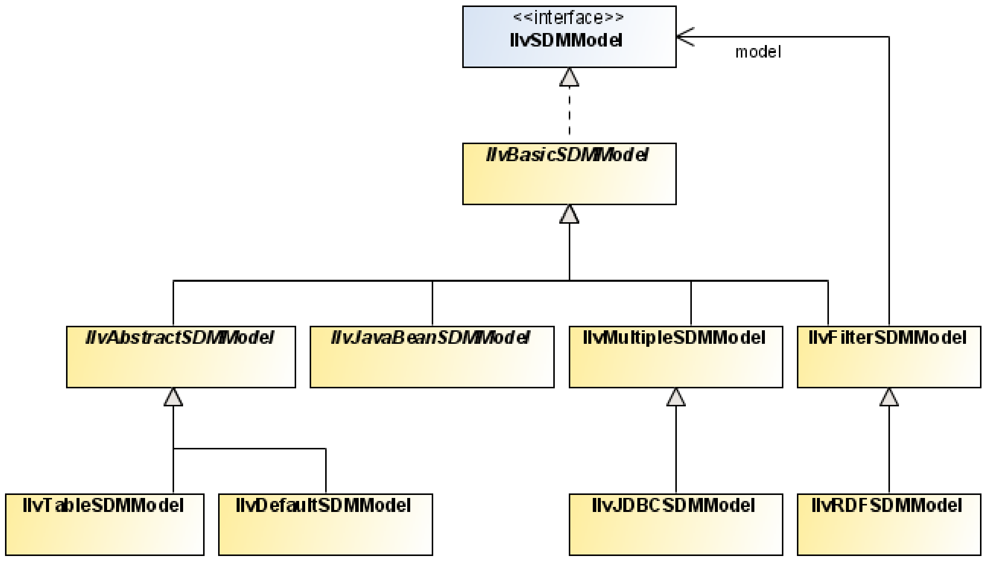
Summary of data model strategies

Data Characteristics	Recommended Class
New or from XML	`IlvDefaultSDMModel`
Database (display only)	`IlvJDBCSDMModel`
JavaBeans	`IlvJavaBeanSDMModel`
Java classes or other existing objects (for example, Swing)	`IlvAbstractSDMModel`
	`IlvBasicSDMModel`

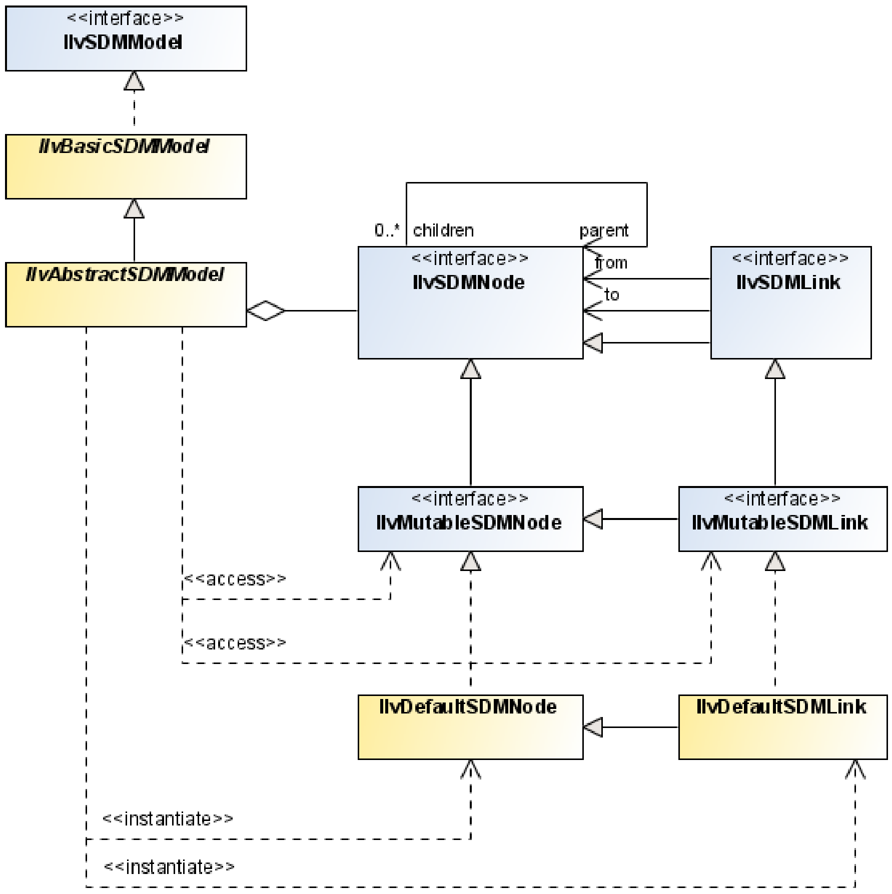
If the graphic objects are to be moved interactively, SDM tries to store the new object locations as x,y properties in the model. However, the model can refuse to set the x,y properties, for example when it is a read-only model. To cope with such a case, the SDM library provides a metadata system that manages new properties as if they were stored in the true model.

To enable metadata, call `ilog.views.sdm.IlvSDMEngine.setMetadataEnabled(true)`. Metadata is saved only in XML files. Other persistant mechanisms are responsible for saving metadata if they need to.

The default model does not need metadata.



The SDM Model class relationships



The abstract SDM Model class relationships

Implementing the behavior of data model objects

The methods that implement the possible behavior of data model objects are supplied as the `IlvSDMModel` interface. Various supplied data model classes implement different subsets of these methods.

The methods in the interface recognize links as distinct from other objects and allow for parent-child relationships to implement a hierarchical model. Each data model object has a unique identifier (ID), a nonunique tag, and various properties (attributes). For links, the From and To properties store the end points. The monitoring of changes is available through a listener mechanism.

Methods for navigating among nodes and links

The three following methods navigate the model object tree.

- ♦ `getObjects()` returns the enumeration of all top-level objects.
- ♦ `getChildren(java.lang.Object)` returns all the children of the given model object.
- ♦ `getParent(java.lang.Object)` returns the parent of the given object in the model tree, or `null` if the object is at the top level.

When a model object is a link, the following methods apply:

- ♦ `isLink(java.lang.Object)` returns `true` if the given parameter is a link. The model must be prepared to return valid values on `getFrom` and `getTo` calls on a link.
- ♦ `getFrom(java.lang.Object)` returns the model object that is the source of the given link.
- ♦ `getTo(java.lang.Object)` returns the model object that is the destination of the given link.

Methods for retrieving ID, tag, and properties

The following methods give information on a model object:

- ♦ `getID(java.lang.Object)` returns the unique ID (a String) associated with the object. IDs are used to define the references from a link to its source and destination nodes.
- ♦ `getObject(java.lang.String)` returns the model object associated with the given ID.
- ♦ `getTag(java.lang.Object)` returns the tag of a model object. The tag is a String that describes the type of the model object. Several objects may share the same tag. For example, the tag is the name of the XML element in an XML data file.
- ♦ `getObjectPropertyNames(java.lang.Object)` returns a String array that represents all the defined properties for this object.
- ♦ `getObjectProperty(java.lang.Object, java.lang.String)` returns an object representing the value of the given property on the given object.

Methods for implementing listeners

The model is dynamic: it can change itself. To keep the graphical display consistent with the model state, there is a listener mechanism that sends events when a modification occurs in the model.

- [`addSDMModelListener(ilog.views.sdm.event.SDMModelListener)` | `removeSDMModelListener(ilog.views.sdm.event.SDMModelListener)`] manages a listener list for model events. A model event should be fired when the model hierarchy or a model object property is changed.

- [`addSDMPropertyChangeListener(ilog.views.sdm.event.SDMPropertyChangeListener)` | `removeSDMPropertyChangeListener(ilog.views.sdm.event.SDMPropertyChangeListener)`] manages a listener list for property change events. A property change event should be fired when a model object property is changed.

Methods for editable models

All the editable methods are meant to be implemented if the model predicate `isEditable()` returns `true`.

The following methods modify the model hierarchy:

- `createNode(java.lang.String)` and `createLink(java.lang.String)` create, respectively, a node and a link. The argument represents the object tag. These methods return a model object.

- `addObject(java.lang.Object, java.lang.Object, java.lang.Object)` adds the first argument to the model below the second one (or at the top level if the second argument is `null`), and before the third argument (or at the end if the third argument is `null`).

- `removeObject(java.lang.Object)` removes the given object from the model.

- `clear()` removes all objects at once.

The following model methods change object properties:

- `setFrom(java.lang.Object, java.lang.Object)` and `setTo(java.lang.Object, java.lang.Object)` modify the source and destination objects of a link. The first argument is the link; the second one is the new end point.

- `setObjectProperty(java.lang.Object, java.lang.String, java.lang.Object)` changes a model object property. The first argument is the model object, the second one is the property name, and the third one is the property value.

- `setID(java.lang.Object, java.lang.String)` allows you to change the ID of a model object. It is wise to call this method before adding the object to the model to make sure that there is always a unique, stable ID for each model object.

- `isAdjusting()` and `setAdjusting(boolean)` can be used to warn the model that several properties are about to be changed.

Connecting data sources to the diagram component

A diagram component (`IlvDiagrammer` instance) has an associated SDM engine instance. By default, the SDM engine is connected to the default SDM model.

There are several ways for you to load data into a diagram component:

♦ *XML data source*

♦ *JDBC data source*

♦ *Flat file data source*

♦ *In-memory data source*

♦ *Custom data model*

XML data source

The simplest way of loading data into an `IlvDiagrammer` instance is to load an XML file containing the descriptions of the objects to add to the model.

If you have available an XML file in diagram format, you can load it directly in the Designer without needing to write any Java™ code.

The data model implementation is class `IlvDefaultSDMModel` with user-defined types (tags) read from the XML file.

To load an XML file and display its contents in Java code, use:

```
diagrammer.getEngine().setXMLFile("file:example.xml");
```

For details of the advanced uses of XML, see *Handling XML data files in Java*.

JDBC data source

You can load a JDBC™ -compliant database into the Designer without needing to write any Java code. For an example, see Importing data from a database in *Using the Designer*.

The data model implementation is class `IlvJDBCSDMModel`with no user-defined types.

Flat file data source

You can load data from a flat file, for example, a comma-separated values (CSV) file, into the Designer without needing to write any Java code. For an example, see Importing data from a flat file in *Using the Designer*.

The data model implementation is class `IlvDefaultSDMModel` with no user-defined types.

In-memory data source

If you use the in-memory data source, you effectively load a basic data model and you can extend it in the Designer without needing to write any Java code.

The initial data model implementation is class `IlvDefaultSDMModel` and you can add user-defined types in the Designer.

Custom data model

If you have written custom data model classes, you can assign the data model to the SDM engine of an `IlvDiagrammer` instance as follows:

```
IlvSDMModel sdmModel = new MyModel();
diagrammer.getEngine().setModel(sdmModel);
```

The data model implementation is user-defined.

See also *Using a custom data model in the Designer*.

JavaBeans example

Shows how to display a set of JavaBeans™ using IBM® ILOG® JViews Diagrammer.

In this section

The Molecule example
Presents an example that displays a chemical molecule.

The Atom, Bond, and Molecule classes
Describes the classes used to represent atoms, bonds, and molecules.

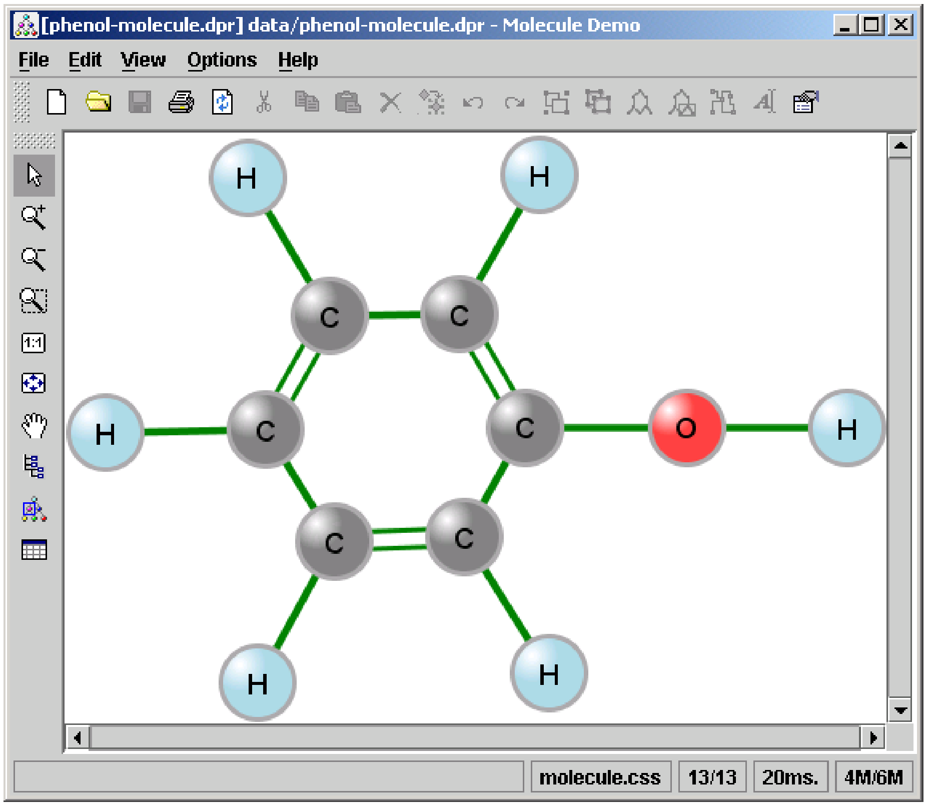
The Molecule model
Describes the class used to connect the `Atom`, `Bond`, and `Molecule` classes to the diagram component.

The Phenol Molecule data source
Describes the class used to represent a data source that contains a Phenol molecule.

Loading the Molecule into the diagram component
Shows how to define a project file that loads the Molecule data source.

The Molecule example

The molecule example displays a chemical molecule. The atoms are the nodes of the graph and the bonds between the atoms are the links of the graph, see the following figure.



The Molecule example: Phenol

The example is supplied with IBM® ILOG® JViews Diagrammer in the directory **<installdir>/jviews-diagrammer86/codefragments/datamodel/molecule**.

The Atom, Bond, and Molecule classes

The atoms and the bonds are represented in the application by the Java ™ classes `Atom` and `Bond`. These classes obey the JavaBeans™ conventions, for example, the `Atom` class has a property called `symbol` which represents the abbreviated symbol of the element; this property can be accessed through the methods `setSymbol` and `getSymbol`.

The Atom class

The following code example shows part of the `Atom` class.

Bean property in the Atom class

```
/**
 * A class that represents an atom.
 */
public class Atom
{

...

  private String symbol;

   public String getSymbol()
  {
    return symbol;
  }

  public void setSymbol(String symbol)
  {
    this.symbol = symbol;
  }


}
```

The `Atom` class has also a `name` property (the name of the element) , and an `id` property, which identifies the atom in the molecule.

The Bond class

The `Bond` class has two properties `firstAtom` and `secondAtom` which contain the identifiers of the two atoms linked by the bond, and also a `type` property which can have the values `single` or `double`.

The Molecule class

A molecule is represented by an instance of the class `Molecule`. A molecule contains a list of atoms and a list of bonds.

The following code example shows the `Molecule` class.

Arrays of objects in the Molecule class

```
public class Molecule
{
  private ArrayList atoms = new ArrayList();
  private ArrayList bonds = new ArrayList();

  public Atom[] getAtoms()
  {
    return (Atom[]) atoms.toArray(new Atom[0]);
  }

  public Bond[] getBonds()
  {
    return (Bond[]) bonds.toArray(new Bond[0]);
  }

}
```

The Molecule model

To connect the existing classes `Atom`, `Bond` and `Molecule` to the diagram component, you need to write a custom SDM model. Since there are already JavaBeans™ that represent the nodes and links of the graph (the `Atom` and `Bond` classes), the easiest solution is to write a subclass of `IlvJavaBeanSDMModel`. This subclass is called `MoleculeModel`.

The following code example shows the `MoleculeModel` class.

```
public class MoleculeModel extends IlvJavaBeanSDMModel
{
```

This class represents a molecule, so you can store an instance of the Molecule class as follows:

```
 private Molecule molecule;
```

The constructor takes the molecule as a parameter. The constructor must also initialize the model to know which Bean property holds the node identifier and which Bean properties hold the link ends.

The following code example shows the constructor.

```
  public MoleculeModel(Molecule molecule)
  {
    this.molecule = molecule;
    setIDProperty("id");
    setFromProperty("firstAtom");
    setToProperty("secondAtom");
  }
```

The call to `setIDProperty` tells the superclass `IlvJavaBeanSDMModel` that it can use the `id` property (through the `setId` and `getId` methods of the `Atom` class) to set and retrieve the identifier of a node.

The calls to `setFromProperty` and `setToProperty` tell the superclass which properties represent the two end nodes of a link. For example, to retrieve the source node of a link (a `Bond` instance), the model will call `getFirstAtom` on the `Bond` instance.

The model needs to retrieve all the nodes and links of the graph, that is the `Atom` and `Bond` objects of the molecule.

The following code example shows the `getObjects` method, which retrieves the objects.

```
public Enumeration getObjects()
  {
    Vector v = new Vector();
    Atom[] atoms = molecule.getAtoms();
    for (int i = 0; i < atoms.length; i++) {
      v.add(atoms[i]);
    }
    Bond[] bonds = molecule.getBonds();
    for (int i = 0; i < bonds.length; i++) {
      v.add(bonds[i]);
    }
    return v.elements();
  }
```

The model must be able to differentiate between nodes and links.

The following code example shows the `isLinks` method, which returns `true` for `Bond` objects and `false` for `Atom` objects.

```
public boolean isLink(Object obj)
  {
    return obj instanceof Bond;
  }
}
```

The Phenol Molecule data source

The classes and methods of the Molecule example represent a molecule so that the diagram component can display it. To display a specific molecule, you need to load the `Molecule` instance from a data source.

The following code example shows a data source that contains a Phenol molecule. This class is a subclass of `IlvDiagrammerDataSource`.

```
public class PhenolMoleculeDataSource extends IlvDiagrammerDataSource
{ ...
```

The Read method

To load a data source into a diagram component, call its `read` method, which is shown in the following code example.

```
public void read(IlvDiagrammer diagrammer) throws IlvDiagrammerException
  {
    Molecule phenol = Molecule.createPhenolMolecule();
    MoleculeModel model = new MoleculeModel(phenol);
    diagrammer.getEngine().setModel(model);
  }
```

The code lines in the `read` method are as follows:

1. Create the phenol `Molecule` instance with a static method for this purpose.
2. Wrap the `Molecule` instance into the molecule model.
3. Load the new model into the diagram component.

Empty methods

The data source class has other methods which must be implemented because they are abstract, but they do nothing. The following code example shows these methods.

Implementation of unused abstract methods

```
  public void write(IlvDiagrammer diagrammer)
          throws IlvDiagrammerException
  {
  }

  protected void serializeImpl(Element element)
          throws IlvDiagrammerException
  {
  }

  protected void deserializeImpl(Element element)
```

```
          throws IlvDiagrammerException
  {
  }
```

The `write` method is not needed because the molecule data cannot be written.

The `serialize` method is not needed because there is no need to write any additional information to define the data source in a project file.

The `deserialize` method is not needed because there is no need to read any additional information to load the data source from a project file.

Loading the Molecule into the diagram component

You must define a project file to tell the diagram component to use the data source. The project file is an XML file called `phenol-molecule.idpr`. This file contains the class name of the data source, and also the path of the style sheet used to represent the molecule.

The following code example shows the project file contents.

The project file for the Molecule example

```
<diagrammer style="molecule.css">
  <datasource class="PhenolMoleculeDataSource"/>
</diagrammer>
```

To display the molecule, you can simply load the project file into the diagram component as follows:

```
Diagrammer.setDataFile(new URL("file:phenol-molecule.idpr"));
```

Note: The sample uses the prebuilt class `IlvDiagrammerApplication`, so you do not see the call to `setDataFile`. Instead the project file is passed as a command-line argument to the application.

NonJavaBeans example: abstract model variant

Describes how to display a set of Java™ objects that are not JavaBeans™ by subclassing the abstract class `IlvAbstractSDMModel`.

In this section

The Tree Model example
Presents an example that displays the contents of a Swing JTree object as a graph.

The Swing JTree
Explains how to create a Swing JTree object.

The TreeSDMModel class
Describes how to write a data model that transforms the tree data into an SDM model.

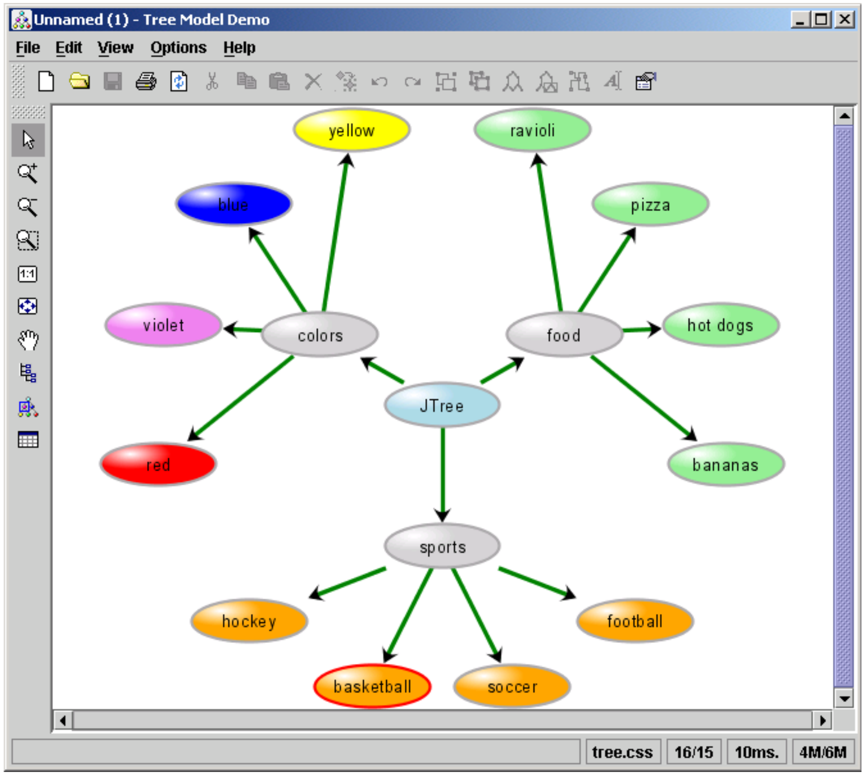
The TreeSDMNode class
Describes the `TreeSDMNode` class, which represents the nodes of the graph.

The TreeSDMLink class
Describes the `TreeSDMLink` class, which represents the links of the graph.

Loading the data model and style sheet into the diagram component
Shows how to load the data model and style sheet into the diagram component.

The Tree Model example

The Tree Model example displays the contents of a Swing JTree object as a graph. The nodes of the graph are the items in the tree (also called tree nodes), and the links of the graph represent the parent-child relationships between the items., see the following figure.



The Tree Model example: Swing JTree

This example uses the base class `IlvAbstractSDMModel` and implements nodes and links using the `IlvSDMNode` and `IlvSDMLink` interfaces, for the following reasons:

- ♦ The nodes to be represented (the tree nodes) are existing Java™ objects, but they are not JavaBeans™ .
- ♦ The links do not exist as Java objects; you need to create them.

The example is supplied with IBM® ILOG® JViews Diagrammer in the directory **<installdir>/jviews-diagrammer86/codefragments/datamodel/treemodel** .

The Swing JTree

The data to be displayed is the contents of a Swing JTree. The example simply uses the contents of a default JTree, which contains names of various colors, sports and food. You create the JTree object as shown in the following code example.

```
TreeModel treeModel = new JTree().getModel();
```

The TreeSDMModel class

To display the data model of a tree in a diagram component, you must write a data model that transforms the tree data into an SDM model. In this example, the model is implemented by the class `TreeSDMModel`, which is a subclass of `IlvAbstractSDMModel`. Its definition is as shown in the following code example.

```
public class TreeSDMModel extends IlvAbstractSDMModel
{
...
```

Root of the tree

The nodes of the graph will be represented by instances of a class `TreeSDMNode`. The `TreeSDMModel` class creates a node that represents the root of the tree, and keeps a reference to it as shown in the following code example.

```
 private TreeSDMNode root;

  public TreeSDMModel(TreeModel treeModel)
  {
    root = new TreeSDMNode(treeModel, null, treeModel.getRoot());
  }
```

When asked to return the top-level objects of the graph, the `getObjects` method just returns a single element: the root of the tree, as shown in the following code example.

```
public Enumeration getObjects()
  {
    Vector v = new Vector();
    v.addElement(root);
    return v.elements();
  }
```

Model is not editable

The `clear` method must be implemented, but it will never be called since the `TreeSDMModel` class in this example represents a model that is not editable. The following code example shows the method implementation.

```
  public void clear()
  {
    // Nothing, this model is immutable.
  }
}
```

The TreeSDMNode class

The class `TreeSDMNode` represents the nodes of the graph. Its definition is as shown in the following code example.

```
public class TreeSDMNode implements IlvSDMNode
{
```

Data stored

Each node in the graph (graphic object) has a reference to the corresponding node in the tree (model object). References to the parent node (graphic object) and the tree model are also needed. The children of the node are stored in a vector.

Keeping track of the data: current node, parent, children, and model

```
private TreeSDMNode parent;
private TreeModel treeModel;
private Object treeNode;
private Vector children;

public TreeSDMNode(TreeModel treeModel, TreeSDMNode parent, Object treeNode)
{
  this.parent = parent;
  this.treeModel = treeModel;
  this.treeNode = treeNode;

  createChildren();
 }
```

Children vector

Note that the constructor calls the method `createChildren`. This method traverses the tree model and creates a `TreeSDMNode` instance for each item in the tree. In addition, it creates an instance of `TreeSDMLink` to draw a link between the parent node and each child node.

Creating the child nodes and links (graphic objects)

```
private void createChildren()
{
  children = new Vector();

  // scan all the children of the root tree node, and create
  // a TreeSDMNode for each:
  //
  int count = treeModel.getChildCount(treeNode);
  for(int i = 0; i < count; i++){
    Object childTreeNode = treeModel.getChild(treeNode, i);

    // Create the TreeSDMNode. Note that this will recursively create
    // the SDM nodes for all sub-nodes.
    //
   TreeSDMNode childSDMNode = new TreeSDMNode(treeModel, this, childTreeNode)
;
    children.addElement(childSDMNode);

    // Create a parent/child link:
    //
    TreeSDMLink childSDMLink = new TreeSDMLink(this, childSDMNode);
    children.addElement(childSDMLink);
  }
}
```

Implementation of the IlvSDMNode interface

The following methods are the implementations of the methods belonging to the `IlvSDMNode` interface, which are inherited by `TreeSDMNode`.

The `getTag` method returns the type (or “tag”) of the node. In this example, the type is `treenode`.

Retrieving the node type: the getTag method

```
  public String getTag()

  {
    return "treenode";
  }
```

The `getID` method returns the identifier of the node. The identifier of a node is its hash code.

Retrieving the node ID: the getID method

```
  public String getID()
{
    return String.valueOf(hashCode());
  }
```

The `getChildren` method returns the children of the node, which are stored in the `children` data member.

Retrieving the child objects: the getChildren method

```
public Enumeration getChildren()

  {
    return children.elements();
  }
```

The `getParent` method returns the parent node of the current node.

Retrieving the parent object: the getParent method

```
  public IlvSDMNode getParent()
  {
    return parent;
  }
```

The `getProperty` and `getPropertyNames` methods must be implemented to give the diagram component access to the properties of the node. In this example, there is support for two properties, `userObject` and `CSSclass`:

- The `userObject` property returns the sample food, color, and sports names.
- The `CSSclass` property returns the type of item: `food`, `color`, or `sport`.

Retrieving a property: the getProperty method

```
  public Object getProperty(String property)
  {
    if(treeNode instanceof DefaultMutableTreeNode){
      if(property.equals("userObject")){
        return ((DefaultMutableTreeNode)treeNode).getUserObject();
      }
      if(property.equals("CSSclass") && ((DefaultMutableTreeNode)treeNode).
getParent() != null){
        return ((DefaultMutableTreeNode)((DefaultMutableTreeNode)treeNode).
getParent()).getUserObject();
      }
    }
    return null;
  }
```

The `getPropertyNames` method retrieves the two property names.

Retrieving property names: the getPropertyNames method

```
  public String[] getPropertyNames()
  {
    if(treeNode instanceof DefaultMutableTreeNode)
      return new String[] { "userObject", "CSSclass" };
    else
      return new String[0];
  }
```

The TreeSDMLink class

The class `TreeSDMLink` represents the links of the graph. Its definition is as shown in the following code example.

```
public class TreeSDMLink implements IlvSDMLink
{
...
```

The link implementation can therefore make use of all the predefined methods of the `IlvSDMLink` interface.

Data Stored

Each link has a reference to a parent node and a child node, which can be retrieved using the `getFrom()` and `getTo()` methods, see the following code example.

Keeping track of link data: parent and child

```
  private TreeSDMNode parent, child;

  public TreeSDMLink(TreeSDMNode parent, TreeSDMNode child)
  {
    this.parent = parent;
    this.child = child;
  }

  public IlvSDMNode getFrom()
  {
    return parent;
  }

  public IlvSDMNode getTo()
  {
    return child;
  }
```

Implementation of the IlvSDMNode interface

The `IlvSDMLink` interface inherits from the `IlvSDMNode` interface and so the same methods must be implemented as for a node.

The tag (type) of links is `treelink`. and this value can be retrieved using the `getTag` method, see the following code example.

```
public String getTag()
  {
    return "treelink";
  }
```

The remaining methods are mostly empty, since in this example links have no properties.

Loading the data model and style sheet into the diagram component

The Tree Model sample is a subclass of `IlvDiagrammerApplication`. It therefore has the full application facilities as described in Customizing the predefined application.

The Data model

To load the `TreeSDMModel` object, the sample overrides the `createDiagrammer()` method to perform extra operations, see the following code example.

```
protected IlvDiagrammer createDiagrammer()
  {
    // Create a default tree model. The JTree creates one
    // if you don't specify a model: let's use it.
    //
    TreeModel treeModel = new JTree().getModel();

    // Create the IlvDiagrammer instance.
    //
    IlvDiagrammer diagrammer = super.createDiagrammer();

    // Create a default tree model. The JTree creates one
```

The sample creates an instance of `TreeSDMModel`, giving it a default tree model as a parameter, see the following code example.

```
 // Create the Tree -> SDM model adapter.
    //
    TreeSDMModel sdmModel = new TreeSDMModel(treeModel);
```

The sample then gets the SDM engine for the `IlvDiagrammer` instance and sets the data model to the `TreeSDMModel` instance, as shown in the following code example.

```
  // Tell the IlvDiagrammer to use this model:
    //
    diagrammer.getEngine().setModel(sdmModel);

    return diagrammer;
  }
```

The style sheet

The style sheet used to display the tree model is located in **<installdir>/jviews-diagrammer86/codefragments/datamodel/treemodel/data**.

To load the style sheet, you pass the relative path to the sample as a startup argument.

```
public TreeModelDemo()
```

```
  {
    super(new String[]{
      "-title", "Tree Model Demo",
      "-style", "data/tree.css",
    });
  }
```

The style sheet specifies a Tree layout in Radial mode with various parameters, in the two rules shown in the following code example.

```
SDM {
   GraphLayout : "Tree" ;
}

GraphLayout {
   flowDirection : "Bottom";
   layoutMode : "RADIAL";
   globalLinkStyle : "ORTHOGONAL_STYLE";
   position : "300,200";
   parentChildOffset : "10";
   siblingOffset : "10";
}
```

The style sheet also specifies the colors of the various objects.

Note that the `CSSclass` property of the `TreeSDMNode` class is recognized automatically by the CSS engine as specifying a CSS class because of its name, and therefore the values of this attribute can be used directly in rules. This allows you to set colors at the level of types of object with a simple syntax, see the following code example.

```
node.food {
  fillColor2 : "lightgreen" ;
  }

// node.treenode[userObject='food'] node.treenode {
//   fillColor2 : "lightgreen" ;
// }
```

The comment lines show the alternative, longer way to set the color for food objects. This longer way relies on a parent-child construct in CSS for Java. Note that the parent-child construct is not supported in the Designer.

NonJavaBeans example: basic model variant

Describes how to display a set of Java™ objects that are not JavaBeans™ by subclassing the basic class `IlvBasicSDMModel`.

In this section

The second Tree Model example
Presents another example that adapts the Swing JTree model to an SDM model.

The TreeSDMModel2 class
Describes the `TreeSDMModel2` class, which transforms the tree data into an SDM model.

The TreeLink class
Describes the `TreeLink` class, which holds references to the parent and child tree nodes.

Loading the data model
Shows how to load the data model for this example.

The second Tree Model example

This section makes use of the same example as for the `IlvAbstractSDMModel` variant, which involves adapting a Swing JTree model to an SDM model. Therefore, for a description of the common parts, see the subsections of that example as follows:

- Example description, see *The Tree Model example*
- Style sheet description, see *The style sheet*
- How to load the data model and style sheet, see *Loading the data model and style sheet into the diagram component*

The remaining subsections that follow describe the data model implementation for this variant.

The example is supplied with IBM® ILOG® JViews Diagrammer in the directory **<installdir>/jviews-diagrammer86/codefragments/datamodel/treemodel2**.

The TreeSDMModel2 class

To display the data model of a tree in a diagram component, you must write a data model that transforms the tree data into an SDM model. In this example, the model is implemented by a subclass of `IlvBasicSDMModel`, as shown in the following code example.

```
public class TreeSDMModel2 extends IlvBasicSDMModel
{
...
```

This SDM model will use the tree nodes taken from the JTree model directly, instead of implementing a new class to represent the nodes of the graph. This approach has the advantage of saving one object allocation for each node.

However, a new class is needed for the links, because there is no object that represents a parent-child relationship in a JTree model.

Reference to the tree model

The `TreeSDMModel2` class keeps a reference to the tree model, as shown in the following code example.

```
private TreeModel treeModel;
  private ArrayList links = new ArrayList();

  public TreeSDMModel2(TreeModel treeModel)
  {
    this.treeModel = treeModel;

    createLinks(treeModel.getRoot());
  }
```

Method for creating links

The `createLinks` method creates the parent-child links and stores them in a list, as shown in the following code example.

```
private void createLinks(Object treeNode)
  {
    for(int i = 0; i < treeModel.getChildCount(treeNode); i++){
      Object childNode = treeModel.getChild(treeNode, i);
      links.add(new TreeLink(treeNode, childNode));

      createLinks(childNode);
    }
  }
```

Method for retrieving all nodes and links

The `getObjects` method (shown in the following code example) returns all the nodes of the tree recursively, and also returns the links that have been created in the constructor. Note that, in this variant, the data model is flat, that is, the tree hierarchy does not translate into subgraphs, so all nodes and links are at the same level in the SDM model.

Retrieving nodes and links

```
public Enumeration getObjects()
  {
    Vector v = new Vector();

    getTreeNodes(treeModel.getRoot(), v);

    for (int i = 0; i < links.size(); i++) {
      v.addElement(links.get(i));
    }

    return v.elements();
  }

  private void getTreeNodes(Object parentNode, Vector v)
  {
    v.addElement(parentNode);
    for(int i = 0; i < treeModel.getChildCount(parentNode); i++){
      getTreeNodes(treeModel.getChild(parentNode, i), v);
    }
  }
```

Methods of the SDM model interface

The methods of the SDM model interface are implemented directly in the subclass of `IlvBasicSDMModel`, instead of being implemented in the node and link classes, see the following code example.

Implementing the SDM model interface

```
  public String getTag(Object obj)
  {
    if(isLink(obj))
      return "treelink";
    else
      return "treenode";
  }

  public boolean isLink(Object obj)
  {
    return obj instanceof TreeLink;
  }

  public Object getFrom(Object link)
  {
```

```
  return ((TreeLink)link).getParentNode();
}

public Object getTo(Object link)
{
  return ((TreeLink)link).getChildNode();
}

public Object getObjectProperty(Object object, String property)
{
  if(object instanceof DefaultMutableTreeNode){
    if(property.equals("userObject")){
      return ((DefaultMutableTreeNode)object).getUserObject();
    }
    if(property.equals("CSSclass") &&
       ((DefaultMutableTreeNode)object).getParent() != null){
      return
      ((DefaultMutableTreeNode)((DefaultMutableTreeNode)object).
           getParent()).getUserObject();
    }
  }
  return null;
}

public String[] getObjectPropertyNames(Object object)
{
  if(object instanceof DefaultMutableTreeNode)
    return new String[] { "userObject", "CSSclass" };
  else
    return new String[0];
}
```

The TreeLink class

The links are represented by instances of a very simple class that just holds references to the parent and child tree nodes, see the following code example.

```
public class TreeLink
{
  private Object parentNode;
  private Object childNode;

  public TreeLink(Object parentNode, Object childNode)
  {
    this.parentNode = parentNode;
    this.childNode = childNode;
  }

  public Object getParentNode()
  {
    return parentNode;
  }

  public Object getChildNode()
  {
    return childNode;
  }
}
```

Loading the data model

Most of the code in the sample is the same as in the first variant; the only difference is the way the model is created, see the following code example.

```
// Create the Tree -> SDM model adapter.
//
   TreeSDMModel2 sdmModel = new TreeSDMModel2(treeModel);
```

Using a custom data model in the Designer

You can make use of a custom data model in the Designer but certain limitations apply.

The way to prepare a custom data model for use in the Designer is to export it to XML

Exporting your data model

The data model needs to be available in XML format, preferably diagram format, for the Designer to load it. You can either write the XML file manually or generate it from the Java™ classes in Java code.

If you have XML data that is not in diagram format, see *Handling XML data files in Java*.

Loading the XML file

You can create a project file for an exported XML file and its associated style sheet in Java code (see *The Phenol Molecule data source*). You can load the project file in Java code as well, see *Loading the Molecule into the diagram component*. Once you have a project file, you can also open it in the Designer to style the nodes and links interactively in Style Editing Mode. If the custom data model is read-only, you will not be able to edit the model in Diagram Editing Mode.

Writing your application

Your application will be based on the style sheet and on your Java classes. If you saved the style sheet in the Designer, then it applies styles to the node and link classes used for graphic representation by the Designer instead of to your Java classes. Therefore, your application will need to adapt the style sheet to your Java classes.

Handling XML data files in Java

Explains how the class `IlvSDMEngine` allows you to read, write, and transform XML data.

In this section

Reading data
Shows several methods of reading the data from an XML file.

Writing data
Shows several methods of writing the contents of the data model to an XML file.

Reading and writing custom XML formats
Describes how to read and write data using a custom XML format.

Reading data

There are several ways available to read an XML file:

- Call the method `setXMLFile(java.lang.String)`.
- Call the method `readXML(java.io.InputStream)` to read XML data from an input stream.
- Call the method `readDOM(org.w3c.dom.Document)` to read data directly from a DOM tree.

Writing data

There are several ways available to write the contents of the data model to an XML file:

- ♦ Call the method `writeXML(java.lang.String)` , specifying the filename:

```
myEngine.writeXML("output.xml");
```

- ♦ Call the method `writeXML(java.io.OutputStream)` to write XML data to an output stream.
- ♦ Call the method `writeDOM(org.w3c.dom.Document)` to write data directly to a DOM tree.

Reading and writing custom XML formats

It is possible to read and write data in any XML format. To do this, you have two solutions:

♦ Transform the XML format to the IBM® ILOG® JViews Diagrammer SDM format using XSLT transformations

♦ Write a subclass of the utility class `IlvXMLConnector`

Using XSLT

XSLT is the simplest solution for reading and writing custom XML formats, because it requires no Java™ coding. This solution is easier to apply when the structure of the custom XML format is relatively close to the required structure.

For documentation and resources on XSLT, see the W3C web site (*http://www.w3c.org*). Proceed as follows:

1. Write two XSLT files: one that will convert an XML document from the custom format to the SDM format (for reading), and another one that will convert a document from the SDM format to the custom format (for writing).

2. Create an instance of the class `IlvXSLConnector` (a subclass of `IlvXMLConnector`), specifying the two XSLT files. To do this through the style sheet add a rule with the following structure:

```
XMLConnector{
    class : "ilog.views.sdm.util.IlvXSLConnector";
    inputTemplates : "url(custom2sdm.xsl)";
    outputTemplates : "url(sdm2custom.xsl)";
}
```

   where the two XSLT files are `custom2sdm.xsl` (for reading), and `sdm2custom.xsl` for writing.

Note that the two XSLT files must be written so that no information is lost, that is, the two transformations should be the exact inverse of one another. If the custom XML format contains information that is not useful for IBM® ILOG® JViews Diagrammer purposes or that cannot be translated easily, this information can be stored as metadata (see Metadata in the XML data file in *Using the Designer*).

It is possible to specify only one transformation. For example, you may specify only the input templates file if you read only custom XML files, but never modify and write them back.

Subclassing the XML connector

There may be cases when you will not be able to write XSL transformations between the custom XML format and the SDM format, usually because the structures are too different and the XSLT language does not allow you to implement the transformation.

In such cases, proceed as follows:

1. Write a subclass of `IlvXMLConnector`.

2. Create an instance of your custom XML connector in the style sheet, by writing a rule with the following structure:

```
XMLConnector{
    class : "my.package.MyConnector";
}
```

Content on demand

Describes the content on demand feature.

In this section

About content on demand
Describes the purpose and use cases of the content on demand feature.

Concepts
Describes the concepts involved in content on demand.

Classes involved
Describes the classes used by the content on demand feature.

Using content on demand
Explains how to implement the content on demand feature.

Content on demand and tiling systems
Describes what makes the difference between tiling systems and content on demand feature.

About content on demand

The content on demand feature allows an SDM model to delay the loading of its objects content in order to save resources. Unlike a tiling system, content on demand requires that all the objects be present in the model. The objects can be hollow. Assuming that hollow model objects have low memory and time footprints, content on demand allows you to fill them on demand and empty them when they are no more needed. Content on demand sends a notification when the content of a set of objects needs to be loaded, and, optionally, when it can be unloaded.

Content on demand addresses two main use cases:

♦ Models that are time-consuming in terms of loading, and applications that are slow at start up. The lazy loading of content speeds the start up of the Diagrammer application because the model and the graphic representation of the model objects are light and simple. Objects are filled on demand, and the customization and rendering time is used only for the requested objects.

♦ Fat models that need to load content for a subset of objects only. Typically, only the objects that are visible in the main view should be fully loaded. When panning or zooming, the content of newly visible objects is loaded and the content of non visible objects can be unloaded, which keeps the model memory footprint to a minimum.

The classes of the content on demand feature are located in the package `ilog.views.sdm.modeltools`. The entry point is the class `IlvContentController`.

A content on demand sample is available in **<installdir>/jviews-diagrammer86/samples/diagrammer/content-on-demand**.

Concepts

The content on demand feature operates at the granularity of SDM model objects. Content on demand makes use of the following concepts.

Content controller

The content controller maintains a state for each model object. A model object can be in one of three states:

- **locked** The object content is loaded and in use.
- **loaded** The object content is loaded, but not locked. It is considered as cached and will be unloaded when the cache is full.
- **unloaded** The object content is not loaded.

The state changes after events, such as a user request, a zoom or a pan change (in fact visible area changes), have taken place. The controller then sends a notification to explicitly load or unload a set of objects.

Usually, the lifecycle of an object content is: **unloaded**, then **locked**, then **loaded**, then potentially back to **unloaded**.

The cache is virtual. The object content is held in the SDM model. When the state changes to **loaded**, this simply means that the content remains in the model. As a matter of fact, the SDM model implements the cache for the object content, and the controller manages this cache.

The controller contains the states of the objects as well as a handler that processes the load and unload commands.

Cache size tuning

Extreme values for the cache size have a radical impact on the content on demand behavior.

- Null (0) cache size: The objects are either in use (locked) or unloaded. Use this value to minimize the memory footprint.
- Infinite cache size: The objects are loaded when needed but never unloaded. This is a pure lazy load behavior.

For other cache size values, the objects are loaded when locked and remain in the cache until they are flushed (FIFO way) to keep the average memory size close to constant. Note that when objects are requested to be locked, they are loaded before the cache is purged to avoid unloading an object that is going to be locked again. This means that the cache can temporarily be larger that the expected size.

Locking requests

A locking request is an event sent to the controller to lock, load or unload objects. There are two types of event:

- **Area events** The controller is requested to lock all objects that intersect a given area. A mode indicates whether previously locked objects should remain locked or should be unlocked, that is, loaded (default). Typically, the area may be computed from the visible part of the main view in order to load all visible objects.

- **Set events** The user sends a request to the controller to lock, load, or unload a given set of objects. For example, a drill-down mechanism loads the content of the selected object.

Both types of event are non exclusive: An area event may be followed by a set event, and conversely.

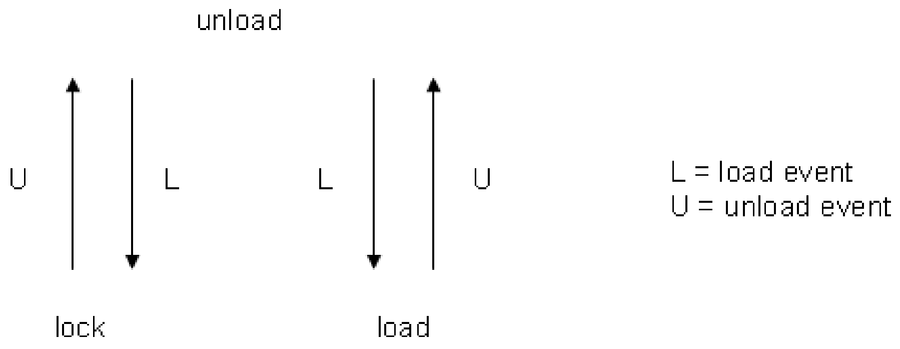
Notifications

The controller does not know by itself how to load the content of SDM model objects. On request, it calls a handler, `IlvContentHandler`, that performs the load and unload actions by upgrading the SDM model. Then, the usual SDM model notifications take over to customize and refresh the graphic representations.

A load event is sent when the object state changes from unloaded to locked or loaded.

An unload event is sent when the object state changes from locked or loaded to unloaded.

No notification is sent when the state changes between loaded and locked.



Multithreading

The handler is able to work asynchronously, that is, if the controller asks to load an object, the handler marks the object as locked while the object may be loaded in a separate thread. In any case, the handler must notify the SDM model when the action is complete. The locking requests are synchronized, so that the handler holds the controller synchronization lock until completion of the event.

SDM model changes

The controller is aware of SDM model structural changes when:

- `IlvSDMEngine.setModel()` sets a new model: All the objects are unloaded without notification to the handler.

- The SDM model deletes an object: The object is unloaded in the controller without notification to the handler.

To summarize, the controller forgets about the obselete objects of the model.

Classes involved

The content on demand feature involves the following classes:

The IlvContentController class

The `IlvContentController` class defines the following methods:

♦ `setSDMEngine(IlvSDMEngine engine)` Sets the SDM engine to work on.

♦ `setContentHandler(IlvContentHandler handler)` Sets the handler.

♦ `setCacheSize(int size)` Sets the cache size.

Methods to send requests

♦ `lockArea(IlvRect area, int mode)` Locks the objects that intersect with the given rectangle. The mode indicates the status of previously locked objects. The allowed values are:

- `IlvContentHandler.OVERRIDE` Previously locked objects are unlocked.
- `IlvContentHandler.AUGMENT` Previously locked objects remain locked.

The class `ilog.views.sdm.modeltools.IlvVisibleAreaListener` provides a view listener that locks automatically the visible area of an SDM view.

♦ `processObjects(Object[] objects, int action)` Locks, loads, or unloads the given objects, according to the action value, which can be one of:

- `IlvContentHandler.LOCK` Lock action.
- `IlvContentHandler.UNLOCK` Unlock action (once unlocked, objects can be unloaded at any time).
- `IlvContentHandler.UNLOAD` Unload action.

♦ `processObjects(Enumeration objects, int action)` Similar to the previous method but objects are defined in an Enumeration.

♦ `processObjects(Iterator objects, int action)` Similar to the previous method but objects are defined in an Iterator.

♦ `processAllObjects(int action)` Similar to the previous method but operates on all objects.

All these methods are synchronized and return an integer which represents the number of objects that have switched to the requested state.

Methods to retrieve the current state of an object

The following method retrieves the current state (LOCK/UNLOCK/UNLOAD) of an object in the controller. Depending on the handler implementation (typically, if actions are performed in a different thread), the state may differ from reality, for example, a content is not yet available although the state is LOCK.

♦ `int getObjectStatus(Object obj)` Returns the current state of the given object.

Other methods of the `IlvContentController` class:

- `IlvSDMEngine getSDMEngine()` Returns the SDM engine.
- `IlvContentHandler getContentHandler()` Returns the content handler.
- `void setNotificationEnabled(boolean on)` If `on` is `false`, the handler is not notified.
- `boolean isNotificationEnabled()` Returns whether notifications are enabled.
- `int getLockedCount()` Returns the number of locked objects.
- `int getLoadedCount()` Returns the number of objects in the cache.

The IlvContentHandler interface

The `IlvContentHandler` interface defines the following methods:

- `loadContent(IlvContentControler source, Object[] objs)` Loads the content of the given objects into the SDM model.
- `unloadContent(IlvContentControler source, Object[] objs)` Unloads the content of the given objects into the SDM model.

The `source` argument indicates the controller that sends the request.

The IlvVisibleAreaListener class

The `IlvVisibleAreaListener` class computes the visible area of a view as it changes and sends it as a request to the controller. The constructor takes the controller as parameter. To start listening to the visible area changes on a view, call the method:

- `installListener(IlvManagerView target)` Installs a transformer listener.

Example of use:

```
IlvVisibleAreaListener l = new IlvVisibleAreaListener(contentControler);
l.installListener(contentControler.getSDMEngine().getReferenceView());
```

The method `requestAreaToLock` can be subclassed to control the area to lock, as follows:

- `int requestAreaToLock(IlvRect area)` Called when the view transformer has changed. The default implementation simply calls the controller method `lockArea(area, OVERRIDE)` and returns the number of objects that have been loaded.

Using content on demand

The content controller needs to be associated with an SDM engine and a handler.

Using the content on demand feature involves the following steps:

To use the content on demand feature:

1. Associate the content controller with the SDM engine and the handler.

```
_controller = new IlvContentController();
_controller.setSDMEngine(diagrammer.getEngine());
_controller.setContentHandler(new ContentHandler());

private class ContentHandler extends IlvContentHandler() {
   // here we load/unload objects
   private void loadObject(Object node, boolean load) {
     if (load) {
       model.setObjectProperty(node, CONTENT, ...);
     } else {
       // unload values
       model.setObjectProperty(node, CONTENT, null);
     }
   }

   // callback for loading content
  public void loadContent(IlvContentController source, Object[] objects)
 {
     // prevent too much notifications
     diagrammer.setAdjusting(true);
     // loop over objects to load
     for (int i=0; i<objects.length; i++) {
       loadObject(objects[i], true);
     }
     diagrammer.setAdjusting(false);
   }

   // callback for unloading content
   public void unloadContent(IlvContentController source, Object[]
objects) {
     diagrammer.setAdjusting(true);
       for (int i=0; i<objects.length; i++) {
         loadObject(objects[i], false);
       }
       diagrammer.setAdjusting(false);
     }
   }
}
```

2. Optionally, change the cache value using the `setCacheSize(int size)` method.

   You can adjust the cache size according to the desired behavior. The default value is infinite, which means that all loaded content is never unloaded.

```
_controller.setCacheSize(1024);
```

3. Install a view listener that will send requests to the controller.

```
IlvVisibleAreaListener l = new IlvVisibleAreaListener(contentControler);
l.installListener(contentControler.getSDMEngine().getReferenceView());
```

4. Now navigate in the view to automatically load or unload objects.

Content on demand and tiling systems

The content on demand feature is similar to a tiling system, such as `ilog.views.tiling`, with the following major differences:

- ♦ The granularity of content on demand is an SDM model object, not a tile. However, it is possible to implement a tiling behavior outside the controller. For example, when the handler is asked to load an object, all the objects of the same group can be loaded as well.
- ♦ Content on demand does not create objects, it loads and unloads content. In other words, the controller knows only about the objects currently in the SDM model.
- ♦ The user has the possibility to load or unload objects that are not in the visible area.

Using CSS syntax in the style sheet

Gives an introduction to the CSS (cascading style sheets) standard in general and to the extension, called CSS for Java, which is used in IBM® ILOG® JViews Diagrammer, and then describes how you can customize nodes and links to give a variety of visual effects

In this section

Overview
Provides an overview to the CSS style sheets used in IBM® ILOG® JViews Diagrammer.

The origins of CSS
Presents a short history of cascading style sheets (CSS).

The CSS syntax
Describes the syntax used in cascading style sheets.

Applying CSS to Java objects
Shows how to apply the CSS to Java™ objects.

Customizing general nodes in the style sheet
Shows how to customize general nodes in the style sheet.

Customizing general links in the style sheet
Shows how to customize general links in the style sheet to affect its colors, borders, ends and joins, decorations, arrows, and hooks.

Overview

The style sheet syntax conforms to the CSS2 (Cascading Style Sheet level 2) specification with a few divergences.

The general format of a style rule in a style sheet is therefore:

```
selector {
  declaration1;
  declaration2;
...
}
```

For visualization purposes, the selector applies to objects in the data model, and is used for pattern-matching; the declarations apply to the corresponding graphic objects, and are used for rendering.

A declaration has the form:

```
propertyName : value ;
```

An example of a style rule is:

```
node.person[sex="female"]{
        fillColor2   : "red";
        personLegend : "Female";
}
```

This rule makes all nodes representing female persons red, and sets the legend for such nodes to the word Female.

This section introduces and describes CSS briefly and then explains in more detail the version of CSS used in IBM® ILOG® JViews Diagrammer and typical uses of CSS for customizing nodes and links.

The origins of CSS

Cascading style sheets (CSS) are a powerful mechanism to customize HTML rendering inside a Web browser. The CSS2 specification comes from the W3C, and has now reached the status of a W3C recommendation. See *http://www.w3.org/TR/REC-CSS2/*.

The CSS syntax is a great improvement over the *.Xdefault* resource mechanism of the X Window System. The basic idea remains the same: matching a pattern and setting resource values. CSS is devoted to HTML rendering, matching HTML tags and setting style values. XML is another CSS target, especially as used within the SVG (Scalable Vector Graphics) recommendation from the W3C.

The CSS syntax

Describes the syntax used in cascading style sheets.

In this section

Style rule
Describes style rules, as used in CSS documents.

Selector
Describes selectors, as used in CSS documents.

Declaration
Describes declarations, as used in CSS documents.

Priority
Describes priority, as used in CSS documents.

Cascading
Defines cascading in CSS documents.

Inheritance
Describes inheritance in CSS documents.

Style rule

A CSS document (a style sheet) consists of a set of *style rules*

A CSS document (a style sheet) consists of a set of *style rules*. Each style rule starts with a selector and is followed by a declaration block enclosed by curly braces. The selector defines a pattern, and the declarations are applied to the objects that match the pattern.

The basic example below shows how to apply the color red to all emphasis elements in the HTML document.

```
em { color : red ; }
```

where `em` is the selector, and “`color : red ;`” is a declaration.

It is possible to group several rules with the same declarations. Use a comma “,” to separate the selectors. For example:

```
em, b { color : red ; }
```

Selector

The W3C states that “A selector represents a structure. This structure can be understood for instance as a condition that determines which elements in the document tree are matched by this selector, or as a flat description of the HTML or XML fragment corresponding to that structure.”

Examples of selectors:

- H3
- P.footer
- TABLE#bigtable > TR
- TABLE#bigtable TD
- node
- node[x="2"]
- node:selected
- node#subgraph1 > #id2

A selector is composed of one or more simple selectors.

Examples of simple selectors:

- H3
- P.footer
- TABLE#bigtable
- TR
- node
- node[x="2"]
- node:selected
- #id2

A simple selector is made of minimal building blocks.

Examples of minimal building blocks of selectors:

- H3
- .footer
- node
- [x="2"]
- :selected

♦ #id2

When two or more simple selectors are aggregated into a selector, they are separated by combinators. A combinator is a single character which semantics is described in the following table. Extra spaces are ignored.

Combinator symbols

Transition	Meaning
E F	Matches an F element that is a descendant of an E element
E > F	Matches an F element that is a child of an E element
E + F	Matches an F element immediately preceded by an E element

The minimal building blocks of a selector are listed in the following table. For an explanation of the Specificity column, see *Priority*.

Minimal building blocks of a selector

Building Block	Matching Rule	Specificity
e	Matches any element of type *e*	0-0-1
#*myid*	Matches any element with an ID equal to *myid*	1-0-0
.*myclass*	Matches any element with class *myclass*	0-1-0
:*myclass*	Matches any element with pseudo-class *myclass*	0-1-0
[*myattr*]	Matches any element with the *myattr* attribute that exists and <> null	0-1-0
[*myattr*=”*warning*”]	Matches any element whose *myattr* attribute value is exactly equal to *warning*	0-1-0
[*myattr*~=”*warning*”]	Matches any element whose *myattr* attribute value is a list of space-separated values, one of which is exactly equal to *warning*	0-1-0
*	Matches any element	0-0-0

For example, the following line:

```
P.pastoral.marine { color : green ; size : 10pt ; }
```

matches `<P class="pastoral marine old">`, sets the color of the paragraph to `green`, and sets the font size to `10`.

All rules start and end with an implicit “ * ” pattern. This means that a selector can match anywhere inside the hierarchy.

Declaration

Declarations are key-value couples. The separator is a colon (:). Each declaration is terminated by a semicolon (;).

The key should represent a predefined graphic attribute (`foreground`, `size`, `font`, and so forth) and the value is a literal whose type depends on the key (such as `red`, `10pt`, or `serif`). All key-value pairs are String. It is recommended that you quote values with quotation marks `" "` or single quote marks `' '` when the values contain nonalphanumeric characters.

Priority

The priority of the rules depends on their relative *specificity*. Specificity is computed as three numbers, `a-b-c` (in a number system with a large base). The number of ID building blocks in the selector gives the first number `a`, the number of classes, pseudo-classes and attributes gives `b`, and the number of element types gives `c`.

The examples in the following table are in priority order, with the most specific first.

Priority Order Example

Selector	Specificity
#title > #author.full	“2-1-0”
#title	“1-0-0”
P.intro P.citation	“0-2-2”
UL OL LI.red	“0-1-3”

When two rules give the same specificity number, the order of appearance gives the priority (the last seen overrides previous rules).

Priority is used as follows: first the declarations of all rules that match the same objects are merged, and then the priority is applied only if there is a conflict (same key value) within the merged declaration block.

Cascading

Cascading consists of supplying several sources for the style.

In HTML environments there are three sources: the browser, the user, and the document. Cascading fixes another weight according to the source of the style. Document style takes precedence over user style, which takes precedence over browser style when the specificity number is the same.

There are two more tokens, `!important` and `inherit`. They are used to alter the cascading priority inside declarations.

A style sheet can also import other sheets (internal cascading). The syntax is:

```
@import "[url]" ;
```

Import statements must precede the first rule in a style sheet. Priorities of the imported rules are computed as if the rules replaced the import statements. Here is an example of import:

```
@import "common.css" ;
```

Inheritance

The main principle of CSS is the inheritance of declarations. Once the rules are checked against the source document, the matched declarations are sorted according to the priority order of the rules. The declarations are merged, with higher priority settings overriding lower ones in case of conflict.

The resulting set of key-value pairs represents all the declarations that the style sheet applies to a particular document.

Applying CSS to Java objects

Shows how to apply the CSS to Java™ objects.

In this section

Overview
Provides an overview of how to map the CSS mechanism to the hierarchy of Java™ objects

The CSS engine
Describes the CSS engine and what it does at load time and at run time.

The data model
Describes the input data model and the information it supplies to the CSS engine.

CSS recursion
Explains recursion in CSS documents.

Expressions
Explains expressions in CSS documents.

Divergences from CSS2
Describes the differences between the CSS2 syntax and the style sheets used in IBM® ILOG® JViews Diagrammer.

Overview

The CSS selector mechanism was designed to match elements in HTML or XML documents. It can also be used to match a hierarchy of Java™ objects accessible from a model interface. In this context, the CSS level 2 recommendation is transposed for the Java language and used to set Bean properties according to the Java object hierarchy and state.

In applying CSS to Java objects, the term model objects is used as the equivalent of the term elements in the W3C recommendation.

The CSS declarations for each model object are sorted and used according to the application that controls the CSS engine. The declarations represent property settings on a target object. The target object concerned depends on the way the CSS engine is used.

IBM® ILOG® JViews Diagrammer uses CSS declarations to create and customize one graphic object for each object in the data model, and to create and customize renderers according to user settings.

The CSS engine

The CSS engine has different responsibilities at load time and at run time:

♦ At load time: creating and customizing graphic objects and renderers

♦ At run time: customizing the graphic objects according to model changes

Usually the left side of a declaration represents a Bean property of the graphic object. The right side is a literal and, if it needs type conversion, the Java method `setAsText` is invoked on the Property Editor associated with the Bean property.

The data model

The input data model represents the seed of the “CSS for Java™ ” engine. It provides three important kinds of information to the CSS engine, required to resolve the selectors:

♦ The tree structure of objects, which will be exploited by selector transitions.

♦ Object type, ID, and tag (or user-defined type), which match element type, ID, and CSS classes. IDs and types are strings; CSS classes are words separated by a space character. ID is not required to be unique, although it is wise to assume so.

♦ Attribute, which matches an attribute of the same name in an attribute condition within the selector.

The target object is the graphic object associated with the model object. The declarations change property values of the graphic object that corresponds to the matching model object, thereby customizing the graphic appearance given by the rendering.

In IBM® ILOG® JViews Diagrammer the target object is an `IlvGraphic` instance (see Graphic objects in *The Essential JViews Framework,* for information about `IlvGraphic` objects) or a composite graphic.

Object types and attribute matching

The following code example shows a rule that matches the object of class (type) `test_Vehicle`, with the attribute `model` equal to `sport`, and sets the property `icon` of the graphic object associated with this object (defined elsewhere) to `sport-car.gif`.

```
test_Vehicle[model=sport] {icon : "sport-car.gif";}
```

Attribute matching can be used to add dynamic behavior: a `PropertyChange` event occurring on the model can activate the CSS engine to set new property values on the graphic objects.

The following code example shows a rule that changes the color of any object of CSS class `computer` whenever the model attribute `state` is set to `down`.

```
.computer[state = down] {color : "gray"}
```

Object identifiers and CSS classes

The Java model provides a `getID(java.lang.Object)` method which represents the ID of a model object. This ID can be checked against the # selector of a rule.

If there is a single CSS class in the selector, it is resolved with the `getTag(java.lang.Object)` method in the model.

Additional CSS classes can be set for an object in a property called `CSSclass`. The engine automatically merges the result of `getTag` with the value of the `CSSclass` property.

CSS classes are not necessarily related to data model semantics; they are devices to add to the pattern-matching capabilities in the style sheet. An object belongs to only one class (its type) but can belong to several (or no) CSS classes. A check on a CSS class is for its presence or absence. Therefore a CSS class can be seen as an attribute without a value.

By default, an XML element name is defined as a CSS class. If, for example, a simple XML file contains the element names `root` and `leaf`, then the following code example shows how to change the color of leaf nodes to an RGB color specification.

```
node.leaf {
    fillColor1 : 255,198,202 ;
    foreground : 153,40,100  ;
}
```

The RGB color specification shown for the foreground (border) color is magenta.

Class name

The `class` property is a reserved keyword indicating the class name of the generated graphic object. Obviously the class declaration is applied only when there is a creation request. If the model state is changed, the graphic objects are customized by applying only new declarations coming from new matching rules of the style sheet. The class declaration is then simply ignored.

To change the class, you must remove the model object and add it again.

The right side of a class declaration may be:

♦ The class name, loaded with the system class loader. For example:

  ```
  link {
    class      : ilog.views.sdm.graphic.IlvGeneralLink;
    foreground : red;
  }
  ```

  There are two supplied base classes: `IlvGeneralLink` for links (see *Customizing general links in the style sheet*) and `IlvGeneralNode` for nodes (see *Customizing general nodes in the style sheet*).

♦ A factory (interface `IlvRectangularObjectFactory`). The factory requires an `IlvRect` object which should be present in the declarations. This rectangle will be passed as a parameter of the factory. For example:

  ```
  node {
    class   : ilog.views.interactor.IlvMakeFilledRectangleInteractor;
    IlvRect : 0,0,100,200;
  }
  ```

  There is a factory for most of the available graphic components. See `IlvGraphicFactories` for more details.

♦ A pathname to a file. The class name is forwarded to the Beans library (method `java.beans.Beans.instantiate`) so a serialized Bean is suitable. For example:

```
link {
  class      : data.beans.gauge;
  foreground : red;
}
```

When `beanName` is used as a serialized object name, the given `beanName` is converted to a resource pathname and a trailing `.ser` suffix is added. An attempt is made to load a serialized object from that resource.

In this example, `Beans.instantiate` would try to read a serialized object from the resource `data/beans/gauge.ser`.

Pseudo-classes and pseudo-elements

Pseudo-classes are the minimal building blocks of a selector which match model objects according to an external context. The syntax is like a CSS class but with a colon instead of a dot. For example, `node:selected` matches a node only if the node is selected. The user agent can resolve this pseudo-class at run time according to the state of each node.

A pseudo-class has the same specificity as a CSS class.

Pseudo-elements are metaclasses, like pseudo-classes, but match document structure instead of the user agent state.

Model indirection

The right side of a declaration resolves to a literal that is determined at run time by a *Property Editor*. However, if the literal is prefixed by @, the remainder of the string is interpreted as a model attribute name. The declaration takes the value from the model object, as shown in the following code example.

```
node { label : "@caption" ; title : "CSS rocks" ;}
```

The `label` property will be set to the value of the attribute called `caption` in the model. If the specified atribute does not exist for the object, it is searched for recursively in the model ascendancy. The `title` property will be set to the literal `CSS rocks`.

Such indirection is also used in the opposite direction, that is, to retrieve the name of the model attribute that controls a graphic property. This allows user interactions to modify the data model correctly. Two special names, `@_ID` and `@_TAG`, represent values returned by the model method calls `getID(java.lang.Object)` and `getTag(java.lang.Object) getTag` respectively.

Resolving URLs

Sometimes declaration values are URLs relative to the style sheet location. A special construct, standard in CSS level2, allows you to create a URL from the base URL of the current style sheet. For example:

```
imageURL : url(images/icon.gif) ;
```

This declaration extends the path of the current style sheet URL with `images/icon.gif`. This construct is very useful for creating a style sheet with images located relative to it, because the URL remains valid even if the style sheet is cascaded or imported elsewhere.

CSS recursion

You are likely to want to specify a Java™ object as the value of a declaration. A simple convention allows you to recurse in the style sheet, that is, to define a new Java object which has the same style sheet but is unrelated to the current data model.

@# construct

Prefix the value with `@#` to create new Beans when required as shown in the following code example.

Creating a bean in a declaration

```
form {
      date : "@#dateBean" ;
      title : "CSS rules" ;
}
Subobject#dateBean {
      class : 'java.util.Date' ;
      time  : '23849291' ;
}
```

The `@#` operator extends the current data model by adding a dummy model object as the child of the current object. The object ID of the dummy object is the remainder of the string, beyond the `@#` operator. The type of the dummy object is `Subobject`. The dummy object inherits CSS classes and attributes from its parent.

The CSS engine creates and customizes a new subobject according to the declarations it finds for the dummy object. This means, in particular, that the Java class of the subobject is determined by the value of the `class` property. The newly created subobject becomes the value of the `@#` expression. In the declarations for the subobject, attribute references through the `@` operator refer to the attributes of the parent object.

Once the subobject is completed, the previous model is restored so that normal processing is resumed.

In code example *Creating a bean in a declaration*, a `java.util.Date` object is created, with the `time` property set to `23849291`. This new object is assigned to the `date` property of the `form` object.

@= and @+ constructs

There are two refinements of the `@#ID` operator:

♦ '**@=ID**'

  Using `@=ID` instead of `@#ID` shares the instance. The first time the declaration is resolved, the object is created as with the `@#` operator. But for all subsequent access to the same value, `@=ID` will return the same instance, the one created the first time, without applying the rules. Note that all instances created with `@=`are cleared when a new style sheet is applied.

♦ '**@+ID**'

Using @+ID instead of @#ID avoids useless creation. Basically @+ID customizes only the object currently assigned to the property, unless it does not exist or its class is not the same as the one defined in the #ID rule. In this case, the object is first created, then customized, and then assigned to the property, the same as with an @# construct.

The need for these refinements arises from a performance issue. The @#operator creates a new object each time a declaration is resolved. Usually a declaration is applied whenever a property changes. Under certain circumstances, the creation of objects may lead to expensive processing, so IBM® ILOG® JViews Diagrammer provides an optional mechanism to minimize the creation of objects during property changes.

@| construct

A CSS declaration value starting with @| is interpreted as an expression (see *Expressions*).

@ construct

A CSS declaration value that is exactly @ means cancel the property setting made in a previous rule. This construct is useful to prevent a property from being modified, especially when the default value is unknown. For example:

Canceling a property setting

```
node {
   width : 23 ;
}

node.fixed {
   width : @ ;
}
```

These two rules say that the width property value should be set to 23, unless the node has the CSS class fixed. Without the @ ability, the default value of width would have to be written down in the CSS.

Expressions

The value in a CSS declaration is usually a literal. However, it is possible to write an expression in place of a literal.

If the value begins with `@|`, then the remainder of the value is processed as an expression.

The syntax of the expressions, after the `@|` prefix, is close to the Java™ syntax. The expression type can be arithmetic (type `int`, `long`, `float`, or `double`), `Boolean`, or `String`. Examples:

```
@|3+2*5                -> 13
@|true&&(true||!true)  -> true
@|start+end              -> "startend"
```

An expression can refer to model attributes. The syntax is the usual one:

`@|@speed/100+@drift` -> 1/100 of the value of `speed` plus the value of `drift` , where `speed` and `drift` are attributes of the current object.

`'@|"name is: " + @name'`-> "name is: Bob", if the value of current object attribute `name` is "Bob." Note the use of quotes to keep the space characters. You could use the backslash (\) character instead, directly preceding the space characters to retain them. The backslah works to quote any character that directly follows it. Use of the backslash character makes sure that the character thus quoted is not interpreted by the expression parser.

The standard functions `abs()`, `acos()`, `asin()`, `atan()`, `ceil()`, `cos()`, `exp()`, `floor()`, `log()`, `pi`, `rint()`, `round()`, `sin()`, `sqrt()`, and `tan()` are accepted, as in, for example:

```
@|3+sin(pi/2) -> 4
```

There are some default functions in the `ilog.views.sdm.renderer` package: `concat`, `int`, `long`, `float`, `double`. The first one concatenates its parameters as String; the others evaluate basic numerical expressions (only the four operators +, -, *, / are allowed).

If the CSS engine encounters an error while it is resolving an expression, it silently ignores the declaration.

Custom functions

Users of CSS for Java can register their own functions, which can be part of an expression. A custom function must implement `IlvSDMCSSFunction`. This is an abstract class, but technically you should consider it just like an interface.

The signature of the main method is as follows:

```
public Object call(Object[] args, Class type, IlvSDMEngine engine,
                   Object node, Object target, Object closure);
```

- ♦ When a function is evaluated, the parameters are first resolved as subexpressions. Then the final values of parameters are passed to the `args` array.

- The parameter `type` is the expected type of the function, when known. A `null` value is possible. Implementation should take care to return an object of this type; otherwise the conversion will only be performed if it can be (that is, if it is a simple conversion between primitive types or to String).
- The parameters `engine`, `node`, and `target` are determined at invocation time as follows: `engine` is the current SDM engine, `node` is the current model object being customized, and `target` is the graphic object being customized. Not all functions need these parameters (see, for example, code example *Calling the custom function Average*).
- The parameter `closure` allows the caller to retrieve the context on exit from the method.

If an error occurs during the `call`, the exception will be reported and the current property setting will be canceled.

The following code example gives an example of a function that computes the average value of its parameters.

Custom function example: average of parameters

```
import ilog.views.sdm.renderer.IlvSDMCSSFunction;
import.ilog.views.sdm.IlvSDMEngine;

public class Average extends IlvSDMCSSFunction {
   //default contructor
   public Average() { }

   // Returns 'avrg'
   public String getName() {
      return "avrg";
   }

   // Returns ','
   public String getDelimiters() {
      return ",";
   }

   // Returns the average of arguments.
   public Object call(Object[] args, Class type, IlvSDMEngine engine,
                      Object node, Object target) {
      // Assume only double, for the sake of simplicity.
      double result = 0d;
      for (int i=0; i<args.length; i++) {
         if (args[i] != null) {
            result += Double.parseDouble(args[i].toString());
         }
      }
      result /= args.length;
      return new Double(result);
   }
}
```

The following code example shows an example of how to call a custom function, where the custom function is the `Average` class, which has the return value `avrg`. Note that this function does not require information from the engine.

Calling the custom function Average

```
node {
   width : @|avrg(@param1,@param2);
}
```

The following code example gives an example of a CSS function for SDM, which returns the graphic object (`IlvGraphic`) associated with the object whose ID is specified as argument.

Custom function example: Get Graphic Object From ID

```
class SDMGetGraphic extends IlvSDMCSSFunction {

   public SDMGetGraphic() {
   }

   public String getName() {
     return "getGraphicFromId";
   }

   public Object call(Object[] args, Class type, IlvSDMEngine engine,
                      Object node, Object target) {

     if (args.length < 1)
       throw new IllegalArgumentException("getGraphicFrom Id needs an id");

     String id = (String)args[0];
     IlvSDMModel model = engine.getModel();
     Object ref = model.getObject(id);
     if (ref == null)
       return null;
     IlvGraphic graphic = engine.getGraphic(ref,true);
     return graphic;

   }

   public Feedback[] invert(Object[] args, Object value,
                            IlvSDMEngine engine,
                            Object node) {

     return null;
   }
}
```

The following code example shows an example of calling the `getGraphicFromId` function to return the graphic object associated with the current object. Note that this function does require information from the engine to retrieve the current object and its associated graphic object.

Calling the custom function GetGraphicFromId

```
   graphic : @|getGraphicFromId(@__ID) ;
```

Registering custom functions

You must register custom functions before using them in a style sheet.

To register a function, you can simply call `registerFunction` in `ilog.views.sdm.renderer.IlvStyleSheetRenderer` (given an `IlvSDMEngine`, use `IlvRendererUtil.getRenderer` to find the active instance of `IlvStyleSheetRenderer`).

It is also possible to register a function in the style sheet, provided that the function is a JavaBean™ . You can set the property `functionList` to the list of custom functions, as class names separated by commas. This property is available in the `IlvStyleSheetRenderer` class. The following code example shows an example.

Registering a list of custom functions

```
StyleSheet {
   functionList : "myPackage.RevertFunction,tests.RandomFunction";
}
```

Expert feature: inverting expressions

In very special situations, expressions must be inverted. For example, if there is a rule:

```
node {
   x : @|log(@X) ;
}
```

and the representation of the node is moved horizontally, then the attribute named `X` in the model should be updated according to the expression and the new `x` value (here: `X` = 10^`x`).

If you can modify a representation object in the interface, the model attributes described in the style sheet may change. When the style sheet maps a property to an attribute directly, as in:

```
label : @name ;
```

the update is automatic. But if the mapping is realized through a function, as in:

```
label : @|concat(name is: ,@name) ;
```

then the function must be able to invert its operation (here, if the string evaluates to: “`name is: Bob`”, then the inverse function should set the `name` attribute to `"Bob"`).

Note: In this release, expressions cannot be inverted. However custom functions can be inverted, using the `invert` method.

Custom functions can be inverted using the `invert` method, which is defined as follows:

```
public Feedback[] invert(Object[] args, Object value, IlvSDMEngine engine,
   Object target)
```

The parameters `args`, `engine`, `target` have the same meaning as in `call`. The parameter `value` is the final value, and the method returns an array of couples (`name`, `value`). The `name` is the attribute name, and `value` is the value to set for this attribute. The result is an array when there are several attributes to update.

If a function cannot be inverted, then a null value should be returned.

The following code example shows how the Average function implements the `invert` method.

The Invert function for updating the data model from the style sheet

```
public Feedback[] invert(Object[] args, Object value, IlvSDMEngine engine,
                         Object target) {
   ArrayList result = new ArrayList();
   //The simplest to do is to set all attributes to the same final value.
   for (int i=0; i<args.length; i++) {
      // assume there is only @ construct.
      String att = args[i].toString();
      // Skip '@', which is the first character.
      att = att.substring(1);
      //Create and fill Feedback structure.
      Feedback f = new IlvSDMCSSFunction.Feedback();
      f.property = att;
      f.value = value;
      // Record feedback.
      result.add(f);
   }
   // Convert to Feedback[].
   return result.toArray(new IlvSDMCSSFunction.Feedback[result.size()]);
}
```

As you see in this example, there are two assumptions in the implementation of `invert` for `Average`:

- ♦ All attributes are set to the same value, which is the average value.
- ♦ Parameters of the `Average` function must be attributes only.

Most functions are difficult (sometimes even impossible) to invert. They usually need a compromise to support a minimum operation. But remember that `invert` needs to be implemented only when the interface allows you to modify the rendered object in such a way that the style sheet change needs to be reflected in the data model. Usually it is better to modify the model directly.

The default functions try at best to deliver a sensible result for `invert`. They accept model indirection (that is, `@name`) but no recursive calls to other functions.

Divergences from CSS2

Java™ objects are not HTML documents. The CSS2 syntax remains, so that a CSS editor can still be used to create the style sheet. However, the differences lead to adaptations of the CSS mechanism so that its power can be fully exploited and to some specific behavior.

Cascading

Cascading is explicit: the API offers a means of cascading style sheets. However, the `!important` and `inherit` tags are not supported for the sake of simplicity.

Pseudo-classes and pseudo-elements

The pseudo-class construct is fully implemented and used to represent renderer-specific states or GUI items. The list of predefined pseudo-classes and where they are used is as follows:

♦ `init` (is automatically enabled at creation time only. `:init` rules are not used if the bean is customized only)

♦ `selected` (any renderer)

♦ `collapsed` or `expanded` (`expandCollapseRenderer`)

♦ `<renderer_name>`, for example, `legendRenderer` (to specify that the rule applies only to rendering properties and not graphic object properties)

♦ `renderer` (to specify a property belonging to a renderer, as opposed to one with the same name belonging to the graphic object)

♦ `tree` (Workflow Modeler)

♦ `table (`Workflow Modeler)

You can add custom pseudo-classes with the method `IlvSDMEngine:setPseudoClasses`.

The CSS2 predefined pseudo-elements and pseudo-classes (`:link`, `:hover`, and so forth) are not implemented because they have no meaning in Java.

Attribute matching

The attribute pattern in CSS2 makes the following checks for strings: presence [`att`], equality [`att=`val], and inclusion [`att~=`val] . The `|=` operator is disabled.

For Java objects, there are the following numeric comparators `>`, `>=`, `<>`, `<=`, `<`, with the usual semantics.

There are also equal and not-equal comparators which make the distinction between string comparison and numerical comparison:

♦ Equal: "A==B" is true if and only if A and B are numerically equal (for example, 10 == 10.0); use "=" to test the equality of two Strings.

- Not-equal: "A~B" is true if and only if A and B are two different Strings (for example, "10" ~ "10.0"); use "<>" to test the inequality of two numbers.

Operators available in the attribute selectors

Operator	Meaning	Applicable To
A	present	strings
A=val	equals	strings
A~val	not equals	strings
A~=val	contains the word	strings
A==val	equals	numbers
A<>val	not equals	numbers
A<val	less than	numbers
A<=val	less than or equals	numbers
A>val	greater than	numbers
A>=val	greater than or equals	numbers

Syntax enhancement

CSS for Java requires the use of quotation marks when a token contains special characters, such as dot (.), colon (:), at sign (@), number sign (also known as hash sign, #), space ( ), and so on.

Quotes can be used almost everywhere, in particular to delimit a declaration value, an element type, or a CSS class with reserved characters.

The closing “;” is optional.

Null value

Sometimes it makes sense to specify a null value in a declaration. By convention, null is a zero-length string `''` or `""`. For example:

```
node.not-handled {
  class : '' ;
}
```

When a null class name is specified, no object is created at all, and no error is reported as it would be for a malformed class name.

The notation `''` is also used to denote a null array for properties expecting an array of values.

Empty string

The null syntax does not allow you to specify an empty string in the style sheet. Instead, you can create an empty string, as shown in the following code example.

```
node {
  label : @#emptyString ;
}

Subobject#emptyString {
  class : 'java.lang.String';
}
```

Better still, you can use the sharing mechanism to avoid the creation of several strings. The `@=` construct will create the empty string the first time only and will then reuse the same instance for all other occurrences of `@#emptyString`, see *Sharing an empty string* .

Sharing an empty string

```
node {
  label : @=emptyString ;
}

Subobject#emptyString {
  class : 'java.lang.String';
}
```

Customizing general nodes in the style sheet

Shows how to customize general nodes in the style sheet.

In this section

Overview
Provides some preliminary information for customizing the display of nodes.

Controlling the node’s shape
Describes the basic shapes available for general nodes.

Controlling the node’s skin
Describes the properties that control the node's skin.

Controlling the node’s border
Describes the properties used to control the node's border.

Controlling the node’s label
Lists the features used to control the node's label.

Controlling the node’s icon
Describes the properties used to control a node's icon.

Automatic resizing
Describes the properties used to control automatic resizing of a node.

Decorations
Describes the properties used to add decoration to a node.

Overview

To customize the display of nodes, start by setting default properties at the highest level, which is the node level.

Example of default properties for nodes

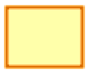
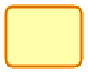
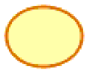
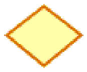
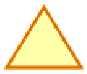
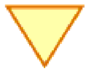
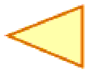
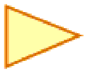
```
node {
   class        : ilog.views.sdm.graphic.IlvGeneralNode ;
   shapeType    : RoundRectangle ;
   shapeWidth   : 15 ;
   borderWidth  : 2 ;
   foreground   : 153,0,153 ; // dark magenta - border color
   fillColor1   : 198,226,255; // slate grey - inner color
}
```

In the customizer for a node, you can customize the node’s shape, skin, border, icon, label, and decorations.

Controlling the node’s shape

The basic shape of the general node is controlled by the `shapeType` property. The possible values are as listed in the following table.

Available shape types for a node

shapeType : “Rectangle”;	
shapeType : “RoundRectangle”;	
shapeType : “Ellipse”;	
shapeType : “Diamond”;	
shapeType : “TriangleUp”;	
shapeType : “TriangleDown”;	
shapeType : “TriangleLeft”;	
shapeType : “TriangleRight”;	
shapeType : “Marker”;	

If needed, you can set the shape of the node directly through the `setShape` method. This lets you use any custom shape. You can specify a custom shapes in any of the following ways:

♦ By referring to an SVG file in the style sheet:

```
node {
   shape : "url(myCustomShape.svg)";
}
```

♦ In Java code:

```
IlvGeneralPath myShape = new IlvGeneralPath(...);
generalNode.setShape(myShape);
```

The horizontal and vertical sizes of the shape are controlled through the properties `shapeWidth`, `shapeHeight`, and `shapeAspectRatio`.

There are basically two policies to set the width and height of the shape:

♦ You can set the properties `shapeWidth` and `shapeHeight`. In this case the aspect ratio of the shape will not be preserved. For example:

```
node {
    class       : "ilog.views.sdm.graphic.IlvGeneralNode";
    shapeType   : "RoundRectangle";
    shapeWidth  : "100";
    shapeHeight : "50";
}
```

♦ You can set the property `shapeWidth` to the desired width and the property `shapeAspectRatio` to the desired width/height ratio. For example:

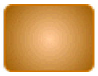
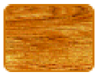
```
node {
    class            : "ilog.views.sdm.graphic.IlvGeneralNode";
    shapeType        : "RoundRectangle";
    shapeWidth       : "100";
    shapeAspectRatio : "2";
}
```

If you change the shape width in another rule, the aspect ratio will be preserved.

Controlling the node’s skin

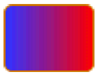
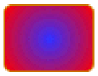
The following properties control the way colors are used in the shape: `fillStyle`, `fillColor1`, `fillColor2`, `fillStart`, `fillEnd`, `fillAngle`, and `fillTexture`.

The `fillStyle` property specifies the type of Paint object used to fill the shape. The possible values are as listed:

fillStyle : “SOLID_COLOR”;	
fillStyle : “LINEAR_GRADIENT”;	
fillStyle : “RADIAL_GRADIENT”;	
fillStyle : “TEXTURE”;	

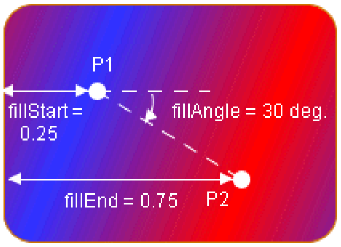
The `fillColor1` and `fillColor2` properties specify the colors used:

- ♦ In `SOLID_COLOR` mode, the shape is filled with `fillColor1`.
- ♦ In `LINEAR_GRADIENT` and `RADIAL_GRADIENT` modes, the gradient starts with `fillColor1` and ends with `fillColor2`:

fillStyle : “LINEAR_GRADIENT”; fillColor1 : “blue”; fillColor2 : “red”;	
fillStyle : “RADIAL_GRADIENT”; fillColor1 : “blue”; fillColor2 : “red”;	

In `LINEAR_GRADIENT` and `RADIAL_GRADIENT` modes, the `fillStart`, `fillEnd`, and `fillAngle` properties define the geometry of the gradient. A gradient is defined by two points called P1 and P2. The following figures explain the meaning of the properties.

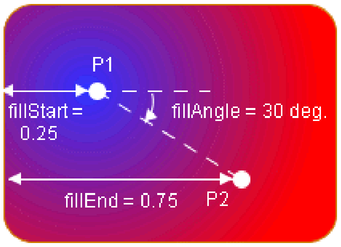
The following figure shows the geometry for a linear gradient.



Linear gradient

Note that the linear gradient is always in “reflect” mode, so the colors go back and forth from `fillColor1` to `fillColor2` outside the (P1, P2) segment.

The following figure shows the geometry for a radial gradient.



Radial gradient

The `fillTexture` property specifies the URL of an image file to use as a texture in `TEXTURE` mode.

Controlling the node’s border

The stroke (that is, the border) of the shape is controlled by the properties `strokeColor`, `strokeWidth`, `strokeDashArray`, `strokeEndCaps`, `strokeLineJoins`, `strokeMiterLimit`, and `strokeDashPhase`.

The `strokeColor` property sets the color used to paint the stroke.

The other properties are used to create an instance of `java.awt.BasicStroke`:

- `strokeWidth` specifies the width of the stroke.
- `strokeDashArray` is used to create dashed or dotted strokes. It is an array of floating-point values that specify the lengths of the alternate painted and transparent segments.
- `strokeEndCaps` specifies the shape of the ends of the dash segments.
- `strokeLineJoins` and `strokeMiterLimit` specify how the stroke looks at the angles between two segments.

The following code example shows how to create a blue dashed stroke, with rounded segment ends, visible segments that have a length of 4, and transparent segments that have a length of 2, in the style sheet.

Styling rule for the border (stroke) of a node

```
node {
   strokeColor     : "blue";
   strokeDashArray : "4,2";
   strokeEndCaps   : "CAP_ROUNDS";
}
```

For more details of the stroke-related properties, see the documentation of the `BasicStroke` class in the Java™ documentation.

Controlling the node's label

You can make use of the following features for a label on a node:

- *Label string*
- *Multiline label*
- *Label position*
- *Autowrap*
- *Truncation*
- *Zoom*
- *Label font, color, and anti-aliasing*

Label string

The `label` property controls the string displayed by the node's label.

You can set the label to the empty string (`""`) if you do not want to display any label on the node.

Multiline label

A label can have several lines. To define a multiline label, simply put newline ('`\n`') characters in the value of the `label` property to separate the lines.

The alignment of multiline labels is controlled by the property `labelAlignment`, which can take the values `Left` (left-aligned), `Center` (centered), or `Right` (right-aligned).

The spacing between the lines of multiline labels is controlled by the property `lineSpacing`.

Label position

The `labelPosition` property controls the placement of the label relative to the shape. This property can take the values defined by the interface `IlvDirection`. For example, setting `labelPosition` to `Top` places the label above the shape. `Center` places the label inside the shape. The default position is `Bottom`..

The spacing between the shape and the label is controlled by the property `labelSpacing`.

Autowrap

If the `labelMode` property is set to `WORD_WRAP`, the label is automatically cut into several lines to fit into the width of the shape.

You can choose the characters where a break due to word wrapping is allowed by setting `wordWrapChars`. In autowrap mode, the newline characters contained in the label are ignored.

The property `labelMargin` controls the margin to leave between the border of the shape and the label when word wrapping is active:

- If the label is inside the shape, the label is kept narrower than the shape by two times the margin.
- If the label is outside the shape, the label is allowed to be wider than the shape by two times the margin.

Truncation

If the `labelMode` property is set to `TRUNCATE`, the label is automatically truncated to fit into the width of the shape. This is different from autowrap mode: the label is not cut into several lines, but simply truncated, and the truncated label always has a single line.

The property `labelMargin` controls the margin to leave between the border of the shape and the (truncated) label, as in `WORD_WRAP` mode.

The end of the label is replaced by the string `"..."` (You can change this suffix using the method `setTruncatedLabelSuffix`.

Zoom

If the `labelZoomable` property is set to `"true"`, the label grows when the view is zoomed in and shrinks when the view is zoomed out. If the `labelZoomable` property is set to `"false"`, the label size stays constant.

You can choose to make the label invisible when the view is zoomed in or out past a certain zoom level. The property `minLabelZoom` specifies the minimum zoom level past which the label will no longer be visible when zooming out. The property `maxLabelZoom` specifies the maximum zoom level past which the label will no longer be visible when zooming in.

The `labelScaleFactor` lets you apply a scale factor to the label.

Label font, color, and anti-aliasing

The properties `labelFont` and `labelColor` control the font and color of the label.

The property `labelAntialiasing` determines whether the label should be drawn using anti-aliasing or not.

Controlling the node’s icon

To display an icon inside the shape, set the property `icon` to the URL of the icon to display.

You can define the icon URL in the style sheet as:

♦ An image file in any format supported by the Java™ VM:

```
node {
   icon : "url(myIcon.jpg)";
}
```

♦ An SVG file:

```
node {
   icon : "url(myIcon.svg)";
}
```

♦ An IBM® ILOG® JViews prototype:

```
node {
   shape : "url(myPrototypeLibrary.ivl#myPrototype)";
}
```

Set the icon property to the empty string if you do not want an icon.

The icon is always displayed inside the shape (or centered on top of the marker if the shape type is `Marker`).

If the label is inside the shape (`labelPosition : "Center"`), the position of the icon relative to the label is controlled by the property `iconPosition`, which can be set to any direction defined by the interface `IlvDirection`. For example, `iconPosition : "Left"` places the icon to the left of the label.

Automatic resizing

The general node can automatically compute the size of its shape according to the size of the labels and the icon.

You can control autoresizing in the vertical and horizontal directions independently through the properties `horizontalAutoResizeMode` and `verticalAutoResizeMode`. These properties accept the following values:

NO_AUTO_RESIZE	Autoresize is disabled.
EXPAND_ONLY	The node is allowed to grow in the specified direction, but not to shrink.
SHRINK_ONLY	The node is allowed to shrink in the specified direction, but not to grow.
EXPAND_OR_SHRINK	The node is allowed to expand or to shrink as needed.

If horizontal autoresizing and word wrapping are used at the same time, the general node cannot use the size of the label to compute the shape’s width because word wrapping uses the shape’s width to cut lines. In this case, the label will be word wrapped such that its bounding box is approximately square.

You can control how much space will be left between the border of the shape and its contents (label and icon) using the properties `horizontalAutoResizeMargin` and `verticalAutoResizeMargin`.

Decorations

The general node can display “decoration” graphics in its top-left corner. Decorations are small graphic objects used to represent the state of an object. For example, you could use an icon to show that an error has occurred during the processing of a workflow activity.

To add a decoration to a node, set the `decorations` property. The value of the property is an array of `IlvGraphic` objects.

The following code example shows a style sheet extract that adds a decoration which is an `IlvShadowLabel` object.

Adding a decoration to a node in the style sheet

```
node[status="error"] {
    decorations[0] : "@Subobject#errorDecoration";
}
Subobject#errorDecoration {
    class      : "ilog.views.sdm.graphic.IlvGraphicFactories$ShadowLabel";
    IlvRect    : "0,00,20,20";
    label      : "E";
    foreground : "red" ;
    background : "white" ;
}
```

You can add up to three decorations to a node. The decorations are always displayed in the upper-left corner of the node. The upper-left corner of the first decoration is aligned with the upper-left corner of the node. If several decorations are used, the subsequent decorations are shifted to the right.

Customizing general links in the style sheet

Shows how to customize general links in the style sheet to affect its colors, borders, ends and joins, decorations, arrows, and hooks.

In this section

Controlling the link's look
Lists the different ways that links can be customized.

Obtaining color effects
Describes the properties used to control color effects in links.

Controlling link decorations
Describes the properties used to control decoration in links

Controlling arrows
Describes the properties used to control arrow effects in links

Controlling extra effects
Lists the used used to control extra effects in links.

Summary of link properties
Summarizes the properties used to control links

Controlling the link’s look

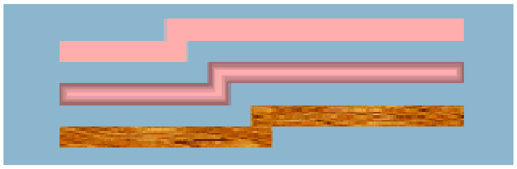
Controlling the link’s look involves:

- *Modes*
- *Adding a border*
- *Different border effects*
- *Line (stroke) ends and joins*
- *Curves*
- *Dashes*

In the description of each of these items, the figure shows three possible customizations and the code that follows the figure shows the style sheet setting for each customization in the same order.

Modes

The link has three major different looks, associated with the property `mode`. For the first and second looks, the `foreground` property sets the main color.

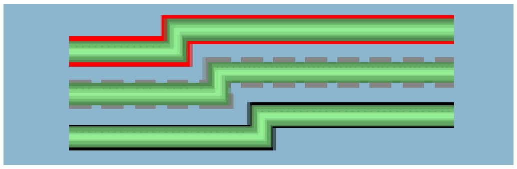

Three possible link modes

Basic link settings and three possible modes

```
link {
        lineWidth   : 10
        foreground  : pink ;
}

link.top {
        mode      : MODE_UNICOLOR ;
}

link.center {
        mode      : MODE_GRADIENT ;
}

link.bottom {
        mode      : MODE_TEXTURE ;
        texture   : 'file:/home/kaplan/JViews30/\
bin/composer/images/textures/wood2.gif' ;
}
```

Adding a border

A border is painted when the `borderWidth` property is greater than `0`, its default value. The default border color is black, and two other properties control the line style (for example, dashes).



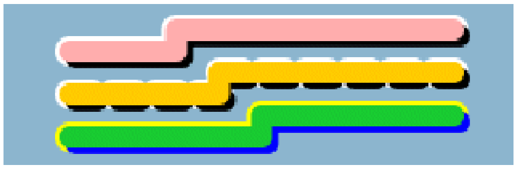
Three possible link borders

Basic link properties and three possible borders

```
link {
        lineWidth   : 10 ;
        foreground  : 144,238,144 ;
        endCap      : CAP_BUTT ;
}

link.top {
        borderWidth   : 4 ;
        borderUpColor : red ;
}

link.center {
        borderWidth   : 4 ;
        borderUpColor : gray ;
        borderStyle   : 10,5 ;
}

link.bottom {
        borderWidth : 2 ;
}
```

Different border effects

The border can have two colors: one for the upper edge and one for the lower edge.



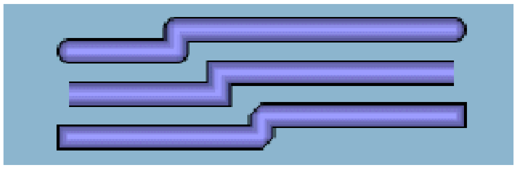
Using two colors in link borders

Rounded two-color links

```
link {
        lineWidth       : 10 ;
        endCap          : CAP_ROUND ;
        lineJoin        : JOIN_ROUND ;
        borderWidth     : 4 ;
        borderUpColor   : white ;
        borderDownColor : black ;
        mode            : MODE_UNICOLOR ;
}

link.top {
        foreground   : pink ;
}

link.center {
        foreground   : orange ;
        borderStyle  : 10,10 ;
}

link.bottom {
        borderUpColor   : yellow ;
        borderDownColor : blue ;
        foreground      : 30, 200, 50 ;
}
```

Line (stroke) ends and joins

`IlvGeneralLink` inherits from `IlvPolylineLinkImage`. The default stroke parameters are `JOIN_MITER` and `CAP_SQUARE`. A miter join is a sharp join formed by extending one edge of each link. You can change the end cap and join styles, as shown in the following figure.



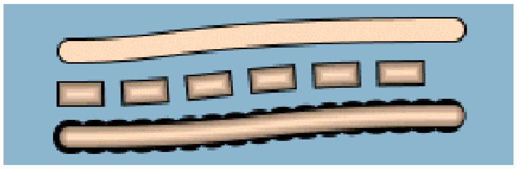
Three possible end and join style combinations

Basic link properties and three possible end and join combinations

```
link {
        lineWidth   : 10 ;
        foreground  : 154,154,255 ;
        borderWidth : 2 ;
}

link.top {
        endCap   : CAP_ROUND ;
        lineJoin : JOIN_ROUND ;
}

link.center {
        endCap   : CAP_BUTT   ;
        lineJoin : JOIN_MITER ;

}

link.bottom {
        endCap   : CAP_SQUARE ;
        lineJoin : JOIN_BEVEL ;
}
```

Curves

The `curved` property uses the link points to feed a Bezier function which renders a curved link. Intermediate points show the path for the Bezier computation. With two points, a standard deviation applies, that is, at 1/4 before the end of the link. The `curved` value is a floating-point value between `0f` and `1f`. A value of `0` means no curve at all (the default), and a value of `1` means the sharpest curve. Use a value of `0.65f` for an attractive curve, see the following figure.



An attractively curved link

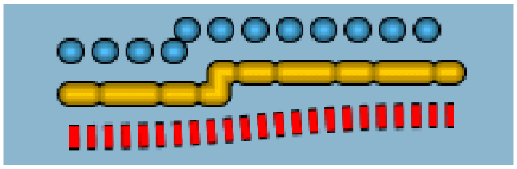
Basic line properties and a curve in three different line styles

```
link {
        lineWidth   : 10 ;
        endCap      : CAP_ROUND ;
        lineJoin    : JOIN_ROUND ;
        foreground  : 255,218,185 ;
        borderWidth : 2 ;
        curved      : 0.65 ;
}

link.top {
        mode : MODE_UNICOLOR ;
}

link.center {
        endCap    : CAP_SQUARE ;
        lineStyle : 10,20 ;
}

link.bottom {
        borderStyle : 1,10 ;
        borderWidth : 6 ;
}
```

Dashes

Dashes provide interesting effects when combined with `endCap` values. Dashes are controlled by the lineStyle property. They are expressed as a float array. Alternate entries in the array represent lengths of the opaque and transparent segments of the dashes. Note that the `lineStyle` specification is not affected by zooming.

Note that the dash specification also applies to the border (as shown in the following figure) unless the `borderStyle` property overrides it (not shown).



Three uses of dashes for links

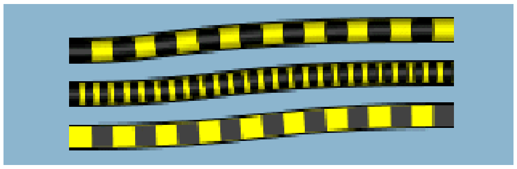
Basic link properties and three links with dashes (one a curve)

```
link {
        lineWidth   : 10 ;
        endCap      : CAP_BUTT ;
        lineJoin    : JOIN_ROUND ;
        borderWidth : 2 ;
}

link.top {
        foreground  : #55BEF3  ;
        lineStyle   : 1,15  ;
}

link.center {
        foreground : orange ;
        lineStyle  : 10,8,20,8 ;
}

link.bottom {
        foreground : red ;
        mode       : MODE_UNICOLOR ;
        endCap     : CAP_BUTT ;
        lineStyle  : 4,4 ;
        curved     : 0.65 ;
}
```

Obtaining color effects

The alternateColor property gives the link a striped appearance. The length of each stripe is equal to the thickness of the link by default. You can specify a different stripe length with the lineStyle property.

The lineStylePhase property sets the initial offset. If no lineStyle is specified, the phase is proportional to twice the line width. In the following example, the bottom link starts the alternate color one segment later than the top one.


Three curved links with stripes

Striped links

```
link {
        lineWidth      : 10 ;
        foreground     : yellow ;
        endCap         : CAP_BUTT ;
        lineJoin       : JOIN_ROUND ;
        alternateColor : darkGray ;
        borderWidth    : 2 ;
        curved         : 0.65 ;
        }

link.top {

}

link.center {
        lineStyle  : 4,3 ;
}

link.bottom {
        mode           : MODE_UNICOLOR ;
        lineStylePhase : 1 ;
}
```

Controlling link decorations

`IlvGeneralLink` allows you to place an `IlvGraphic` object onto a link, at a relative position. You can place the decoration above the link or beside the link (the default).

Note that the syntax to enter the `decorationPosition` value is an array. You can enter all the values at once as a comma-separated list of values or enter the array elements one by one.

Another array property, `decorationOptions`, alters the way in which the decoration is displayed. This property is a bit set of values, so you can specify several options at the same time using the vertical bar operator `|` as follows:

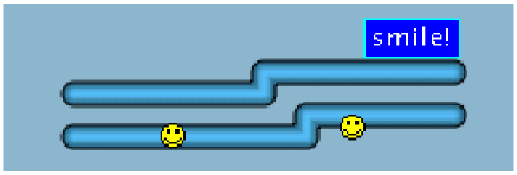
- `DECORATION_FIXED_SIZE`: If set, the decoration will not zoom.
- `DECORATION_ROTATE`: If set, the decoration is rotated to follow the link shape.
- `DECORATION_ANIMATE`: If set, the decoration will move along the link. The speed is defined by the property `animateSpeed`.
- `DECORATION_NOTHING`: Means default values (zoomable, no rotation, no animation).

To place the decoration over the link, set the option `DECORATION_OVER` , then set additional options to control the position of the decoration with respect to the center of the link:

- `DECORATION_ANCHOR_TOP`: Shifts the decoration below the link at the center.
- `DECORATION_ANCHOR_BOTTOM`: Shifts the decoration above the link at the center.
- `DECORATION_ANCHOR_LEFT`: Shifts the decoration to the right of the link at the center.
- `DECORATION_ANCHOR_RIGHT`: Shifts the decoration to the left of the link at the center.

It is possible to mix options: `DECORATION_ANCHOR_TOP|DECORATION_ANCHOR_LEFT` places the top-left corner of the decoration at the link center. The default behavior places the center of the decoration at the center of the link, and you can apply an offset with the x, y values for the decoration bounding box.

The following figure shows three decorations, one is a label and two are icons on the same link.



Decorations on Links

Decorations on links

```
link {
        foreground   : #55BEF3 ;
        lineWidth    : 10 ;
        endCap      : CAP_ROUND ;
        lineJoin    : JOIN_ROUND ;
        borderWidth : 2 ;
}

link.top    {

        decorations[0]      : @#label ;
        decorationPositions : 0.9 ;
}

link.center {
        visible : false
}

link.bottom {
        decorations[0]   : @#smiley ;
        decorations[1]   : @#smiley ;
  decorationOptions[0]   : DECORATION_OVER ;
  decorationOptions[1]   : DECORATION_OVER|DECORATION_ANCHOR_TOP ;
     decorationPositions : 0.25,0.75  ;
}
```

```


Subobject#label {
        class            :\ 'ilog.views.sdm.graphic.
IlvGraphicFactories$ZoomableLabel' ;
        label            : "smile!" ;
        antialiasing     : true ;
        leftMargin       : 5 ;
        rightMargin      : 5 ;
        topMargin        : 5 ;
        bottomMargin     : 5 ;
        foreground       : white ;
        borderOn         : true ;
        stroke           : @=stroke ;
        strokePaint      : @=borderPaint ;
        backgroundOn     : true ;
        backgroundPaint  : @=bgPaint ;
}

// border is cyan
Subobject#borderPaint {
        class            : ilog.views.sdm.graphic.IlvPaint ;
        color            : cyan ;
}

// background is blue
Subobject#bgPaint {
        class            : ilog.views.sdm.graphic.IlvPaint ;
        color            : blue ;
}
```

```
// stroke for the text
Subobject#stroke {
        class            : ilog.views.sdm.graphic.IlvBasicStroke ;
        lineWidth        : 1 ;
}

// smiley icon
Subobject#smiley {
  class         : "ilog.views.sdm.graphic.\
IlvGraphicFactories$Icon"
  IlvRect       : 0,0,12,12 ;
  imageLocation : file:smiley.gif ;
}
```

Note the convenient property called `label` that quickly accesses the first decoration implementing the `IlvLabelInterface`. If no decoration is found, a simple `IlvZoomableLabel` is created at the first available slot of the current decorations array.

Controlling arrows

You can set one of four modes on a link to represent an arrow:

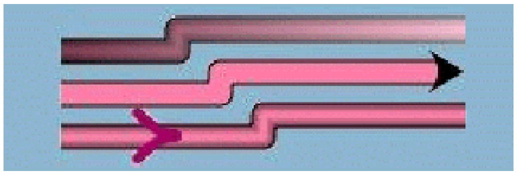
- The `ARROW_FILL` mode value (the default) draws a filled triangular arrowhead.
- The `ARROW_OPEN` mode value draws a two-winged arrow.
- The `ARROW_GRADIENT` mode value shows an oriented link by smoothly varying the luminosity along the link. The link appears darker near the source object and brighter near the destination object.
- The `ARROW_DECORATION` mode value delegates the task of displaying the arrow to one of the link decorations.

Drawing an arrow

In the first two modes, the `arrowPosition` property controls the position of the arrowhead along the link. Its value is a floating-point value between 0 and 1. A value of 0 means the start of the link, and 1 means the end (the default). The arrow direction is obviously aligned with the link segment to which it is attached.

The property `arrowRatio` controls the size of the arrow, which is proportional to the link width. A floating-point value of 0.5 means the arrow has the same size as the link. The default value, 1, means the arrow is twice the link width. This property only makes sense for the first two arrow modes.

The default color for an arrow is black, but you can set a different color with the property `arrowColor`.


Arrowheads and direction

Basic link properties and three ways to show direction

```
link {
        lineWidth   : 10 ;
        endCap      : CAP_BUTT ;
        lineJoin    : JOIN_ROUND ;
        foreground  : 255,130,171 ;
        borderWidth : 2 ;
        oriented    : true ;
}

link.top {
        arrowMode : ARROW_GRADIENT ;
}
```

```
link.center {
        mode          : MODE_UNICOLOR ;
        arrowPosition : 1f ;
        arrowMode     : ARROW_FILL ;
}

link.bottom {
        arrowColor    : #A3056E ;
        arrowPosition : .2 ;
        arrowMode     : ARROW_OPEN ;
}
```

Using a decoration as an arrowhead

The `ARROW_DECORATION` mode value delegates the responsibility for showing an arrow to a decoration.

`IlvGeneralLink` can help display a particular kind of arrow, for example, the kind used in UML drawings.

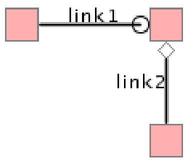
The arrowhead is a simple shape (such as a circle, triangle, or diamond), located at the end of the link. The link does not overlap the shape, nor go to the middle of the shape, nor change at all. Whatever the direction of the last link segment touching the target node, the shape is always toward the link. Since the decorations are `IlvGraphic` objects, that is, basically rectangular objects, this special mode gives better results with orthogonal links.

You can only use this mode if the following conditions are met:

- The `arrowMode` property is set to `ARROW_DECORATION`.
- One of the link decorations has its position set to 1.
- This decoration sets the option `DECORATION_OVER`.
- This decoration sets one of the options `DECORATION_xxx_RETRACT_AT_END`,

  where `xxx` can be `FULL`, `HALF`, or `NO`.

The following figure shows an example of two links in `ARROW_DECORATION` mode, connected to a square node. The horizontal link is ended by a circle; the line continues to the middle of the circle. The vertical link is ended by a diamond; the line stops at the beginning of the diamond.



Two decoration arrowheads

The following code example shows the styling rules used to generate the node, links and arrowheads.

Decorations as arrowheads on links to a node

```
/////////////
// A very simple node
node {
  class                   : ilog.views.sdm.graphic.IlvGeneralNode
  fillColor1              : pink;
  foreground              : gray;
}

////////////
// Link definition

link {
  class                   : ilog.views.sdm.graphic.IlvGeneralLink;
  oriented                : true;
  decorations[1]          : @=arrow; // see Subobject#arrow
  decorationPositions[1] : 1;
  decorationOptions[1]    : 'DECORATION_OVER|DECORATION_HALF_RETRACT_AT_END';
  arrowMode               : ARROW_DECORATION;
  label                   : @id;
  lineWidth               : 1.5;
  endCap                  : CAP_BUTT;
  foreground              : black;
}
```

```


Subobject#arrow {
  class                  : ilog.views.sdm.graphic.IlvGraphicFactories$Ellipse;

  IlvRect                 : 0,0,10,10; // a circle
}

///////////////
// The vertical link inherits from "link { ... }" rule, and
// overwrites some options

#link2 {
  decorations[1]          : @=arrow2; // see Subobject#arrow2
  decorationOptions[1]    : 'DECORATION_OVER|DECORATION_FULL_RETRACT_AT_END';
}

Subobject#arrow2 {
  class                   : ilog.views.sdm.graphic.IlvGraphicFactories$Marker;

  IlvRect                 : 5,5,5,5;
  type                    : 2; // a diamond
}
```

Controlling extra effects

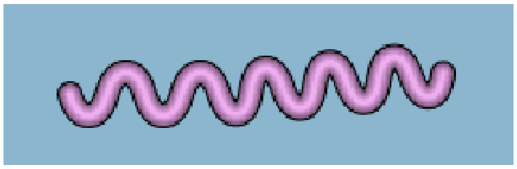
The extra effects available for a link are:

- ♦ *Waves*
- ♦ *Animation*
- ♦ *Zoom*
- ♦ *Neon effect and road effect*

Waves

The wave effect is very effective for representing a wireless connection. The wave specification consists of two numbers: the wave amplitude and its period, in pixels. The property type is a String where two integers separated by "/" represent, respectively, amplitude and period. The effect renders best with straight lines but remains compatible with any shape.

The wave effect can also be mixed with dashes, border, arrow, and so on.



The wave effect for a link with a border

Basic link properties and the wave property

```
link {
        lineWidth   : 10 ;
        endCap      : CAP_ROUND ;
        lineJoin    : JOIN_ROUND ;
        foreground  : 238,174,238 ;
        borderWidth : 2 ;
}

link.top {
        visible : false ;
}

link.center {
        borderWidth : 2 ;
        wave        : 20/30 ;
}

link.bottom {
        visible : false ;
}
```

Animation

You can animate a link with dashes (see the `lineStyle` property) or alternate colors (see the `alternateColor` property). The animation consists of incrementing the `lineStylePhase` value at regular intervals, so that the pattern is shifted at each animation frame. The current implementation updates the pattern every 500 ms.

The `animateSpeed` property controls animation of the link and how much the phase is incremented. If the value is `0`, the animation is stopped. Otherwise, the value must be between `0f` and `1f`, and represents a fraction of dash pattern length. For example, `0.1` means that ten frames pass before you see the first frame again. Note that `0.9` represents the same increment but in the backward direction.

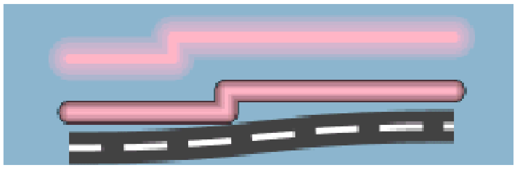
Zoom

`IlvGeneralLink` inherits from `IlvLinkImage`, and keeps its zoom behavior. In particular, the `maximumLineWidth` property sets the maximum size of the line. Also, the link is zoomable only if one edge node is zoomable. Note that the link border thickness does not zoom. The border line style and the link line style follow the zoom level of the link.

Neon effect and road effect

`MODE_NEON` is a minor mode which is a variation of `MODE_GRADIENT`. It displays the link with transparent colors, giving a glowing effect (this works best with larger links and with darker backgrounds). The border is automatically disabled in this mode. An example of using Neon mode is to mark link selection, as shown in the top link in figure*The wave effect for a link with a border*. (The second link shows the same link when it is not selected.)

Previous examples already demonstrated some of the many possibilities that the link appearance offers. By combining some of the properties, you can achieve some fancy effects, such as the “road” link in figure *The wave effect for a link with a border*, where the large border is the same gray color as the link foreground, and the white `alternateColor` looks like the middle line of a road.



The neon effect

Selected (neon) and not selected links, and a road link

```
link {
        lineWidth  : 10 ;
        lineJoin   : JOIN_ROUND ;
        endCap     : CAP_ROUND ;
        foreground : 255,181,197 ;
}

link.top {
        mode      : MODE_NEON ;
        lineWidth : 20 ;
}

link.center {
        borderWidth : 1 ;
}
```

```


link.bottom {
        mode           : MODE_UNICOLOR;
        endCap         : CAP_BUTT ;
        lineWidth      : 15 ;
        foreground     : darkGray ;
        lineStyle      : 20,10;
        lineStylePhase : 5 ;
        alternateColor : white ;
        borderWidth    : 12 ;
        borderUpColor  : darkGray ;
        curved         : 0.65 ;
}
```

Summary of link properties

The following table lists all available properties of `IlvGeneralLink` and their descriptions.

IlvGeneralLink properties

Property Name	Type	Effect	Default	Comment
alternateColor	Color	alternate colored segments	null	Segment colors are `foreground` and `alternateColor`. Segments are the width of the link unless the `lineStyle` property is specified.
arrowColor	Color	color of arrow	Color.black	Sets the arrow color; the default is black.
arrowMode	int	style of arrow	ARROW_FILL	`ARROW_FILL` gives a filled triangular shape, `ARROW_OPEN` gives a 2-arms shape, and `ARROW_GRADIENT` gives a lighter color near the target. Set oriented to enable an arrow. `ARROW_DECORATION` delegates the display of the arrow to a decoration.
arrowPosition	float	position of arrow	1f	`ARROW_FILL and ARROW_OPEN` modes only. It sets the position of the arrow along the link. The default is 1f, that is, at the target of the link. When exactly at 1f, the link is not painted until the very end, so it does not overlap the link.
arrowRatio	float	relative size of the arrow	1f	`ARROW_FILL and ARROW_OPEN` modes only. It sets the size of the arrow according to the link width. The ratio value is multiplied by twice

Property Name	Type	Effect	Default	Comment
				the link width to obtain the arrow size.
borderDownColor	Color	border down color	null	The down side color of the border.
borderStyle	float [ ]	dash style	null	Overrides `lineStyle` if defined.
borderStylePhase	float	dash offset	0f	Overrides `lineStylePhase` only when `borderStyle` is defined.
borderUpColor	Color	border color	null	The color of the border. If `borderDownColor` is specified, it affects only the upside border of the link.
borderWidth	float	border width	0f	The width of the link border.
curved	float	curved lines	0f	Inner points are Bezier control points. For 2 points, a standard deviation is applied. Values below 0f and over 1f are ignored. Set to 0 to disable curves, use 0.65 for best results.
decorations	ilog.views.IlvGraphic[]	puts IlvGraphic onto the link	null	Style sheet entry can be a @# construct that generates an `IlvGraphic` (works only with an indexed setter) .
decorationPositions	float[ ]	sets relative location of the decorations	null	Values are between 0.0 and 1.0.
decorationOptions	int[ ]	displays options for decorations	null	Options are: `DECORATION_NOTHING` for default, an OR-ed set of `DECORATION_ANIMATE` to enable animation according to `animateSpeed` value, `DECORATION_ROTATE` to rotate the decoration according to the link shape, or `DECORATION_FIXED_SIZE` to

Property Name	Type	Effect	Default	Comment
				prevent the decoration from zooming. For a decoration over the link, use `DECORATION_OVER` and `DECORATION_ANCHOR_[TOP|BOTTOM|LEFT|RIGHT]` to set the position of the decoration with respect to the link center. `DECORATION_[FULL	HALF	NO]_RETRACT_AT_END]` is used in `ARROW_DECORATION` special mode to display the decoration at the end of the link.
endCap	int	link termination (and dashes) style	CAP_SQUARE	See `IlvStroke`.
foreground	java.awt.Color	main color	null	Value may be a decimal triple, #-form, or any of the standard Java™ colors, for example, `12,55,200`, `#E023B3`, `cyan`. No alpha is allowed.
label	String	link label	null	Convenient property to access the first decoration that is a label. If none is found, an `IlvZoomableLabel` is created

Property Name	Type	Effect	Default	Comment
				at the first available slot in the decoration array.
lineJoin	int	corner style	JOIN_MITER	See `IlvStroke`.
lineStyle	float [ ]	dash style	null	See `IlvLinkImage`.
lineStylePhase	float	dash offset	0f	See `IlvLinkImage`.
lineWidth	float	line width	0f	The width of the link (border excluded).
minimumLineWidth	float	zoom min. width	0f	The minimum width of the link.
maximumLineWidth	float	zoom max. width	0f	See `IlvLinkImage`.
mode	int	main look	MODE_GRADIENT	`MODE_UNICOLOR` displays the foreground color; `MODE_GRADIENT` displays a brightness gradient derived from the foreground, giving the look of a pipe; `MODE_NEON` displays a transparent gradient derived from the foreground, giving the look of a flashy neon; and `MODE_TEXTURE` displays the texture from the texture property.
oriented	boolean	flag	false	True enables an arrow.
texture	IlvTexture	texture	null	The texture for `MODE_TEXTURE`. The property editor expects a URL to build the texture, so use the string representation of a URL in the style sheet to set this property.
visible	boolean	show the link	true	
wave	String	wavy stroke	0/0	First part is the amplitude, second part is the period. It works best with direct lines.

Using and adding renderers

Explains what a renderer is and how to enable and customize one, lists all predefined renderers with their properties, and gives examples of writing your own renderer in Java™ code and integrating it through Java code or through the style sheet.

In this section

About renderers
Provides background information on renderers.

Using renderers in the style sheet
Shows how to define renderers in the style sheet.

Predefined renderers
Describes the predefined renderers provided in IBM® ILOG® JViews Diagrammer. The following predefined SDM renderers are described, with their global properties and their rendering properties if any

Adding your own renderer
Explains how to extend the rendering features of SDM. Shows how to add your own renderer.

The Flag renderer example
Provides an example of a custom renderer that displays a flag on the node that remains visible even if the object is hidden.

Configuring renderers in Java code
Shows how to customize a renderer using Java™ .

Support for renderers in the Designer
Lists the renderers supported in the Designer.

About renderers

A renderer is a Java™ class that helps to manage the graphical representation of your business data.

ILOG JViews Diagrammer supplies many predefined renderers. Several renderers are usually active at the same time.

The `StyleSheet` renderer is responsible for creating and customizing graphical objects and is always active. The `GraphLayout` renderer applies a layout algorithm and is often active.

Other renderers are created by the SDM engine if they are explicitly enabled in the style sheet.

Using renderers in the style sheet

Shows how to define renderers in the style sheet.

In this section

Enabling a renderer
Describes how to enable a renderer in the style sheet.

Customizing a renderer
Describes how to customize a renderer in the style sheet.

Using rendering properties on objects
Provides information on customizing the behavior of a renderer on specific objects.

Enabling a renderer

To enable a renderer, simply set the declaration `<renderer-name> : true ;` in a style rule with a selector that matches `SDM`.

The following code example shows a style rule that enables the `Decoration` and `LinkLayout` renderers.

```
SDM {
   Decoration : true;
   LinkLayout : true;
}
```

Customizing a renderer

To customize one of the renderers, you write a style rule with a selector that is the renderer name and declarations that set properties of the renderer. Each renderer is implemented by a JavaBean™ , and you can set any property of the Bean in the style rule.

For example, the `LinkLayout` renderer is implemented by the class `IlvLinkLayoutRenderer`. This class defines a property called `performingLayoutOnZoom` which requires the layout algorithm to be reapplied each time the view is zoomed in or out. The following code example shows a style rule that customizes the `LinkLayout` renderer to reapply the layout algorithm in this way.

Customizing a renderer in the style sheet

```
LinkLayout {
   performingLayoutOnZoom : true;
}
```

Some renderers have a “default” property that you can set directly in the SDM rule instead of setting it in the renderer’s customization rule. For example, the default parameter of the `GraphLayout` renderer is the name of the layout algorithm to apply. The following code example shows how to customize this renderer directly in the SDM rule.

Customizing a renderer directly

```
SDM {
   GraphLayout : "Hierarchical";
}
```

Direct customization in the SDM rule is a shortcut for the longer way, which is shown in the following code example.

Customizing a renderer the longer way

```
SDM {
   GraphLayout : true;
}
GraphLayout {
   graphLayout : "Hierarchical";
}
```

Using rendering properties on objects

In addition to the general rules for renderers, you can write specific rules to customize the behavior of a renderer on a per-object basis through *rendering properties*. Rendering properties can be seen as additional properties of the graphic objects which you can set to tell a renderer how it should handle a particular object.

For example, the `GraphLayout` renderer defines a rendering property called `Fixed`. When this property is set to `true`, the `GraphLayout` renderer will keep the position of the object unchanged. The following code example shows a style rule that sets the `Fixed` property.

Customizing the layout of a type of node

```
node.participant {
   shapeType   : "Ellipse";
   Fixed : "true";
}
```

This rule says that the nodes of type `participant` must not be moved when a graph layout is applied. As you can see, you can mix "real" properties of graphic objects (such as `shapeType`) with rendering properties (such as `Fixed`).

Important: By convention, most rendering properties start with an uppercase letter, whereas the properties of graphic objects start with a lowercase letter.

The following predefined renderers do not have rendering properties:

- Coloring
- Decoration
- LabelLayout
- Legend

If a rendering property conflicts with a graphic object property (which should generally not happen), you can use the pseudo-class construct to restrict a rule to a specified renderer. As an example, suppose you have written a custom graphic object which also has a property called `Fixed`. The following code example shows a style rule which specifies that the property is to be set only in the renderer (not in the custom graphic object).

Setting a renderer property for a graphic object

```
node.participant:renderer {
   Fixed : "true";
}
```

The pseudo-class `renderer` is interpreted as the renderer which has the property called `Fixed`.

The pseudo-class specification can also be the name of the renderer with an initial lowercase letter, for example, `graphLayoutRenderer`.

Predefined renderers

Describes the predefined renderers provided in IBM® ILOG® JViews Diagrammer. The following predefined SDM renderers are described, with their global properties and their rendering properties if any

In this section

Class summary
Presents an overview of the predefined renderers and their class relationships.

The Coloring renderer
Describes the `Coloring` renderer.

The Decoration renderer
Describes the `Decoration` renderer.

The Blinking renderer
Describes the `Blinking` renderer.

The GrapherPin renderer
Describes the `GrapherPin` renderer.

The GraphLayout renderer
Describes the `GraphLayout` renderer.

The DrillDown renderer
Describes the `DrillDown` renderer.

The HalfZooming renderer
Describes the `HalfZooming` renderer.

The InfoBalloon renderer
Describes the `InfoBalloon` renderer.

The Composite renderer
Describes the `Composite` renderer.

The Interactor renderer
Describes the `Interactor` renderer.

The LabelLayout renderer
Describes the `LabelLayout` renderer.

The Legend renderer
Describes the `Legend` renderer.

The LinkLayout renderer
Describes the `LinkLayout` renderer.

The Map renderer
Describes the `Map` renderer.

The StyleSheet renderer
Describes the `StyleSheet` renderer.

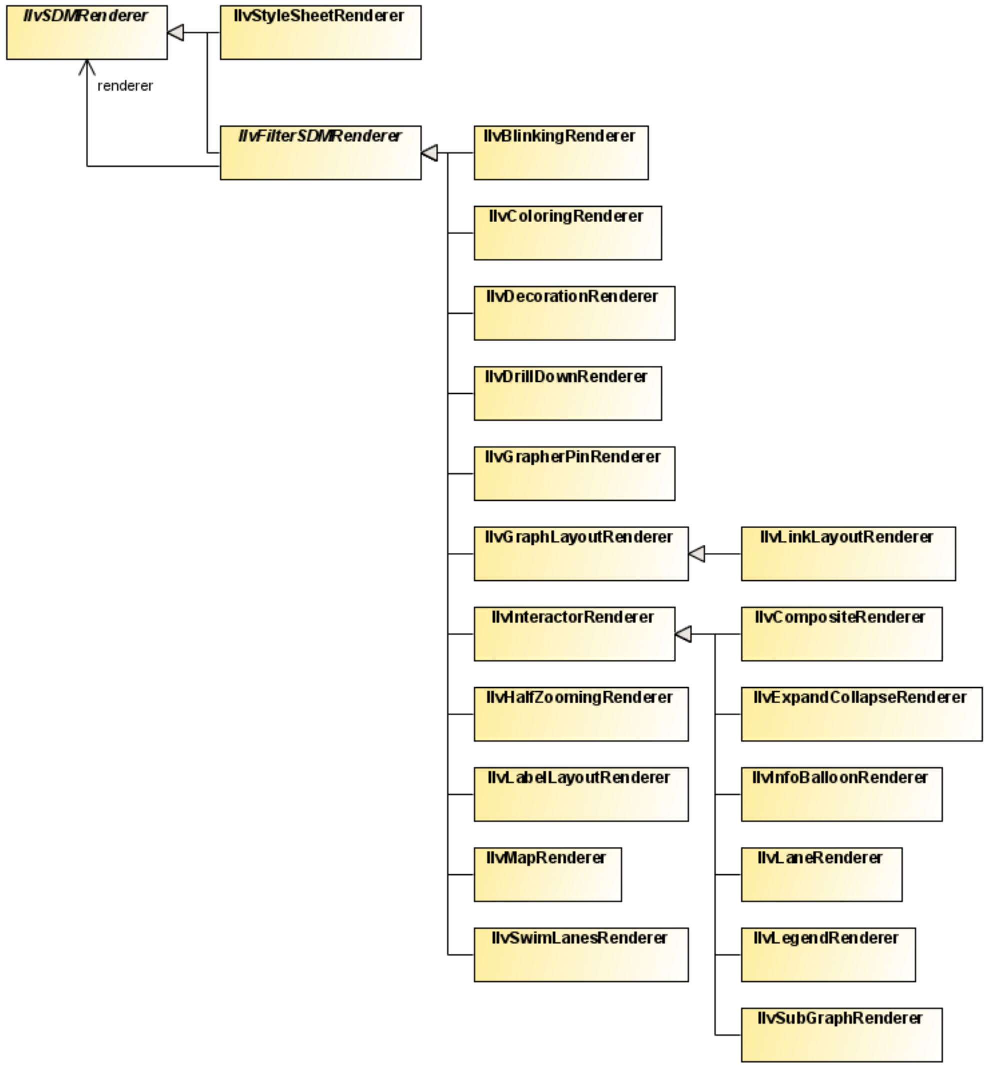
The Map StyleSheet renderer
Describes the Map StyleSheet renderer.

The SubGraph renderer
Describes the `SubGraph` renderer.

The SwimLanes renderer
Describes the `SwimLanes` renderer.

Class summary

This section describes the following predefined SDM renderers with their global properties and their rendering properties if any.



Predefined renderers class relationships

The Coloring renderer

The `Coloring` renderer allocates colors automatically to nodes or links or both, depending on one of their properties. This is useful when you want to represent objects with different colors depending on a given property, but you do not know in advance how many different values the property will take. There are two ways to set colors.

With the first way, you use the custom function called `coloring` of the `Coloring` renderer which returns the color mapped to the model property value. The following code example shows a style rule that sets the foreground color according to the level of the node.

Using a custom function to set coloring in the style sheet

```
node[level] {
   foreground : @|coloring(level);
}
```

This rule states that all nodes that define the `level` property will be visualized with the `foreground` property set to a color computed according to the `level` value. Objects with the same `level` value will have the same color.

With the second way, you use the two properties `colorProperty` and `indexProperty` of the `Coloring` renderer. These properties can be set to achieve a similar, although more global, behavior, as shown in the following code example.

Using the Coloring renderer to set global coloring

```
Coloring {
   colorProperty : foreground;
   indexProperty : level;
}
```

This rule sets the `foreground` property for all nodes that have the `level` property.

The first way is preferred since the object selectors can be more specific than the coloring renderer global setting.

The following table lists the properties of the `Coloring` renderer:

Global properties of the Coloring renderer

Property	Type	Default	Description
alpha	float	1.0	The alpha value (that is, the transparency) of the allocated colors.
brightness	float	1.0	The brightness of the allocated colors.
colorProperty	String	“foreground”	The color property to set on the graphic objects.
hue	float	0.0	The hue with which the color allocation algorithm will start.
indexProperty	String	`null`	The object property used as an index to allocate colors.
saturation	float	1.0	The saturation of the allocated colors.

The `Coloring` renderer does not have any per-object rendering properties.

See the class `IlvColoringRenderer` for more details.

The Decoration renderer

The `Decoration` renderer lets you add graphic objects in addition to the objects that represent the nodes and links, and change the background color or texture. You can use the `Decoration` renderer to display additional information such as titles or annotation labels. The objects you add are static: they cannot change in response to model object states.

You can load decoration objects from a `.ivl` file or define them in the style sheet, as follows:

1. Give a name to each decoration.
2. Attach the `decorationNames` property of the `Decoration` renderer to all the names collected at step 1.
3. Add a style rule for each decoration, where the selector is an ID with the decoration name, and the declarations are what are needed to create and customize the decoration.

The following code example shows style rules that add a title and an image to the diagram.

Adding a title and image to the diagram with the Decoration renderer

```
SDM {
Decoration : true ;
}

Decoration {
decorationNames : 'title,logo' ;
}

#title {
  class      : "ilog.views.sdm.graphic.IlvGraphicFactories$ShadowLabel";
  IlvRect    :   '5,5,100,20';
  font       : "dialog-18-bold";
  foreground : black;
  background : white ;
  label      : 'The main window.';
}

#logo {
  class         : "ilog.views.sdm.graphic.IlvGraphicFactories$Icon";
  IlvRect       : '5,35,20,20';
  imageLocation : url(images/company.gif);
}
```

The following table lists the properties of the `Decoration` renderer.

Global properties of the Decoration renderer

Property	Type	Default	Description
background	Color	white	The view background color, as defined in `IlvManagerView`.
backgroundFile	URL		Sets a URL that points to a `.ivl`, `.svg`, `.svgz`, or `.dxf` file. The file contains graphic objects that are added to the diagram.
backgroundPatternLocation	URL		The view background pattern, as defined in `IlvManagerView`.
decorationNames	String[]	`null`	The names of the style rules that define the decoration objects.

The `Decoration` renderer does not define any per-object rendering properties.

See the class `IlvDecorationRenderer` for more details.

The Blinking renderer

The `Blinking` renderer offers a convenient API to obtain color or visibility blinking on graphic objects. It allows you to specify a uniform timing for all graphic objects with visibility blinking:

```
SDM {
    Blinking : "true";
}
Blinking {
    onPeriod : 1000;   // Every second
    offPeriod : 1000;  // Every second
}
```

The blinking can be defined in CSS in different ways:

♦ Toggling between colors

  The classes `IlvBlinkingColor` and `IlvBlinkingMultiColor` represent blinking colors. They can be used as Color for various properties of graphic objects that support blinking.

  The following example illustrates how to specify a blinking color in CSS:

```
@|blinkinColor(color1,color2)
@|blinkinColor(color1,color2,color1Timing,color2Timing)
Bcolor1Timing/color2Timimg[color1/color2]
Btiming[color1/color2/color3/...]
```

  The first two lines work only when the `Blinking` renderer is installed. The first line creates a blinking color with the timings for the color taken from the `Blinking` renderer. The second and third lines are essentially equivalent. The fourth line creates a blinking color with multiple subcolors. Note that the `Blinking` renderer only enables the appropriate syntax, but it is not necessary for the blinking effect. The third and the fourth lines work even when no `Blinking` renderer is installed.

  Here is a concrete example:

  Color blinking

```
node {
     class: "ilog.views.sdm.graphic.IlvGeneralNode";
     label: "@name";
     // toggles from red to black (2 states)
     fillColor1: "B1000/1000[red/black]";
     // toggles from red to yellow (2 states)
     fillColor2: "@|blinkingColor(red,yellow)";
     // toggles from black to transparent red to yellow (3 states)
     labelColor: "B500[black/#aaff0000/yellow]";
 }
```

♦ Toggling between paints

  The classes `IlvBlinkingPaint` and `IlvBlinkingMultiPaint` represent blinking paints. They can be used as Paint for various properties of graphic objects that support blinking. Blinking paints work even when no `Blinking` renderer is installed.

  The following example illustrates how to specify a blinking paint in CSS:

Paint blinking

```
node {
    class: "ilog.views.graphic.IlvGeneralPath";
    fillPaint: "@=fillPaint";
    fillOn: "true";
}
Subobject#fillPaint {
    class: "ilog.views.util.java2d.IlvBlinkingPaint
(onPaint,offPaint,onPeriod,offPeriod)";
    onPaint: "@#onPaint";
    offPaint: "@#offPaint";
    onPeriod: "1000";
    offPeriod: "1000";
}
Subobject#onPaint {
    class: "java.awt.GradientPaint(point1,color1,point2,color2)";
    point1: "0.0,0.0";
    color1: "white";
    point2: "72.0,72.0";
    color2: "255,0,51";
}
Subobject#offPaint {
    class: "java.awt.GradientPaint(point1,color1,point2,color2)";
    point1: "0.0,0.0";
    color1: "255,0,51";
    point2: "72.0,72.0";
    color2: "white";
}
```

- Toggling the visibility

  The renderer property `ToggleVisibility` switches the visibility of the whole object periodically. The blinking timings are taken from the `Blinking` renderer, therefore this specification works only when the `Blinking` renderer is installed and enabled.

  Visibility blinking

  ```
  node {
      class: "ilog.views.sdm.graphic.IlvGeneralNode";
      ToggleVisibility: "true";
  }
  ```

  When you use this specification, the visibility is toggled with the same timings for all the objects. You can also specify individual timings per node; this does not require the `Blinking` renderer to be installed. It is illustrated in the following example:

  ```
  node {
      class: "ilog.views.sdm.graphic.IlvGeneralNode";
      blinkingOnPeriod: 1000;
      blinkingOffPeriod: 1000;
  }
  ```

The following table lists the global properties of the `Blinking` renderer.

Global properties of the Blinking renderer

Property	Type	Default	Description
onPeriod	int	1000	The duration, in milliseconds, of the first color specified in the `blinkingColor` function, or of the visible period when `ToggleVisibility` is set to `true`.
offPeriod	int	1000	The duration, in milliseconds, of the second color specified in the `blinkingColor` function, or of the invisible period when `ToggleVisibility` is set to `true`.

The following table lists the per-object properties of the `Blinking` renderer.

Per-object properties of the Blinking renderer

Property	Type	Default	Description
ToggleVisibility	boolean	false	If the value of this property is `true`, the graphic representation will toggle the visibility.

See the class `IlvBlinkingRenderer` for more details.

The GrapherPin renderer

The `GrapherPin` renderer defines the exact positions where links will be connected to the nodes of the graph. For example, the following code example shows a rule which declares that a node of tag `split` has three connection points: one at the top of the node's bounding rectangle, one on the left, and another one on the right.

Specifying link connection points in the style sheet

```
node.split {
  GrapherPin[0] : "Top";
  GrapherPin[1] : "Left";
  GrapherPin[2] : "Right";
}
```

Other rules can then make use of the connection points defined to specify where different types of links are to be connected. For example, the following code example shows a rule that connects all links of tag `transition` whose `condition` property is `true` from the right of the source node to the top of the destination node.

Specifying how the ends of a transition link are connected

```
link.transition[condition="true"] {
  FromPin : "2";    // pin #2 = Right (see node.split rule)
  ToPin : "0";      // pin #0 = Top (see node.split rule)
}
```

The grapher pins are taken into account during editing when you create a new link or when you reconnect an existing link. They are displayed as small circles during the interaction.

The `GrapherPin` renderer does not have any global properties.

The following table lists the per-object rendering properties of the `GrapherPin` renderer.

Per-object properties of the GrapherPin renderer

Property	Type	Default	Description
FromPin	int		The grapher pin to which the link will be connected on the source node. The value of the property is the index *n* of the `GrapherPin[` *n* `]` property of the node, and the `GrapherPin[` *n* `]` entry specifies the connection point.
GrapherPin[*n*]	position, or x,y		The grapher pin of index *n* for this node. The value of the property can be either a position relative to the node's bounding box (for example: `Top`, `BottomLeft`) or an *x*, *y* couple,

Property	Type	Default	Description
			indicating the relative X and Y offsets of the grapher pin (0 = left or top, 1 = bottom or right, 0.5 = center).
ToPin	int		The grapher pin to which the link will be connected on the destination node. The value of the property is the index *n* of the `GrapherPin[` *n* `]` property of the node.

For more details, see the class `IlvGrapherPinRenderer`.

The GraphLayout renderer

The `GraphLayout` renderer lets you apply an automatic layout algorithm to the nodes of the graph. The layout algorithm can be any of the algorithms made available by the Graph Layout capability.

The graph layout algorithm will be applied once when the data model is loaded, and also every time a new object is dynamically added to the model. The algorithm can be temporarily disabled and then re-enabled.

The following table lists the properties of the `GraphLayout` renderer.

Global properties of the GraphLayout renderer

Property	Type	Default	Description
connectingLinksToShape	boolean	`false`	Specifies whether the links should be connected to the shape of the node or to the bounding box of the node (including the label). Connection to the shape is possible if the node is an `IlvGeneralNode` object, and does not take the label into account if it is outside the shape.
enabled	boolean	`true`	Enables/disables the graph layout algorithm.
graphLayout	IlvGraphLayout	`null`	Specifies which algorithm to apply. You can use the full class name, for example: `IlvHierarchicalLayout` or an abbreviated name (for example, `Hierarchical`).
incrementalLayout	boolean	`false`	Tells the renderer to perform an incremental layout (if the selected algorithm is a hierarchical layout). If this property is `true`, all the nodes that are not selected will be marked for incremental layout. As a result, the layout will try to keep these nodes at their current position in the hierarchy, and place the selected nodes regardless of their current position. Note that this is related to a partial layout, but is not exactly the same. In particular, nodes that are not selected may still be moved slightly after an incremental layout. See

Property	Type	Default	Description
			`IlvHierarchicalLayout` for a complete description of incremental layout.
partialLayout	boolean	`false`	Tells the renderer to lay out only the selected objects. Objects that are not selected are marked as fixed before the layout.
savingNodePositions	boolean	`false`	Specifies whether the new positions of the nodes must be saved to the data model after the layout is applied.

Setting properties of IlvGraphLayout objects

You can set any property of the `IlvGraphLayout` subclass corresponding to the selected layout algorithm. The following code example shows an example of some typical rules.

Typical graph layout rules

```
SDM {
    GraphLayout : "Hierarchical";
}

GraphLayout {
    enabled         : "false";
    globalLinkStyle : "ORTHOGONAL_LINKS";
    flowDirection   : "Bottom";
}
```

In this example, `enabled` is a property of the `GraphLayout` renderer itself, but `globalLinkStyle` and `flowDirection` are properties of the `IlvHierarchicalLayout` object used by the renderer.

The properties of each layout algorithm are fully explained in the *Using Graph Layout Algorithms*. The properties of the `IlvGraphLayout` subclasses conform to the JavaBeans™ conventions: if a class has a pair of methods called `setMyProp` (with a single parameter) and `getMyProp` (without parameters), then you can set the property `myProp` in the style sheet.

Note: If the value of the property is an enumeration of integer values defined by static member variables of the class, you can use the name of the variable alone, or the variable name prefixed by the class name alone, or the variable name prefixed by the fully qualified class name. For example, these declarations are all valid:

globalLinkStyle : "ORTHOGONAL_LINKS";

globalLinkStyle : "ORTHOGONAL_LINKS";

globalLinkStyle : "IlvHierarchicalLayout.ORTHOGONAL_LINKS";

```
globalLinkStyle :
"ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout.ORTHOGONAL_LINKS";
```

The following table lists the per-object rendering properties of the `GraphLayout` renderer:

Per-object properties of the GraphLayout renderer

Property	Type	Default	Description
GraphLayout	IlvGraphLayout	`null`	Lets you define a different graph layout algorithm for each subgraph. If one is not specified, the same algorithm is applied recursively to all subgraphs.
LayoutCluster	String	n/a	Deprecated, replaced by ClusterId
LayoutFixed	String	`false`	Deprecated, replaced by Fixed.
LayoutGroup	String	`null`	Lets you apply the algorithm to different groups of objects, one group after the other.
LayoutIgnored	boolean	`false`	If `true`, the object is ignored by the layout.
LayoutStarCenter	boolean	`false`	Deprecated, replaced by StarCenter.
ClusterId	String	n/a	For Circular Layout only: A list of cluster identifiers (at least one) to which a node belongs. For example, the ClusterId value "`cluster3`" means that the node belongs to the cluster with this identifier; the ClusterId value "`cluster3,2`" means that the node belongs to the cluster `cluster3` and has the order index "2"; the ClusterId value "`cluster3,2;cluster5,3`" means that the node belongs to two clusters, `cluster3` and `cluster5`, and has the order index "2" on the first cluster and the order index "3" on the second cluster. It is mandatory to specify a value for the node to be managed by the circular layout.
StarCenter	boolean	`false`	For Circular Layout only: If `true`, the object is placed at the center of a star configuration.
Fixed	boolean	`false`	If `true`, the node is not moved by the graph layout (for links, the link is not reshaped).

Setting node or link parameters on graph layout objects

You can set any parameter of a graph layout algorithm that applies to a particular node or link in the style sheet. Such a parameter is defined by a method of the form:

```
setMyParam(Object node, <type> value);
```

or

```
setMyParam(Object link, <type> value);
```

Node or link parameters are set in the style sheet as follows:

```
node {
    MyParam : "value";
}
```

The name of the property is the name of the method, without the `set` prefix. Note that node and link parameters are capitalized, unlike Bean properties.

For example, the hierarchical layout defines a `setFromPortSide` method that lets you choose which side of the origin node a given link will be connected to. The signature of the method is:

```
    setFromPortSide(Object link, int side);
```

In the style sheet, you can set this parameter as follows:

```
link {
    FromPortSide : “WEST”;
}
```

The value of the property can be any simple type (integer, String, float), or it can be the name of a public constant defined by the graph layout class, for example, `WEST` , which is defined in the class `IlvHierarchicalLayout`.

The DrillDown renderer

The DrillDown renderer allows you to display more and more objects as you zoom in the view. The objects are dispatched to different manager layers with visibility filters, which are configured according to the object property DrillDownZoom. All the objects with the same drill-down zoom are placed in a manager layer that is visible only when the view's zoom level is greater than the drill-down zoom.

The following code extract illustrates how to enable the DrillDown renderer:

```
SDM {
  DrillDown: "true";
}
```

The following table lists the global properties of the DrillDown renderer.

Global properties of the DrillDown renderer

Property	Type	Default	Description
showingIndicator	boolean	false	Specifies whether an icon should be displayed in the top-left corner of the view to alert the user that some objects are not visible at the current zoom level. The indicator disappears when the user zooms in enough to make all objects visible.

The following code extract shows how to configure the DrillDown renderer global property:

```
DrillDown {
  showingIndicator: "true";
}
```

The following table lists the per-object properties of the DrillDown renderer.

Per-object properties of the DrillDown renderer

Property	Type	Default	Description
DrillDownZoom	double	0	Specifies the zoom level above which the objects become visible.

The following code extract illustrates how to specify the drill down zoom level for objects based on data model properties. In this scenario, links become visible according to the data model property distance:

Specifying drill down zoom levels

```
link[distance>="1000"] {
  DrillDownZoom : "0";
}
link[distance<"1000"] {
  DrillDownZoom : "2";
}
link[distance<"100"] {
  DrillDownZoom : "4";
}
link[distance<"10"] {
  DrillDownZoom : "8";
}
```

See the class `IlvDrillDownRenderer` for more details.

The HalfZooming renderer

The `HalfZooming` renderer lets you control the way your graphic objects will grow or shrink when you zoom in or out. You can specify the following parameters:

- ♦ Maximum zoom level above which the objects will stop growing; their size will stay fixed past this zoom level.
- ♦ Minimum zoom level below which the objects will not be drawn any more; they will be replaced by gray boxes, or will keep a fixed size.

You can also specify an initial zoom factor by which the minimum and maximum zoom levels will be multiplied.

The following table lists the properties of the `HalfZooming` renderer.

Global properties of the HalfZooming renderer

Property	Type	Default	Description
alpha	double	1.0	Specifies the alpha globally for all objects.
grayedWhenUnzoomed	boolean	`true`	If `true`, the nodes are drawn as gray boxes when the zoom level is below the minimum zoom, and the gray boxes zoom out normally. If `false`, the nodes simply stop zooming out and keep the same size.
initialZoom	double	1.0	Specifies the initial zoom globally for all objects.
maxZoom	double	1.0	Specifies the maximum zoom globally for all objects.
minZoom	double	1.0	Specifies the minimum zoom globally for all objects.
unzoomedBackground	Color	grey	The background (the fill color) of the boxes displayed instead of zoomed out objects.
unzoomedForeground	Color	black	The foreground (the border color) of the boxes displayed instead of zoomed out objects.

The following table lists the per-object rendering properties of the `HalfZooming` renderer.

Per-object properties of the HalfZooming renderer

Property	Type	Default	Description
Alpha	double		Overrides `alpha` for a given object.
InitialZoom	double		Overrides `initialZoom` for a given object.
MaxZoom	double		Overrides `maxZoom` for a given object.
MinZoom	double		Overrides `minZoom` for a given object.
UnzoomedBackground	Color		Overrides `unzoomedBackground` for a given object.
UnzoomedForeground	Color		Overrides `unzoomedForeground` for a given object.

See the class `IlvHalfZoomingRenderer` for more details.

The InfoBalloon renderer

The `InfoBalloon` renderer displays information about a node or a link when the user clicks in it. The information is displayed in a customizable popup window called a balloon.

The contents of the balloon are specified through strings that have the following format:

```
"line1,First Title: ,prop1;line2,Second Title: ,prop2;..."
```

where `prop1`, `prop2`, and so on are the names of the object's properties for which values are to be displayed.

The following table lists the properties of the `InfoBalloon` renderer.

Global properties of the InfoBalloon renderer

Property	Type	Default	Description
linkBalloonLines	String		Specifies the contents of the balloon globally for all the links.
nodeBalloonLines	String		Specifies the contents of the balloon globally for all the nodes.
prototype	String		Sets the prototype(s) that represents the balloon.

The following table lists the per-object rendering properties of the `InfoBalloon` renderer.

Per-object properties of the InfoBalloon renderer

Property	Type	Default	Description
InfoBalloonLines	String		Specifies the contents of the balloon for a particular object. Overrides `nodeBalloonLines` or `linkBalloonLines`.

See the class `IlvInfoBalloonRenderer` for more details.

The Composite renderer

The `Composite` renderer provides support for symbols and composite graphics to represent nodes and links. In particular, it allows you to:

- ♦ Perform Swing actions on predefined gestures defined for any child of a composite graphic.
- ♦ Define which child the links connect to.

The following code extract illustrates how to enable the `Composite` renderer:

```
SDM {
  Composite: "true";
}
```

The following table lists the per-object properties of the `Composite` renderer.

Per-object properties of the Composite renderer

Property	Type	Default	Description
LinkConnectionRectangle	int	-1	Specifies the index used with the indexed property `children` of composite graphics. The links connect to the nearest edge of the bounding box of the child at the position defined by this property. This property is defined on the objects of type node.
FromCompositePin	String	null	Specifies the name of a decoration in a composite graphic, which is used to connect the origin of a link. This property is defined on the objects of type link.
FromCompositePinPosition	String	Center	Connects the origin to this position of the decoration. This property is defined on the objects of type link.
ToCompositePin	String	null	Specifies the name of a decoration in a composite graphic, which is used to connect the destination of a link. This property is defined on the objects of type link.
ToCompositePinPosition	String	Center	Connects the destination to this position of the decoration. This property is defined on the objects of type link.

The following code extract illustrates how to configure the link connection rectangle:

```
node {
  LinkConnectionRectangle: 0;
}
```

The following code extract illustrates how to specify the decoration to be used to connect links:

```
link {
  FromCompositePin:"decoration";       // use the name defined above
  FromCompositePinPosition:"Center";
  ToCompositePin:"decoration";         // use the name defined above
  ToCompositePinPosition:"Center";
}
```

See the class `IlvCompositeRenderer` for more details.

The Interactor renderer

The `Interactor` renderer is a base class for all renderers that attach object interactors to graphic objects. This renderer can also be used directly to install an object interactor specified in the style sheet.

The following table lists the properties of the `Interactor` renderer.

Global properties of the Interactor renderer

Property	Type	Default	Description
interactor	String	`null`	Specifies the class name of the object interactor to attach to all the graphic objects.

The following table lists the per-object rendering properties of the `Interactor` renderer.

Per-object properties of the Interactor renderer

Property	Type	Default	Description
Interactor	String	`null`	Specifies the class name of the object interactor to attach to a particular graphic object. Overrides the `interactor` property.
processMouseMoveEvent	boolean	false	Specifies whether mouse motion events are dispatched in addition to mouse click events.

See the class `IlvInteractorRenderer` for more details.

The LabelLayout renderer

The `LabelLayout` renderer performs a label layout algorithm. It is useful when the graphic objects that represent the data model have labels and you want to minimize the overlap of these labels with other labels or other graphic objects. The algorithm will try to move the labels around their original position (for node labels) or along their link (for link labels) to minimize the overlaps.

The `LabelLayout` renderer uses the classes of the package `ilog.views.graphlayout.labellayout`, available since IBM® ILOG® JViews 5.0. The label layout algorithm applied is the "simulated annealing" algorithm.

The following table lists the properties of the `LabelLayout` renderer.

Global properties of the LabelLayout renderer

Property	Type	Default	Description
enabled	boolean	`true`	Enables/disables the label layout algorithm.
layoutOfGeneralLinkLabelsEnabled	boolean	`true`	If `true`, the labels of `IlvGeneralLink` objects will be laid out. If `false`, general link labels are left unchanged.
layoutOfGeneralNodeLabelsEnabled	boolean	`true`	If `true`, the labels of `IlvGeneralNode` objects will be laid out. If `false`, general node labels are left unchanged.

The `LabelLayout` renderer does not have any per-object rendering properties.

You can set any property of the `IlvAnnealingLabelLayout` object used by this renderer. For details see *Setting properties of IlvGraphLayout objects*.

See the class `IlvLabelLayoutRenderer` for more details.

The Legend renderer

The `Legend` renderer displays a legend box explaining the look of the graphic objects. The legend is generated using an index property. For each object, the value of the index property is queried. If no object with the same value is already shown in the legend, a new entry is added.

Each entry consists of a copy of the graphic object and a label displaying the value of the property for this graphic object.

The legend can be contained in the same view as the generated graph, or it can be in a separate frame.

The legend can contain several columns, each corresponding to a different index property.

The graphic objects in the legend can be customized separately, using the pseudoclass `legend`.

For example, the following code example shows how to set the same color for all the links in the legend.

```
link:legend {
   foreground : "green";
}
```

The following table lists the properties of the `Legend` renderer.

Global properties of the Legend renderer

Property	Type	Default	Description
background	Color	white	The background color of the legend box.
checkBoxesVisible	boolean	`true`	If `true`, each entry has a check box that lets the user show or hide the corresponding graphic objects.
foreground	Color	black	The border color of the legend box.
indexProperty	String	`null`	Specifies the object property to use as an index. You can specify several properties, separated by

Property	Type	Default	Description
			commas. Each index property will be displayed in a separate column of the legend.
inlaid	boolean	`true`	If `true`, the legend box is contained in the diagram. Otherwise, it is displayed in a separate frame.
insideMargin	int	5	The margin around the legend elements.
labelFont	Font		The font used for the labels of the entries.
labelsZoomable	boolean	`false`	Specifies whether the labels must be zoomable.
layer	int	30	The manager layer in which the legend box will be added.
legendVisible	boolean	`true`	Hides or shows the legend.
legendZooming	boolean	`true` if inlaid, `false` if not inlaid	If `true`, the legend items zoom together with the view. If `false`, the legend items do not zoom; they

Property	Type	Default	Description
			always keep the same size when the diagram is zoomed in or out.
linksLength	float[]	50	The length of links in each column of the legend.
linksWidth	float[]	0	The width of links in each column of the legend. Zero means the same width as in the diagram.
maxEntries	int	10	The maximum number of entries for each column.
outsideMargin	int	5	The margin around the legend.
position	int	BottomLeft	The position where the legend will be placed in the diagram.
scrollDelay	int	200	The delay (in milliseconds) before showing the legend after scrolling the diagram.
sortingEntries	boolean	`true`	If `true`, the legend entries are sorted alphabetically. If `false`, the entries appear in the same order as in the data model.
spacing	int	5	The spacing between the elements of the legend.
title	String	`null`	Specifies the title(s) of the column(s). You can pass several values, separated by commas.
titleFont	Font		The font used for the titles of the columns.
updateOnPropertyChange	boolean	`false`	If `true`, the legend is dynamically updated when a property of an object changes in the data model.
xOffset	float	0	The horizontal offset of the legend relative to the border of the diagram.
yOffset	float	0	The vertical offset of the legend relative to the border of the diagram.

The `Legend` renderer does not define any per-object rendering properties.

See the class `IlvLegendRenderer` for more details.

The LinkLayout renderer

The `LinkLayout` renderer is a subclass of the `GraphLayout` renderer which is specialized to apply a link layout algorithm.

The following table lists the properties of the `LinkLayout` renderer (in addition to the properties of the `GraphLayout` renderer).

Global properties of the LinkLayout renderer

Property	**Type**	**Default**	**Description**
addingLinkConnectors	boolean	`true`	If `false`, no link connector is installed by the link layout algorithm.
hierarchical	boolean	`false`	If `true`, and if a GraphLayout renderer is present and configured to use a hierarchical layout algorithm, then the link layout will be performed by the hierarchical layout.
performingLayoutOnZoom	boolean	`false`	If `true`, the link layout is reapplied when the view is zoomed in or out.

The following table lists the per-object rendering properties of the `LinkLayout` renderer.

Per-object properties of the LinkLayout renderer

Property	**Type**	**Default**	**Description**
NodeSideForDestination	int	0	If this property is specified, it tells the link layout algorithm to try to connect links to the specified side of their destination node. The possible values are defined by the class `IlvDirection`.
NodeSideForOrigin	int	0	If this property is specified, it tells the link layout algorithm to try to connect links to the specified side of their origin node. The possible values are defined by the class `IlvDirection`.

You can set any property of the `IlvLinkLayout` object used by this renderer. For details, see *Setting properties of IlvGraphLayout objects*

See also *Setting node or link parameters on graph layout objects*.

See the class `IlvLinkLayoutRenderer` for more details.

The Map renderer

The `Map` renderer displays a background map behind the graph and automatically places geo-referenced objects on top of this map. You need to have the product JViews Maps installed with a valid license to be able to use this renderer.

The map is specified as the URL of an IVL file that will be loaded into the diagram. This IVL file can contain any cartographic information supported by IBM® ILOG® JViews Maps, such as vectorial or raster maps read from various cartographic formats, load-on-demand readers, and so on. The IVL file should also contain the definition of a projection. This projection information will be used to place the objects on the map according to their latitude and longitude.

The following table lists the properties of the `Map` renderer.

Global properties of the Map renderer

Property	Type	Default	Description
autoRegionOfInterest	boolean	`false`	If `true`, the region of interest is computed automatically from the positions of the nodes contained in the data model.
map	String	`null`	The URL of the IVL file containing the map and the projection.
regionOfInterest	IlvRect	`null`	The "region of interest" on the map. This is useful to center on or zoom to a particular region of a larger map. The rectangle is in latitude and longitude coordinates (radians).
regionOfInterestMargin	float	5	The margin remaining around the objects when the region of interest is computed automatically. The margin is expressed as a percentage of the width of the bounding box of the objects.

The following table lists the per-object rendering properties of the `Map` renderer.

Per-object properties of the Map renderer

Property	Type	Default	Description
latitude	double		The latitude of the object.
longitude	double		The longitude of the object.
fitToContentsAllowed	boolean	`true`	Performs a fit-to-contents each time the diagram is reloaded.

The latitude and longitude can be specified in several formats:

♦ As a floating-point value, in which case the value is the latitude or longitude expressed in *radians*.

- In degrees/minutes/seconds, as a string of the form *xx* `D` *xx* `'` *xx* `"` *L*, where *L* is one of the following: `N` (north), `S` (south), `W` (west) or `E` (east). The string will be converted to radians.
- As a string matching one of the formats registered in the Maps package.

Note: The latitude and longitude of the objects are usually stored in the data model. If the style sheet does not specify any values for the latitude and longitude properties, these properties will be read directly from the data model. Most of the time, the style sheet will be used only to translate the names of the data object properties holding the latitude and longitude, if needed. For example, if your data model has properties named LAT and LONG, you will write a rule like this in the style sheet:

```
node {
   latitude  : "@LAT";
   longitude : "@LONG";
}
```

See the class `IlvMapRenderer` for more details.

The StyleSheet renderer

The `StyleSheet` renderer is the default renderer. The `StyleSheet` renderer is always enabled.

The `StyleSheet` renderer implements the basic creation of graphic objects. For each object in the data model, the `StyleSheet` renderer looks up all the style rules that match the object and creates the graphic object specified by the declarations of these rules.

The following table lists the properties of the `StyleSheet` renderer.

Global properties of the StyleSheet renderer

Property	Type	Default	Description
debugMask	int	0	Prints selected debug information.
styleSheets	String[]	`null`	Sets cascading style sheets. This property is usually set on the SDM engine rather than directly on the `StyleSheet` renderer.
linkConnectorEnabled	boolean	`true`	Specifies if a link connector should be installed on the nodes to connect the links to the sides of the nodes, rather than to their centers.
addingLinkConnectors	boolean	`true`	If `false`, no link connector is installed by the style sheet renderer.
nodesLayer	int	10	Specifies the default layer in which the nodes will be added.
linksLayer	int	9	Specifies the default layer in which the links will be added.

The following table lists the per-object rendering properties of the `StyleSheet` renderer.

Per-object properties of the StyleSheet renderer

Property	Type	Default	Description
Anchor	int	Center	The position of the object's location relative to its bounding box (for example, TopLeft will place the object so that its upper-left corner is located at the x, y position).
Layer	int		Overrides nodesLayer or linksLayer for a specific object.
LinkConnector	IlvLinkConnector		Defines the link connector set on the nodes of the graph. By default, an `IlvSDMLinkConnector` object is used.
ToolTipText	String	`null`	Deprecated, replaced by the `IlvGraphic` property `toolTipText`.
x	float		The horizontal coordinate of the object.
y	float		The vertical coordinate of the object.
width	int		The width of the object. If the width is not specified in the style sheet file, and if it is specified in the data model XML file, the value contained in the XML file will be used. See `IlvSDMRenderer.getGraphicProperty` for details.
height	int		The height of the object. If the height is not specified in the style sheet file, and if it is specified in the data model XML file, the value contained in the XML file will be used. See `IlvSDMRenderer.getGraphicProperty` for details.

See the class `IlvStyleSheetRenderer` for more details.

The `StyleSheet` renderer is always present, but you must explicitly declare changes in the style sheet if some properties are to be modified. shows a style rule which instructs the `StyleSheet` renderer to print debug information about all the declarations being processed.

Debugging the style sheet

```
StyleSheet {
   debugMask : "DECL_MASK|DECL_VALUE_MASK" ;
}
```

The Map StyleSheet renderer

The Map StyleSheet renderer is a specific version of *The StyleSheet renderer*. You need to have IBM® ILOG® JViews Maps installed with a valid license to be able to use this renderer.

The Map StyleSheet renderer wraps the node and link layers into an `IlvMapLayer` object. This is done in order for them to be managed by the maps layer tree. For example, this allows the following:

♦ Interactive raising and lowering of layers.

♦ Indication that the layers are always on top of the map background.

In a JViews Maps application you have to enable the Map StyleSheet renderer using Java™ code before you make any use of a style sheet. The following code example shows how to replace the StyleSheet renderer:

```
// replace the stylesheet renderer with map rendererclass.
IlvRendererUtil.addRendererAlias(
IlvRendererUtil.getRendererAlias(IlvStyleSheetRenderer.class.getName()),
IlvMapStyleSheetRenderer.class.getName());
```

The Map StyleSheet renderer provides exactly the same properties as the StyleSheet Renderer. However, any use of the `nodesLayer`, `linksLayer` or `Layer` properties will be overridden by the Map Layer tree order. The following table lists the Map StyleSheet renderer properties.

The Map StyleSheet renderer global properties

Property	Type	Default	Description
debugMask	int	0	Prints selected debug information.
styleSheets	String[]	null	Sets cascading style sheets. This property is usually set on the SDM engine rather than directly on the StyleSheet renderer.
linkConnectorEnabled	boolean	true	Specifies if a link connector should be installed to connect the links to the sides of the nodes, rather than to their centers.
addingLinkConnectors	boolean	true	If `false`, no link connector is installed by the style sheet renderer.

The following table lists the Map StyleSheet renderer per-object rendering properties.

The Map StyleSheet renderer per-object proprieties

Property	Type	Default	Description
Anchor	int	Center	The position of the object's location relative to its bounding box (for example, Top Left? will place the object so that its upper-left corner is located at the x, y position).
LinkConnector	IlvLinkConnector		Defines the link connector set on the nodes of the graph. By default, an `IlvSDMLinkConnector` object is used.
ToolTipText	String	null	Deprecated, replaced by the `IlvGraphic` property, `toolTipText`.
x	float		The horizontal coordinate of the object.
y	float		The vertical coordinate of the object.
width	int		The width of the object. If the width is not specified in the style sheet file, and if it is specified in the data model XML file, the value contained in the XML file will be used. See `IlvSDMRenderer` for details.
height	int		The height of the object. If the height is not specified in the style sheet file, and if it is specified in the data model XML file, the value contained in the XML file will be used. See `getGraphicProperty` for details.

The SubGraph renderer

The `SubGraph` renderer displays subgraphs of the data model as expandable and collapsible nodes. This renderer is useful if the data model is hierarchical, that is, if it contains nodes that have subnodes and sublinks.

An expandable node can be in one of two states: expanded or collapsed. When it is expanded, its subobjects are visible, and a frame is displayed around them. When the expandable node is collapsed, its subobjects are hidden, and the collapsed node looks like a simple node. You can customize the look of the frame and of the collapsed node in the style sheet.

An expandable node can be collapsed and re-expanded interactively by clicking in the icon displayed in its upper-left corner.

The following table lists the properties of the `SubGraph` renderer.

Global properties of the SubGraph renderer

Property	Type	Default	Description
loadOnDemand	boolean	`false`	When set to `true`, the graphic objects inside a collapsed subgraph are created only the first time the subgraph is expanded. This can be used to reduce the startup time of an application.
savingExpandedState	boolean	`false`	If `true`, the state of subgraphs (expanded or collapsed) can be saved to the data model.
subObjectsSelectionAllowed	boolean	`true`	Allows/forbids the selection of subnodes and sublinks in an expanded node.

The following table lists the per-object rendering properties of the `SubGraph` renderer.

Per-object properties of the SubGraph renderer

Property	Type	Default	Description
Expandable	boolean	`true`	This property determines whether a node becomes a subgraph. If the value is `true`, the node is represented as a subgraph that can have children. If the value is `false`, the node is represented as a standard node and its potential children are displayed at the root level.
FrameGraphic	IlvGraphic		Specifies a custom graphic object for the frame. You can set this property to the special value "`default`" to use a default manager frame (displayed as a rectangle with a title bar).
FrameMargin	float, or float,float,float,float	10	Sets the margins between the frame and the subobjects. If a single value is specified, it is used

Property	Type	Default	Description
			for the four sides. If four values are specified, they represent the left, top, right, and bottom margins.
InitiallyExpanded	boolean	`true`	Sets the initial state of the expandable node.
MinusGraphic	IlvGraphic		Specifies a custom graphic object for the “collapse” icon.
MinusGraphicPosition	int	TopLeft	Specifies the position of the “collapse” icon in the node’s bounding box.The possible values are defined by the class `IlvDirection`.
PlusGraphic	IlvGraphic		Specifies a custom graphic object for the “expand” icon.
PlusGraphicPosition	int	TopLeft	Specifies the position of the “expand” icon in the node’s bounding box.The possible values are defined by the class `IlvDirection`.

Note: For compatibility with previous versions of IBM® ILOG® JViews, the name `ExpandCollapse` may be used in the style sheet instead of the name `SubGraph`. The two names are equivalent.

See the class `IlvSubGraphRenderer` for more details.

The SwimLanes renderer

The `SwimLanes` renderer displays vertical or horizontal stripes in the background of the diagram. Swim lanes are used to represent logical groups of nodes. For example, in a workflow process, each swim lane could represent a department of a company, and the activities executed within each department would be placed in the corresponding swim lane.

Swim lanes are represented graphically by vertical or horizontal rectangles that cover the whole view. Each swim lane rectangle contains (geometrically) all the nodes that belong to the swim lane. Each swim lane has a different color which can be automatically generated or customized through the style sheet. Each swim lane also has a title.

The `SwimLanes` renderer can be used alone, but it is most often used in conjunction with a `GraphLayout` renderer configured to use a hierarchical layout algorithm, because the hierarchical layout has the ability to automatically arrange the nodes of the hierarchy according to the swim lanes to which they belong.

The hierarchical layout ensures that the swim lanes do not overlap, but if no hierarchical layout is applied, or if the nodes are moved after the layout is applied, the swim lanes may overlap.

There are two versions of the `SwimLanes` renderer, corresponding to two different implementations. The first one, `IlvSwimLaneRenderer`, is a basic implementation of swim lanes, internally managed by the renderer. The second one, `IlvLaneRenderer`, differs from the first one in several ways:

- ♦ Each lane is the graphical representation of a dedicated object in the data model. The advantage of this new implementation is that the properties of the lanes (position, size, label...) can be stored in the data model.
- ♦ `IlvLaneRenderer` supports hierarchical swim lanes, that is, lanes containing sub-lanes. Lane containers are sometimes called “pools”.
- ♦ Interactions are different (see *IlvLaneRenderer* below).

To select one version of the renderer or the other, specify the following in the SDM rule:

```
SDM {
 SwimLanes : true;
}
```

or

```
SDM {
 Lanes : true;
}
```

`IlvLaneRenderer` is used in the BPMN modeler.

IlvSwimLanesRenderer

Swim lanes are computed according to the value of the `SwimLaneConstraints` property in the style sheet. The value of this property is examined for each node, and a swim lane is created for each different value. Note that this is the same property as the one used by the `GraphLayout` renderer, which makes sure that the `GraphLayout` and `SwimLanes` renderers group nodes into swim lanes in the same way.

For example, the following code example defines two swim lanes: one representing the R&D department of a company, and another one representing the Sales department.

```
node.activity[department=”R&D”] {
   SwimLaneConstraint : “R&D Department”;
}
node.activity[department=”Sales”] {
   SwimLaneConstraint : “Sales Department”;
}
```

With these rules, the diagram will be divided into two swim lanes with different colors, and these will have the titles `“R&D Department”` and `“Sales Department”`.

The following table lists the properties of the `SwimLanes` renderer.

Global properties of the SwimLanes renderer

Property	**Type**	**Default**	**Description**
alpha	float	0.5	The alpha value (that is to say the transparency) of the allocated colors.
brightness	float	0.8	The brightness of the allocated colors.
defaultSwimLanes	String	`null`	Can be used to create default swim lanes in an empty diagram. The string is a comma-separated list containing the titles of the default swim lanes.
draggingEnabled	boolean	`true`	Allows you to drag a node from one swim lane to another.
hue	float	0.2	The hue with which the color allocation algorithm will start.
layer	int	0	The index of the manager layer in which the swim lane graphics will be added.
margin	int	5	The margin between two adjacent swim lanes.
saturation	float	0.5	The saturation of the allocated colors.

The following table lists the per-object rendering properties of the `SwimLanes` renderer.

Per-object properties of the SwimLanes renderer

Property	Type	Default	Description
SwimLaneConstraint	Object	`null`	Defines the swim lane to which the node belongs.
SwimLaneGraphic	IlvGraphic	`null`	Lets you customize the graphic object used to represent the swim lane. By default, an `IlvGeneralNode` is used.
SwimLaneLabel	String	`null`	Lets you customize the title of the swim lane. By default, the title is the value of the `SwimLaneConstraint` property.
SwimLaneColor	Color	`null`	Lets you customize the color of the swim lane. By default, the colors are allocated automatically, based on the hue, saturation, brightness, and alpha properties of the renderer.

You should set the `SwimLaneGraphic`, `SwimLaneColor`, and `SwimLaneLabel` properties on all nodes. Although the SwimLane Renderer makes use of only the first node in a swim lane to set these properties, you may not find it easy to keep track of which node is the first node.

See the class `IlvSwimLanesRenderer` for more details.

IlvLaneRenderer

The Lane Renderer defines properties to customize the created lanes.

The following table lists the properties of a lane.

Lane properties

Property	Type	Default	Description
defaultLength	float	`500`	Default length of the lanes, that is, the width if the lanes are horizontal, or the height if they are vertical.
defaultSize	float	`100`	Default size of the lanes, that is, the height if the lanes are horizontal, or the width if they are vertical.
horizontal	boolean	`true`	Lane orientation.
margin	float	`5`	Margin between the objects inside the lane and the edges of the lane.
spacing	float	10	Minimum spacing between adjacent lanes.
sublaneOffset	float	0	If the lanes are horizontal, this property changes the offset between the left side of a sublane and the left side of the parent lane. This offset is used to leave room for a label.

To create a lane, the style sheet rule that matches the lane object must contain the declaration `LaneName:<name>`, where `<name>` is the name of the lane. If this rule also creates an

`IlvGraphic` that implements `ilog.views.IlvLabelInterface`, then this graphic will be the label of the lane and will appear in front of the lane. In general, the class of the lane graphic representation is `ilog.views.sdm.graphic.IlvDefaultLaneGraphic`, which displays vertical labels for horizontal lanes. In addition to `LaneName`, two more properties are then interpreted:

Lane additional properties

Property	Type	Default	Description
LaneName	string	""	Sets the matching node as a lane (or a pool) and defines its name.
LaneLength	float	defaultLength	Overrides the lane default length.
LaneSize	float	defaultSize	Overrides the lane default size.

Following is an example of a lane specification in the style sheet:

```
  node.Pool, node.Lane {
   // These properties define the look of pools and lanes.
   // The IlvDefaultLaneGraphic is a subclass of IlvGeneralNode
   // that can display its label vertically (when the lane is horizontal).
   //
   class : "ilog.views.sdm.graphic.IlvDefaultLaneGraphic";
   shapeType : "Rectangle";
   fillStyle : "SOLID_COLOR";
   strokeWidth : "1";
   horizontal : "true";
   label : "@Name";
   toolTipText : "@Name";
   labelPosition : "Center";
   labelSpacing : "5";
   minLabelZoom : "0";

   // This property identifies the object as a Pool or a Lane,
   // and tells the Lane renderer that it must handle it.
   //
   LaneName : "@Name";

   Anchor : "TopLeft";
   // Tell the link layout to preserve pools/lanes.
   LayoutFixed : "true";
}
```

The following property can be set on any node to state that it belongs to a lane.

Property	Type	Default	Description
Lane	string	""	Sets the matching object in the named lane.

Example:

```
node[lane] {
   Lane : "@lane";
}
```

This rule states that all nodes that define an attribute "`lane`" are assigned to the lane with the "`lane`" attribute value.

Lane Behavior

When editing is allowed, a lane has the following behaviors:

- ♦ Nodes inside the lane can be moved anywhere. The pools and lanes resize automatically to adjust to the new node positions. Note that, to remove an object from a lane, all you need to do is remove the Lane property from the node.
- ♦ The lane can be manually resized: When the mouse is close to a lane edge, a resize cursor appears, which indicates that the lane can be resized if the mouse is pressed down.

See the class `IlvLaneRenderer` for more details.

Adding your own renderer

Explains how to extend the rendering features of SDM. Shows how to add your own renderer.

In this section

Writing a renderer class
Describes how to write a renderer class.

Registering a renderer
Shows how to register the renderer class in the style sheet.

Loading and customizing a renderer
Shows how to load and customize the registered renderer.

Writing a renderer class

SDM renderers are instances of subclasses of `IlvSDMRenderer`. Instead of creating an instance of this class, you can write a subclass of the Filter class, `IlvFilterSDMRenderer`, which is the base class for renderers that modify the style sheet rendering without completely replacing it.

Graph Layout renderers are subclasses of `IlvGraphLayoutRenderer` . For specific examples, see Using Graph Layout algorithms.

Registering a renderer

Once you have written the renderer class, you must register it in the style sheet. To do this, you can set the `Renderers` property in the SDM customization rule. The following code example shows the general form of the declaration.

The SDM customization rule

```
SDM {
   Renderers : "MyRendererName=my.package.MyRenderer";
}
```

In this declaration of the `Renderers` property,`MyRendererName` is the name that you will use to customize your renderer; `my.package.MyRenderer` is the full class name of your renderer class.

This declaration causes `addRendererAlias(java.lang.String, java.lang.String)`, to be called. The other way to register a renderer is to call this method directly in your Java™ code.

If you register the renderer in the stylesheet, then registering, loading and customizing can be seen as a single integration step.

Loading and customizing a renderer

Once the renderer is registered in the style sheet, you can load and customize it by adding to the style sheet. The following code example shows the property to set to load the renderer and the general form of customization declarations (highlighted).

Renderer rules

```
SDM {
   Renderers      : "MyRendererName=my.package.MyRenderer";
   MyRendererName : "true";
}
MyRendererName {
   property1 : "value1";
   property2 : "value2";
}
```

The Flag renderer example

Provides an example of a custom renderer that displays a flag on the node that remains visible even if the object is hidden.

In this section

Overview
Provides an overview of the example.

Header part
Describes the `FlagRenderer` class as used in this example.

Bean properties
Describes the Bean properties as used in this example.

Private methods
Describes the private methods of the `FlagRenderer` class as used in this example.

Overloading methods of the Filter class
Describes overloading in the `FlagRenderer` class.

Integrating the Flag renderer
Shows how to integrate the new Flag renderer.

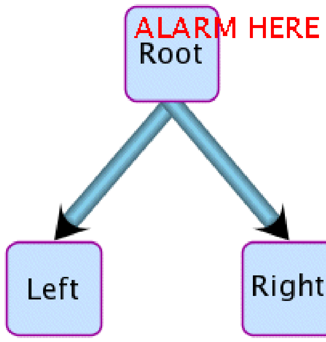
Possible enhancements
Describes enhancements that could be made to the Flag renderer example.

Overview

The Flag Renderer example addresses a typical graphical need that can be answered with an SDM renderer. The need is to set a decoration on a node that will always be visible, for example, if the state of a model object changes and it needs to catch the user’s attention. The renderer must display a flag on the node such that the flag remains visible even if the object is hidden by another graphic object.

The source code of the Flag renderer example is supplied in the **<installdir>/jviews-diagrammer86/codefragments/renderer** directory and described in this section.

The following figure shows an example of a flag on a node, as it would be displayed by the Flag renderer.



An example of a flag displayed by the Flag renderer

Header part

The class `FlagRenderer` extends `IlvFilterSDMRenderer`, so it does not need to redefine all unused renderer methods.

The following code example shows the packages imported and the class declaration.

The statements for the class

```
package tutorial;
import ilog.views.sdm.*;
import ilog.views.sdm.model.*;
import ilog.views.sdm.graphic.*;
import ilog.views.sdm.renderer.*;

import ilog.views.*;
import ilog.views.graphic.*;
/**
 * The class <code>FlagRenderer</code> is a filtering renderer that
 * sticks various IlvGraphic (the flags) to a node.  Flags are located
 * in a separate layer, typically over all other layers to have them
 * always visible.

 * <P>This renderer defines the following graphic properties:
 * <UL>
 * <LI>Flag: an  IlvGraphic that represents the decorations<\LI>
 * <\UL>
 * */
public class FlagRenderer  extends IlvFilterSDMRenderer
{
```

The following code example shows the internal variables and the constructors.

Internal variables and constructors

```
    private int _flagLayer = 20;
    static final String[] REND_CLASS = {"renderer", "flagRenderer" };
    static final String FLAG = "Flag";
    static final String FLAG_GRAPHICS = "Flag-graphic";
    //////////////////////
    // constructors

  /**
   * Creates a new flag renderer for a specified
   * filtered renderer.
   */
  public FlagRenderer(IlvSDMRenderer renderer) {
    super(renderer);
  }

  /**
   * Creates a new flag renderer with a <code>null</code>
   * filtered renderer.
   */
  public FlagRenderer(){
    this(null);
  }
```

The internal variables are as follows:

- `_flagLayer` is the layer where the flags will be added.
- `REND_CLASS` is the CSS pseudo-class of this renderer, for use in the style sheet.
- The other constants set string literals.

Bean properties

The Bean properties allow you to customize the renderer.

In this example, you need to change only the flag layer value. The setter and getter methods are shown in the following code example.

Setter and getter methods for the Bean property Flag Layer

```
    /////////////////////////
    // Bean properties


    /**
     * Sets flag layer. Default is 20.
     * */
    public void setFlagLayer(int v) {
        _flagLayer = v;
    }

    /*
     * Returns flag layer.
     * @see setDebugMask
     */
    public int getFlagLayer() {
        return _flagLayer;
    }
```

Private methods

The following private methods of the Flag Renderer handle the flag on a node:

- `checkNode`
- `cleanNode`
- `moveNode`

The following code example shows the code for `checkNode`.

Checking for a flag

```
//////////////
    // local method
The checkNode method fetches the graphic property to see if a flag is defined
 for the node in its current state.
    // manage graphic props
    private void checkNode(IlvSDMEngine engine, Object obj,
                           IlvGraphic graphic) {
        Object rawFlag = IlvRendererUtil.getGraphicProperty(engine, obj,
                                               FLAG, REND_CLASS, null);
        if (rawFlag != null && rawFlag instanceof IlvGraphic) {
            IlvGraphic g = (IlvGraphic) rawFlag;
            // set flag location at top left corner
            IlvRect r = graphic.boundingBox(null);
            g.move(r.x, r.y);
            // add object
            engine.getGrapher().addObject(g, _flagLayer, true);
            // save flag in the graphic itself
            graphic.addProperty(FLAG_GRAPHICS, g);
        }
    }
```

The `checkNode` method checks if a flag is requested and renders the flag if necessary by adding it at the top left of the object.

The following code example shows the code for `cleanNode`.

Removing a flag

```
// remove previous flag
    private void cleanNode(IlvSDMEngine engine, IlvGraphic graphic) {
        // get the flag object from the source
        Object previous = graphic.getProperty(FLAG_GRAPHICS);
        if (previous != null) {
            // remove it and clear the property
            engine.getGrapher().removeObject((IlvGraphic)previous,true);
            graphic.removeProperty(FLAG_GRAPHICS);
        }
    }
```

The `cleanNode` method is used to remove the flag when it is no longer needed.

The following code example shows the code for `moveNode`.

Moving the flag with the node

```
// adjust flag position
private void moveNode(IlvRect newBBox, IlvGraphic flag) {
    if (flag == null)
        return;
    flag.move(newBBox.x, newBBox.y);
}
```

The `moveNode` method is used to move only the flag, according to the new bounding box of the node on which it appears.

Overloading methods of the Filter class

The Flag Renderer overloads the following `IlvFilterSDMRenderer` methods:

- ♦ `addNodeGraphic(ilog.views.sdm.IlvSDMEngine, java.lang.Object, ilog.views.IlvGraphic, boolean)`
- ♦ `propertiesChanged(log.views.sdm.IlvSDMEngine, java.lang.Object, java.util.Collection, ilog.views.IlvGraphic)`
- ♦ `removeNodeGraphic(ilog.views.sdm.IlvSDMEngine, java.lang.Object, ilog.views.IlvGraphic, boolean)`
- ♦ `nodeGraphicBBoxChanged(ilog.views.sdm.IlvSDMEngine, java.lang.Object, ilog.views.IlvGraphic, ilog.views.IlvRect, ilog.views.IlvRect, java.lang.String[])`

The overloaded methods start by calling `super`, which calls the superclass method of the same name before processing the flag.

Creating the flag

```
/**
 * Creates flag.
 * */
public void addNodeGraphic(IlvSDMEngine engine,
                       java.lang.Object node,
                             IlvGraphic graphic,
                                boolean redraw) {
   super.addNodeGraphic(engine,node,graphic,redraw);
   checkNode(engine, node, graphic);
   }
```

The `addNodeGraphic` method creates the flag when a new node is added.

Cleaning and re-creating a flag

```
/**
 * Recreates flag.
 * */
public void propertyChanged(IlvSDMEngine engine,
                        java.lang.Object object,
                        java.lang.String propertyName,
                        java.lang.Object oldValue,
                        java.lang.Object newValue,
                              IlvGraphic graphic) {
            super.propertyChanged(engine,object,propertyName,
            oldValue,newValue,graphic);
            if (!engine.getModel().isLink(object)) {
               cleanNode(engine, graphic);
               checkNode(engine, object, graphic);
               }
            }
```

A property change may or may not change the flag. Therefore, the `propertyChanged` method clears the flag and re-creates it.

Clearing a flag

```
    /**
     * Clears flag.
     * */
    public void removeNodeGraphic(IlvSDMEngine engine,
                              java.lang.Object node,
                                    IlvGraphic graphic,
                                       boolean redraw){
              super.removeNodeGraphic(engine, node, graphic, redraw);
              cleanNode(engine, graphic);
    }
```

The `removeNodeGraphic` method clears the flag when a node is removed.

Moving the flag with the node

```
    /**
     * Adjusts flag position.
     * */
    public void nodeGraphicBBoxChanged(IlvSDMEngine engine,
                java.lang.Object node,
                IlvGraphic graphic,
                IlvRect oldBBox,
                IlvRect newBBox,
                java.lang.String[] pseudoClasses) {
           super.nodeGraphicBBoxChanged(engine,node,graphic,
                     oldBBox,newBBox,pseudoClasses);
              moveNode(newBBox,(IlvGraphic)graphic.getProperty(FLAG_GRAPHICS)
);
           }
```

The `nodeGraphicBBoxChanged` method is called when the bounding box of the node is changed. Note that no property change event is sent, although x and y may change.

Integrating the Flag renderer

There are two ways to integrate a new renderer:

- *Declare the renderer*
- *Register the renderer*

Declare the renderer

For maximum efficiency, declare the flag renderer in the style sheet.

In this example, there is a specific rule to create the flag if the model object matches a particular state. The `Flag` renderer fetches the graphic property `Flag` from the rule to get the `IlvGraphic` object that will represent the flag, in this case the label “ALARM HERE”.

The following code example shows the style rules needed.

The style rules for the flag

```
node[state=alarm] {
      Flag : @#alarm
}

Subobject#alarm {
        class        : 'ilog.views.sdm.graphic.
IlvGraphicFactories$ZoomableLabel' ;
        label        : "ALARM HERE" ;
        antialiasing : true ;
        leftMargin   : 5 ;
        rightMargin  : 5 ;
        topMargin    : 5 ;
        bottomMargin : 5 ;
        foreground   : red ;
}

node {
     Flag : ’’ ;
}
```

The rules operate as follows:

1. The first rule sets an indirection to create the `IlvGraphic` object when it is needed.
2. The second rule creates the flag as a simple red label.
3. The third rule clears the flag when the node reverts to its normal state. The default value for the flag is then null.

Register the renderer

The simplest way to integrate a renderer is to register the renderer so that it will be understood when the style sheet is read.

To register a renderer, just call the static method `addRendererAlias(java.lang.String, java.lang.String)`, in the class `IlvRendererUtil`.. You must do this before you load a style sheet. The following code example shows the code line (highlighted) for the Flag Renderer.

Registering the Flag renderer in Java™ code

```
public static void main(String[] args)    {
   IlvRendererUtil.addRendererAlias("Flag", "tutorial.FlagRenderer");
   SDMViewer v = new SDMViewer();
   v.init(args);
}
```

The arguments of the method are as follows:

♦ First argument: the symbolic name used in the style sheet

♦ Second argument: the fully qualified class name of the renderer, which will be dynamically loaded when needed

After being registered, the Flag renderer is ready to use in a style sheet, like any other renderer. The following code example shows a simple example of rules that use it.

Simple style rules for a registered Flag renderer

```
SDM {
   Flag : true ;
}
Flag {
   flagLayer : 30 ;
}
```

Possible enhancements

The Flag renderer example is deliberately kept simple. You could make various enhancements, for example, the following:

♦ Visibility

If a node changes its visibility, the flag remains visible. This may seem to be what is wanted at first sight, because the flag is raised even if the node is not present. However, a dangling flag seems strange. Therefore, a better implementation is to listen to the `IlvManager`. property changes on its objects, and then set the visibility of the flag according to the visibility of the node.

♦ Position

You can position the flag somwhere other than the top-left corner. For example, you can set the flag location as an offset when you create it, and use another graphic property, `FlagAnchor`, to set the position of the corner of the node to which the flag is to be attached.

♦ Several flags

You can add several flags to the same node. Make sure that you control the layout of the flags so that they operate in a coherent way.

Configuring renderers in Java code

Shows how to customize a renderer using Java™ .

In this section

Overview
Presents an overview of customizing renderers using Java™ .

Accessing a renderer
Describes the method used to access renderers.

Modifying a renderer
Explains how to modify a renderer.

Setting new renderers
Describes the method used to set a new renderer.

Overview

The simplest and preferred way to customize renderers is through the style sheet, as explained in *Using renderers in the style sheet*. There are situations, however, where it is necessary to access and modify renderers dynamically while the application is running. For example, you may want to disable the `LinkLayout` renderer temporarily and re-enable it later without reloading a new style sheet, which would re-create all the graphic objects. This section explains how to do this in your Java™ code.

Accessing a renderer

You can access renderers by calling the method `getRenderer()` of the class `IlvSDMEngine`.

This method returns the first element of the list of renderers attached to the SDM engine. All but the last element of this list are instances of subclasses of `IlvFilterSDMRenderer`. You can retrieve the next element of the list by calling the method `getFilteredRenderer()`. The last element of the list is usually the `StyleSheet` renderer, which is an instance of the class `IlvStyleSheetRenderer`.

To make it easier to retrieve a particular renderer, the class `IlvRendererUtil` provides a static method `getRenderer(ilog.views.sdm.IlvSDMEngine, java.lang.String)` that looks up a renderer by its name in the chained list.

The following code example shows how to retrieve the `LinkLayout` renderer.

Retrieving a renderer in Java™ code (symbolic name)

```
IlvSDMEngine engine = ...;
IlvLinkLayoutRenderer r = (IlvLinkLayoutRenderer)
    IlvRendererUtil.getRenderer(engine, "LinkLayout");
```

The name passed to the `getRenderer` method can be either the symbolic name of the renderer (the name that is used in the style sheet) as shown in *Retrieving a renderer in Java™ code (symbolic name)* or the full class name of the renderer as shown in the following code example.

Retrieving a renderer in Java code (class name)

```
IlvSDMEngine engine = ...;
IlvLinkLayoutRenderer r = (IlvLinkLayoutRenderer)
    IlvRendererUtil.getRenderer(engine,
                                IlvLinkLayoutRenderer.class.getName());
```

Modifying a renderer

Once you have retrieved the reference to the renderer object, you can simply call its methods to modify it. For example, the following code example shows how to disable and re-enable the `LinkLayout` renderer.

Disabling and re-enabling the LinkLayout renderer in Java™ code

```
IlvLinkLayoutRenderer r = ...;
r.setEnabled(false);
...
r.setEnabled(true);
```

Setting new renderers

The renderers are usually chained by the method `getFilteredRenderer`. The first one is returned by the method `getRenderer()`. If the first renderer must be changed, just call the method `setRenderer(ilog.views.sdm.renderer.IlvSDMRenderer)`, specifying the new renderer.

For example, the following code example shows how to add a renderer to the front of the chained list.

Adding a renderer in Java™ code

```
public void prependRenderer(IlvSDMEngine engine,
                            IlvFilterSDMRenderer newRenderer) {
   IlvSDMRenderer previous = engine.getRenderer();
   newRenderer.setFilteredRenderer(previous);
   engine.setRenderer(newRenderer);
}
```

Support for renderers in the Designer

The following table shows which renderers can be customized in the Designer.

Renderers supported in the Designer

Renderer Name	Supported?
Animation	No
Coloring	No
custom renderer	No
Decoration	No
GrapherPin	No
GraphLayout	Yes
HalfZooming	Yes
InfoBalloon	Yes
Interactor	No
LabelLayout	No
Legend	Yes
LinkLayout	Yes
Map	Yes
Subgraph	Yes

Using and writing interactors

Explains what an interactor is and how to enable and customize one, lists all predefined interactors, and gives examples of writing your own interactor in Java™ code and integrating it through Java code or through the style sheet.

In this section

Predefined interactors
Lists the predefined interactors provided in IBM® ILOG® JViews Diagrammer, and their relationships.

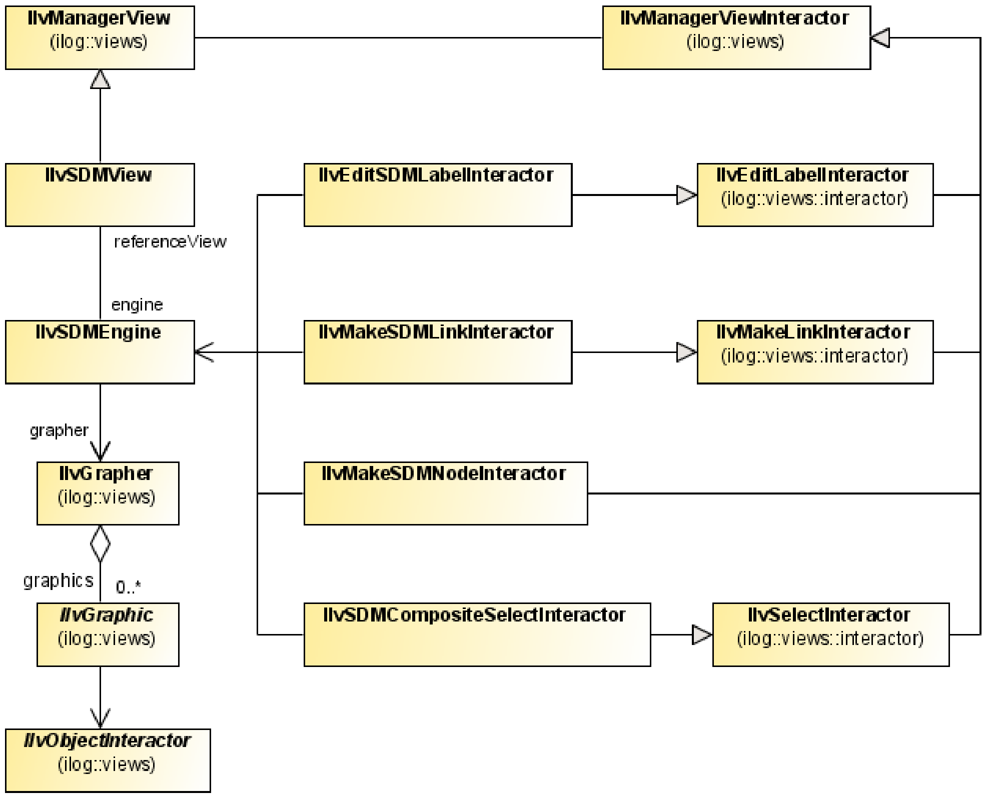
Subclassing view interactors
Shows how to subclass view interactors.

Writing an object interactor
Describes the steps involved in writing an object interactor.

Predefined interactors

An *interactor* is a Java™ class that manages the behavior of an object in response to an event. IBM® ILOG® JViews Diagrammer supplies various predefined interactors.

There are two types of interactor: view and object. You can control the default interactor settings through the style sheet.



Predefined Interactors Class Relationships

View interactors

View interactors affect the display in general. The view interactors of JViews Framework are available in IBM® ILOG® JViews Diagrammer to manage the zoom and pan facilities, and the selection and creation of general nodes and general links. These interactors are in the `ilog.views.interactor` package.

There are two object creation interactors for SDM objects:

♦ `IlvMakeSDMNodeInteractor` creates a new node in the data model.

♦ `IlvMakeSDMLinkInteractor`, creates a new link in the data model.

These interactors create an object as a copy of a supplied “model” object. For JViews Diagrammer applications, you can use the palette bar class (`IlvDiagrammerPaletteBar`) to set up a palette of model objects from which to create objects in a diagram.

There is also another interactor for new SDM objects:

♦ `IlvEditSDMLabelInteractor` allows you to edit the label of a new object as soon as it is created.

Object interactors

An object interactor is local to the graphic object. That means it is possible to specify this type of interactor in the style sheet. There is a renderer called `Interactor` which fetches the `Interactor` graphic property from the style sheet to attach an object interactor to the graphic object. The value of this graphic property is the name of a class that subclasses `IlvObjectInteractor`. The object interactor is then free to do whatever it wants, starting from the graphic object. For example, the `InfoBalloon` renderer subclasses the `Interactor` renderer to display an enhanced tooltip.

Declaring a default interactor

You can enable the `Interactor` renderer in the style sheet. You can also define a default object interactor for all nodes and links, see the following code example:

The Interactor renderer and a default interactor in the style sheet

```
SDM {
   Interactor : true ;
}
Interactor {
   interactor : "MyDefaultInteractor" ;
}
```

Subclassing view interactors

You can subclass the `IlvMakeSDMNodeInteractor` and the `IlvMakeSDMLinkInteractor` to control when a link is created between two nodes, or to perform custom actions when links or nodes are created.

For example, to create a link only between certain types of node, you can subclass `IlvMakeSDMLinkInteractor` and override the `acceptOrigin` and `acceptDestination` methods, see the following code example:

Subclassing the IlvMakeSDMLinkInteractor

```
public class ValidatingLinkInteractor extends IlvMakeSDMLinkInteractor
{

  protected boolean acceptOrigin(IlvPoint p, IlvGraphic fromNode) {
    Object o = getEngine().getObject(fromNode);
   if("false".equals(getEngine().getModel().getObjectProperty(o, "okForOrigin")
))
      return false;
      return super.acceptOrigin(p, fromNode);
  }

  protected boolean acceptDestination(IlvPoint p, IlvGraphic toNode) {
    Object o = getEngine().getObject(toNode);
    if("false".equals(getEngine().getModel().getObjectProperty(o,
"okForDestination")))
      return false;
      return super.acceptDestination(p, toNode);
  }
}
```

In this example, a node whose property `"okForOrigin"` is false is not accepted as the origin of a link, and a node whose property `"okForDestination"` is false is not accepted as the destination of a link.

You can also set custom properties when a node or a link is created by overriding `setNodeProperties` or `setLinkProperties`. For example, to prompt the user for the `"name"` property of a node as it is created, you can subclass `IlvMakeSDMNodeInteractor` and override `setNodePorperties` as follows:

Subclassing the IlvMakeSDMNodeInteractor

```
public class PromptingNodeInteractor extends IlvMakeSDMNodeInteractor
{
  protected void setNodeProperties(IlvSDMModel model, Object node) {
    String label = JOptionPane.showInputDialog(getManagerView(), "Name:");
    if(label != null)
    model.setObjectProperty(node, "name", label);
  }
}
```

Writing an object interactor

Describes the steps involved in writing an object interactor.

In this section

Writing a subclass of IlvObjectInteractor
Shows how to write a subclass of an interactor.

Enabling a custom interactor
Describes how to enable the custom interactor.

Connecting interactors to diagrams using listeners
Shows how to connect the custom interactor to the diagram.

Writing a subclass of IlvObjectInteractor

The following code example shows an outline of the code in `SwitchInteractor` to illustrate a typical interactor.

A typical interactor in Java™ code

```
public class SwitchInteractor extends ilog.views.IlvObjectInteractor {

    /**
     * Callback when user clicks a graphic object
     * */
    public boolean processEvent(IlvGraphic obj,
                                java.awt.AWTEvent event,
                                IlvObjectInteractorContext context) {
        if (obj.getGraphicBag() instanceof IlvGrapher) {
            // find sdm engine
            IlvSDMEngine engine = IlvSDMEngine.getSDMEngine(
                                             (IlvGrapher)obj.getGraphicBag())
;
            // find object model
            if (engine != null) {
                Object modelObject = engine.getObject(obj);
                // cast and do whatever is needed here
                ...
                // event has been processed
                return true;
            }
        }
        // event has not been processed
        return false;
    }
}
```

Enabling a custom interactor

Before you can enable a custom interactor in the style sheet, you must enable the `Interactor` renderer. You then specify the `Interactor` graphic property to override the default interactor for the objects in the selector. The following code example shows how to specify an interactor for switchable nodes.

Attaching a custom interactor to an object type in the style sheet

```
SDM {
   Interactor : true ;
}

...

node.switchable {
   Interactor : ’SwitchInteractor’ ;
}
```

Note: A custom interactor such as SwitchInteractor will not be available in the Designer.

Connecting interactors to diagrams using listeners

In the Designer you can pass the status of a `push_state` interactor in a node to a Designer property. For information on how to do this, see Linking predefined interactors to parameters in *Using the Designer*. To customize the application behavior, use a Java™ listener to provoke custom behavior in your application when the node is clicked. The following code sample shows how to do so:

A listener to follow changes in diagram property values

```
/**
 * Listener for user  interaction. Assume that in the style sheet
 * there is a
 * mapping:
 *   push_state : "@state";
 */
public class ModelListener implements SDMPropertyChangeListener {
 /*
 * @see ilog.views.sdm.event.SDMPropertyChangeListener#propertyChanged(
 *          ilog.views.sdm.event.SDMPropertyChangeEvent)
 */
 public void propertyChanged(SDMPropertyChangeEvent event) {
   String propertyName = event.getPropertyName();
   Object target = event.getObject();
   String value = event.getNewValue().toString();

   if ("state".equals(propertyName)) {
      // push interactor has modified the model
      // invoke callback
      performSomeAction(target, value);
   }
 }
}

...

//Usage:
   IlvDiagrammer diag;
   diag = new IlvDiagrammer();
   diag.getEngine().getModel().addSDMPropertyChangeListener(new
          ModelListener());
```

Managing dynamic symbols

Shows symbol and composite graphic features, as well as how to manipulate symbols and composite graphics programmatically.

In this section

Introducing symbols
Explains symbols, their advantages, and how to use them.

Using symbols
Shows how to use symbols.

Advanced management of symbols and palettes
Addresses advanced users who wish to create symbols in memory by themselves without loading an existing palette from a JAR file.

How to use a symbol in CSS
Describes how to declare and use symbols in CSS documents.

Using composite graphics
Shows how to use composite graphics in your diagrams.

Introducing symbols

Explains symbols, their advantages, and how to use them.

In this section

What is a symbol?
Provides an overview of symbols in IBM® ILOG® JViews Diagrammer.

The advantages of symbols
Lists the advantages of symbols and the benefits that they provide.

What is a symbol?

A symbol is a collection of graphic objects, parameters and conditions used to give a dynamic representation of changing data.

A CSS file is used to store the information necessary to represent a symbol, including the type and position of shapes, text, IVL files, images, and other symbols used to make up the symbol. Symbol data is stored inside the palette file structure. Palettes are stored in standard Java™ `.jar` archive files. Once created, a symbol is manipulated either by being imported into other IBM® ILOG® JViews products such as the Designer, the Dashboard Editor, or directly using the IBM® ILOG® JViews Java API.

Elements inside a symbol can be static to represent specific information or dynamic to display status according to values extracted from external data sources. For example, you can use a PNG image showing a network structure and add elements which change color to show bandwidth on hubs in the network. Other examples of uses for symbols are:

- Indicators on a map, such as weather conditions, interesting locations or vehicle information.
- Gauges, sliders or vue-meters on a dashboard.
- Status indicators and alarms in a SCADA application.

Symbols contain interactors. An interactor listens for mouse events in a symbol, it uses the mouse movement to change the value of a symbol parameter. Changes to parameters result in changes to the graphic element to which they are bound. Using interactors you create symbols that react to user input such as:

- A slider or dial to control sound volume.
- Buttons to enable or disable options in a graphic equalizer.

The advantages of symbols

Symbols are reusable. They can be easily integrated into multiple GUI user applications. To make reuse easier, symbols are organized in categories stored in palettes. A palette is a container from which symbols can be selected and added to other symbols or applications using IBM® ILOG® JViews Diagrammer Symbol Editor, IBM® ILOG® JViews Diagrammer or IBM® ILOG® JViews Diagrammer Dashboard Editor. This is done graphically by dragging and dropping a symbol from an open palette in the GUI editor to a drawing area in the application.

IBM® ILOG® supplies GUI components used to create symbol, diagram and dashboard editing applications. To display a palette, the content of an `IlvPalette` is displayed in Java™ Swing `IlvPaletteViewer` instance. Three predefined types of viewer are provided.

An application that deals with several open palettes uses the `IlvPaletteManager` component. `IlvPaletteManager` contains several `IlvPalette` instances. An `IlvPaletteManager` is linkable to a `IlvPaletteManagerViewer` instance. This allows the user to select and display the current selected palette.

Symbols offer the following advantages:

- ♦ A predefined look and behavior
- ♦ Easy to create using the Symbol Editor
- ♦ Easy integration in application
- ♦ Highly customizable

Using symbols

Shows how to use symbols.

In this section

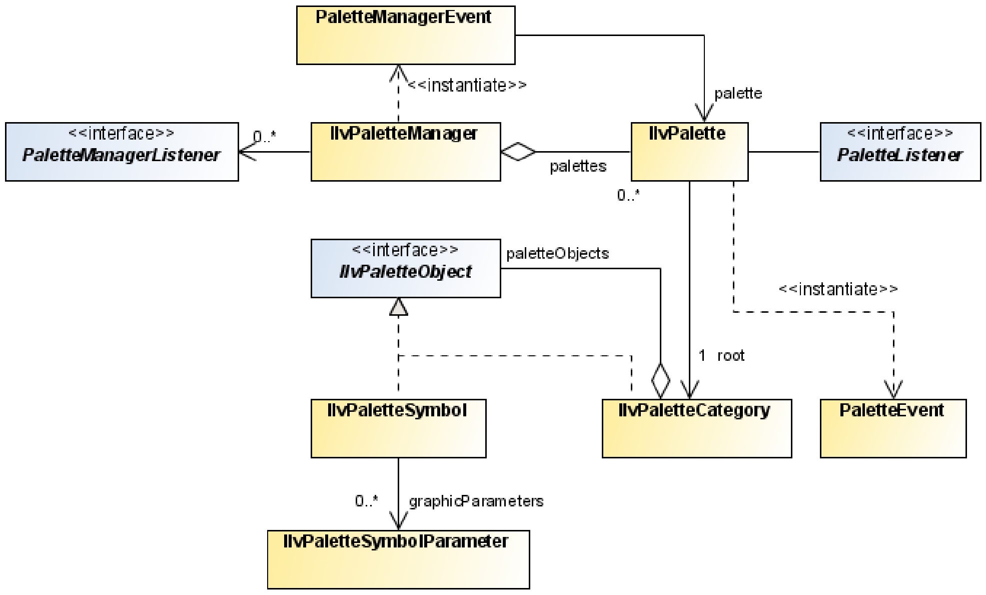
Basic concepts
Provides an overview of symbols and their classes.

Loading palettes
Explains symbol palettes and how to load them.

Saving palettes
Shows how to save symbol palettes.

Basic concepts

A symbol is made up from graphic elements, parameters, conditions and vector or bitmap images. Each graphic element has attributes which define its look and feel. A symbol is defined by the `IlvPaletteSymbol` class. This class defines a symbol. It contains all information necessary for CSS to create instances of the symbol in a GUI application.



Symbol palette classes available in IBM® ILOG® JViews Diagrammer

An `IlvPaletteSymbol` instance contains the following:

♦ The CSS that describes the symbol. This includes graphic element definitions and symbol conditions.

♦ The class name. This acts as the entry point in the CSS.

♦ The symbol parameters.

A *palette* (`IlvPalette`) is a collection of *palette symbols* (`IlvPaletteSymbol`). A palette symbol represents the definition of the symbol. Instances of the symbol based on the same definition may occur in the diagrammer application. An instance of a symbol has a symbol descriptor (`IlvSymbolDescriptor`) which helps find the palette symbol that corresponds to this instance.

Inside a palette, the symbols are organized in a hierarchy of *categories* (`IlvPaletteCategory`). Each palette has at least a root category.

The category path and the symbol name identify a symbol within a palette. For example, `Symbols.Controls.Dial` corresponds to the Dial symbol inside the subcategory Controls of the root category Symbols.

Part of the palette symbol definition indicates which parameters are defined for the symbol. Symbols can have a different behavior or appearance depending on the actual parameter values of the symbol instance. The palette symbol contains the definition of the formal parameter (`IlvPaletteSymbolParameter`), while the actual value of a parameter can be queried using `IlvSymbolDescriptor` for each instance of the symbol.

Finally, the *palette manager* (`IlvPaletteManager`) manages a collection of palettes. The palette manager is only needed when you want to load or save palettes dynamically in the user file system. If you need to load only one fixed palette, it is more convenient to put the palette on the class path and load the palette as resources of the application. This is explained in the next section.

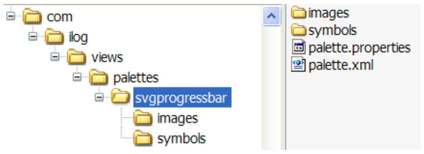
Note: An `IlvPaletteManager` must be attached to an IBM® ILOG® JViews Diagrammer Designer application to follow the currently loaded palettes.

Loading palettes

A palette is a JAR file. The content of the palette is described in an XML file, `palette.xml`, which defines the hierarchy of symbols and categories. The JAR file also contains subdirectories which store the following:

- ♦ Images used by the palette symbols
- ♦ The CSS file describing the symbols
- ♦ Additional resources

The following figure shows the structure of a palette JAR file.



The palette structure

There are two main scenarios when loading a palette:

- ♦ **Scenario 1**: The application uses a fixed set of symbols from a palette. In this case, the easiest way is to put the palette on the class path. No palette manager is needed. This also works for (unsigned) applets.
- ♦ **Scenario 2**: The application uses a variable set of palettes. The application allows you to dynamically load palettes from the user file system at any time. In this case, a palette manager is needed, since only a palette manager can load palettes that are not on the class path. Symbol applications developed as applets must be signed for them to be able to load palettes from the user machine.

To load a palette in Scenario 1:

1. Make sure the palette.jar file is in the class path.
2. Create a palette:

   ```
   new IlvPalette();
   ```

3. Load the palette data from the .jar file.

   You need to know the path to the palette description. For example, the description of the palette `lib/palettes/jviews-palette-shared-symbols-8.6.jar` is in `ilog/views/palettes/shared/palette.xml`. You see the path when you look at the palette in the Symbol Editor.

   ```
   palette.load(getClass().getResource(pathToPaletteXML));
   ```

   Now you have an `IlvPalette` object filled with the palette data.

4. Retrieve a symbol from the palette by specifying its category and name path.

```
IlvPaletteSymbol psymbol = palette.getSymbol("Symbols.Controls.Dial");
```

To load a palette in Scenario 2:

1. The palette.jar must be somewhere on the file system. It does not have to be on the class path. The file system must be accessible, that is, if the application is an applet, it must be signed to access the user's file system.

2. Load the palette through the palette manager:

   ```
   IlvPalette[] palettes = myIlvPaletteManager.load(new URL("file://c:/
   myPalette.jar"));
   ```

   Since the .jar file can contain multiple palettes, an array of palettes is returned. These palettes are also added to the palette manager.

3. Retrieve a symbol from the palette by specifying its category and name path, like in scenario 1.

   For example, if the palettes array contains only one palette:

   ```
   IlvPaletteSymbol psymbol = palettes[0].getSymbol("Symbols.Controls.Dial")
   ;
   ```

Saving palettes

Saving a palette on the file system can be done through the palette manager. The palette manager first saves the palette hierarchy in a working directory and then packages the working directory into a .jar file.

To save a palette:

1. Specify the working directory:

```
myIlvPaletteManager.setWorkingDirectory(new File("/tmp"));
```

2. Save the palette in the current directory:

```
myIlvPaletteManager.save(palette, new File("myPalette.jar"));
```

When a palette is saved, symbol resources, such as images which were imported from outside the palette .jar file are copied into it. Links to these resources are updated accordingly. Therefore, the resulting file `myPalette.jar` is self-contained, that is, it does no longer contain any reference to external files.

Advanced management of symbols and palettes

Addresses advanced users who wish to create symbols in memory by themselves without loading an existing palette from a JAR file.

In this section

Palettes
Describes the classes used to create and modify palettes.

Categories
Describes categories, the logical units used to organize symbols.

Palette symbols
Shows how to use palette symbols.

Palette parameters
Shows how to use palette parameters.

Palette events
Describes the classes used to track palette events.

Palette manager
Shows how to add a palette to a palette manager.

Palette manager events
Describes the classes used to track palette manager events.

Organizing symbols in categories

Presents an example that shows how to organize symbols in palette categories.

Palettes

A palette is a collection of symbols that are organized into categories.

Palettes are defined by the `IlvPalette` class. You need to create a category in the palette and set it as the root category to be able to add symbols and other categories.

An `IlvPalette` instance contains the name, description and icon properties. These String properties are designed to be set for a specific locale.

The following example shows how to create some new palettes:

```
IlvPalette p1 = new IlvPalette();
IlvPalette p2 = new IlvPalette();
IlvPalette p3 = new IlvPalette();

p1.setDefaultLocale(Locale.FRENCH);
p1.setName("Voitures");
```

Categories

Categories are the logical units used to organize symbols.

The category is defined by the `IlvPaletteCategory` class. As well as symbols, a category can contain other sub-categories. You thus organize symbols in a tree of logical groups. You add a category to a symbol palette by calling:

```
//Create a new root category
String categoryID = "root";
IlvPaletteCategory rootCategory = new IlvPaletteCategory(categoryID);

//Add the root category to the palette
palette.setRoot(rootCategory);
rootCategory.setName(categoryID);

//Add a subcategory to the root category
IlvPaletteCategory subcategory = palette.addCategory(rootCategory,
"subCategory");
subcategory.setName("subcategory");
```

Palette symbols

An IlvPaletteSymbol can only exist within a palette.

You need first to create an IlvPalette, then set its root category (and more categories if needed), and finally add the symbol to a category in the palette:

```
IlvPalette palette = new IlvPalette();
palette.setRootCategory("Root");
IlvPaletteSymbol psymbol = palette.addSymbol(palette.getRoot(),"symbol1");
```

The symbol needs a CSS file and other resources.

You must make sure that the CSS file and the resource files are in the class path; otherwise, you must use the IlvPaletteManager.

Assuming that all the required files are in the class path:

```
psymbol.setName("Symbol");
psymbol.setClassName("Symbol");
psymbol.addResource(css);
psymbol.setCSSResourceName(css);
```

Palette parameters

Parameters contain values associated with a symbol. A parameter is bound to a graphic element to cause a transformation, that is, a change in the elements aspect to represent a change in data. For example, an icon that changes color to show an alert.

Parameters are set on a symbol using instances of `IlvPaletteSymbolParameter`. This class contains the definition of one parameter. You add a parameter to a symbol by calling:

```
//Create a new palette
IlvPalette palette = new IlvPalette();
palette.setName(name);
palette.setDescription(french, descr);

//Create a root category for the palette
String categoryID = "root";
IlvPaletteCategory rootCategory = new IlvPaletteCategory(categoryID);
palette.setRoot(rootCategory);
rootCategory.setName(categoryID);

//Create a new symbol
String symbolID = "symbol1";
IlvPaletteSymbol s = palette.addSymbol(rootCategory, symbolID);
s.setName(symbolID);

//Create a new parameter
IlvPaletteSymbolParameter p2 = new IlvPaletteSymbolParameter();
p.setID("progress");
p.setName("progress");
p.setValue(new Integer(0));

//Add the parameter to a symbol
s.addParameter(p);
```

Parameters can accept any value or only an allowed set of values, such as “wait”, “attention”, “go”, or 1, 2, 3. The description of the allowed values for a parameter is set using the `IlvPaletteSymbolParameterValueSet` class. You set allowed values for a class by calling:

```
//Create a new value set
String vsetID = "vset1";
IlvPaletteSymbolParameterValueSet vset = new
 IlvPaletteSymbolParameterValueSet(vsetID);
vset.setName(vsetID);
vset.setType("Integer");

vset.setValues(new int[] { 1, 2, 3 });
vset.setValueName(0, "one");
vset.setValueName(1, "two");
vset.setValueName(2, "three");

//Add the value set to a palette
```

```
palette.setValueSet(vset);

//Create a new parameter
IlvPaletteSymbolParameter p = new IlvPaletteSymbolParameter();
p.setID("progress");
p.setName("progress");
p.setValue(null);

//Add the value set to a parameter
p.setType("set");
p.setValueSet(vset);
```

Please note that the CSS of the symbol must be consistent with the parameters of the symbol. For example, if you wished to use the file Dial.css (in `ilog/views/palettes/shared/symbols/Dial.css`), you would need to define the parameters `name` and `value` for the symbol, as Dial.css refers to these parameters:

```

s.addParameter(
  new IlvPaletteSymbolParameter("value", new Double(0), "double", null));
s.addParameter(
  new IlvPaletteSymbolParameter("name", "DefaultName", "string", null));
...
```

The parameters constitute the visible interface of the symbol. They hide the internals of the CSS, so that it is only through the parameters that you can modify the symbol.

Palette events

When a symbol or a category is added in a palette, a `PaletteEvent` is fired to notify external objects.

IBM® ILOG® supplies the following classes to track palette events:

- `PaletteListener` - an interface for receiving `PaletteEvent`s.
- `PaletteAdapter` - the default implementation of `PaletteListener`.

Palette manager

The following code example shows how to add palettes to an `IlvPaletteManager` instance.

```
IlvPalette p1 = new IlvPalette();
IlvPalette p2 = new IlvPalette();
IlvPalette p3 = new IlvPalette();
IlvPaletteManager pm = new IlvPaletteManager();
pm.add(p1);
pm.add(p2);
pm.add(p3);
```

Palette manager events

When a palette is added or removed from an `IlvPaletteManager` instance, a `PaletteManagerEvent` is fired.

IBM® ILOG® supplies the following classes to track palette events:

♦ `PaletteManagerListener` - an interface for receiving `PaletteManagerEvent`s.

♦ `PaletteManagerAdapter` - the default implementation of `PaletteManagerListener`.

Organizing symbols in categories

The following example shows how to create symbols and organize them in palette categories. The example is divided into two steps:

1. Set the root category in a new palette.

```
IlvPalette palette = new IlvPalette();
palette.setName(¨myPalette¨);
palette.setDescription(¨multiple traffic signals¨);
palette.setPackageName("roadsigns/ ");
//Create and set the root category
IlvPaletteCategory root = new IlvPaletteCategory(rootID);
root.setName("symbols");
palette.setRoot(root);
```

2. Set a hierarchy of categories in a palette.

```
//The first categories under the root.
IlvPaletteCategory c1 = palette.addCategory(root, id1);
c1.setName("category 1");
c1.setLongDescription("The first category that contains…");

IlvPaletteCategory c2 = palette.addCategory(root, id2);
c2.setName("category 2");
c2.setLongDescription("The second category that contains…");

//Now create sub categories in the first ones.
IlvPaletteCategory c101 = new IlvPaletteCategory(id101);
c101.setName("category 101");
c1.add(c101);
IlvPaletteCategory c102 = new IlvPaletteCategory(id102);
c102.setName("category 102");
c1.add(c102);

IlvPaletteCategory c201 = new IlvPaletteCategory(id201);
c201.setName("category 201");
c2.add(c201);

//Finally, create some symbols in different categories.

IlvPaletteSymbol s100 = palette.addSymbol(c1, ids101);
s.setName("s100");
s.setCSSResourceName(css100);
s.setClassName(className100);
s.setIconResourceName(resdir + "icon.gif");
s.addResource(resourceURL1);
s.addResource(resourceURL2);


IlvPaletteSymbol s1010 = palette.addSymbol(c101, ids1010);
IlvPaletteSymbol s1011 = palette.addSymbol(c101, ids1011);

IlvPaletteSymbol s1020 = palette.addSymbol(c102, ids1020);
IlvPaletteSymbol s200 = palette.addSymbol(c2, ids200);
IlvPaletteSymbol s201 = palette.addSymbol(c2, ids201);
IlvPaletteSymbol s202 = palette.addSymbol(c2, ids202);
IlvPaletteSymbol s2010 = palette.addSymbol(c201, ids2010);
```

Note: The Symbol Editor helps you create symbols and palettes very easily and intuitively.

How to use a symbol in CSS

A symbol is used in a cascading style sheet (CSS) just like any other class. The symbol parameters are declared with the class in the CSS. The only difference is that the class declaration has another syntax:

```
class : @|symbolResource(<resource file>, <entry point>);
```

where:

- `<resource file>` is the path to a CSS file inside the palette, which must be reachable from the class path through `java.lang.ClassLoader.getResource()`.
- `<entry point>` is the entry point in the CSS file, usually `"Symbol"`.

Example:

```
node {
    class :
"@|symbolResource(ilog/views/palettes/shared/symbols/Rectangular.css,Symbol)
";
    name : "no-name" ;
}
```

If the palette is not in the class path, it can be specified as an extra argument of the `"class"` declaration:

```
class : @|symbolResource(<resource file>, <entry point>, <package name>,
           <content file>, <palette URL>);
```

where:

- `<package name>` is the symbol package name in the palette.
- `<content file>` is the path of the palette content, usually `palette.xml`.
- `<palette URL>` is the URL of the palette.

Example:

```
node {
    class :
 "@|symbolResource(ilog/views/palettes/shared/symbols/
  Rectangular.css,Symbol,Symbols.Basic.Rectangular,ilog/views/palettes/shared/

  palette.xml,file:data/palettes/jviews-palette-shared-symbols.jar)";
    name : "no-name" ;
}
```

The symbol reference is resolved through `getResource()` first. The full version of the `"class"` declaration, which contains the palette URL, is needed only for editing purposes when the palette is not yet in the class path. For deployed applications, it is recommended to rely on the class path version, which means adding all the palettes in use in the class path.

Using composite graphics

Shows how to use composite graphics in your diagrams.

In this section

What is a composite graphic object?
Provides a short overview of the Composite Graphics capability in IBM® ILOG® JViews Diagrammer and describes the Composite Graphics classes.

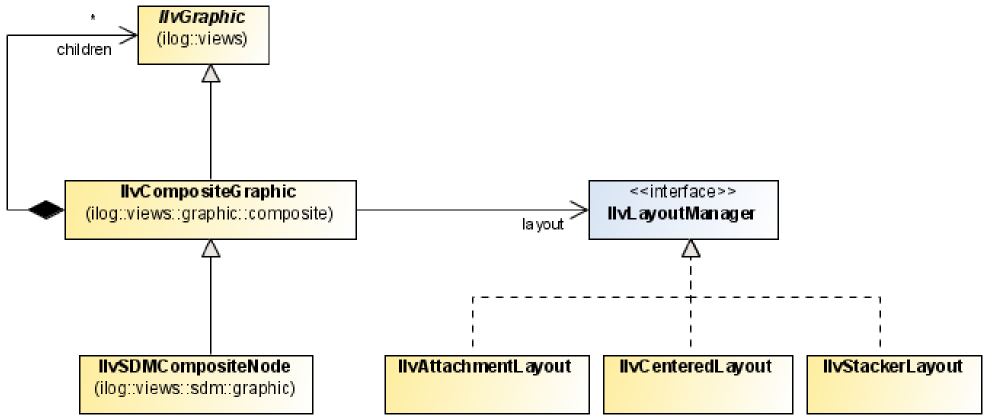
Building composite nodes in CSS
Shows how to build a composite node in the style sheet.

Building composite graphics in Java
Shows how to build a composite node using Java™ .

What is a composite graphic object?

The Composite Graphics capability consists of a set of classes that help you to combine simple graphic objects to build more complex graphic objects according to one of several possible layouts. You can modify a composite graphic object dynamically to add or remove child elements. This facility supports nested composite graphics in which a child element is itself a composite graphic object.

In IBM® ILOG® JViews Diagrammer, a Composite Graphic object is an instance of `IlvSDMCompositeNode`. This class is in the SDM package but it is a subclass of `IlvCompositeGraphic` which is a JViews Framework class, see the following class diagram.



Composite graphics classes available in IBM® ILOG® JViews Diagrammer

A composite graphic is made up of child graphics, which are positioned according to a composite layout by a Layout Manager, `IlvLayoutManager` .

Each child can be an `IlvGraphic` object or itself an `IlvCompositeGraphic` object.

There are three possible Layout classes:

♦ `IlvAttachmentLayout`

♦ `IlvCenteredLayout`

♦ `IlvStackerLayout`

These classes are in the package `ilog.views.graphic.composite.layout`.

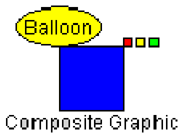
`IlvSDMCompositeNode` can have child graphics in different layers; this feature allows you to keep one child always on top.

In IBM® ILOG® JViews Diagrammer, you can create composite graphic objects through the style sheet, because the CSS for Java syntax has been extended to handle composite graphics concepts.

Building composite nodes in CSS

This section explains how to style nodes as composite graphics by building a composite graphic directly in the style sheet.

If you follow the steps given in this section, you will obtain the composite graphic shown in the following figure.



The example composite graphic

Note that this composite graphic itself contains composite graphics.

This composite graphic will be used to represent nodes of a specific user-defined type.

The data to be displayed comes from the XML file shown in the following code example.

The example XML data file

```
<?xml version="1.0" encoding="UTF-8"?>
<SDM>
  <MyNode >
    <property name="x">100</property>
    <property name="y">100</property>
    <property name="label">Composite Graphic</property>
    <property name="alarm">Balloon</property>
  </MyNode>
</SDM>
```

There is just one node defined, with user-defined type `MyNode`.

The x and y properties give the location of the node.

Each of the other properties will be styled as a child of a composite graphic.

To build composite nodes in CSS:

1. Enable Composite Graphics

   To make use of composite graphics in the style sheet, enable the Composite Graphics capability in the SDM rule, as follows:

   ```
   SDM {
      Composite:true;
   }
   ```

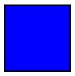
2. Customize some objects as composite graphics

   Add rules to customize objects with the user-defined type `MyNode` as composite graphics, as follows:

```
node.MyNode {
   class:'ilog.views.sdm.graphic.IlvSDMCompositeNode';
   children[0]:@+Rectangle;
}
Subobject#Rectangle {
   class:'ilog.views.graphic.IlvRectangle';
   width:40;
   height:40;
   fillOn:true;
   background:blue;
}
```

A composite graphic in IBM® ILOG® JViews Diagrammer is an instance of the class `IlvSDMCompositeNode`. It always has at least one child element. This first child has index 0 in the `children` array. In this example, the first child is created as an `IlvRectangle` object. It is defined with the ID `Rectangle` in a separate rule.

At this stage, the `MyNode` node would be displayed by the SDM engine as shown in the following figure.



A composite node with only the first child element

3. Add a decorated child

To display the label attribute of the `MyNode` object, add another child element to the `MyNode` rule as shown in the following code example.

```
node.MyNode {
   class:'ilog.views.sdm.graphic.IlvSDMCompositeNode';
   layout:@+AttachmentLayout;
   children[0]:@+Rectangle;
   children[1]:@+Label;
   constraints[1]:@=attachmentLabel;
}

...

Subobject#AttachmentLayout {
   class:'ilog.views.graphic.composite.layout.IlvAttachmentLayout';

}

Subobject#Label {
   class:'ilog.views.graphic.IlvText';
   label:@label;

}
```

```
Subobject#attachmentLabel {
   class:'ilog.views.graphic.composite.layout.IlvAttachmentConstraint';
   hotSpot:TopCenter;
   anchor:BottomCenter;
}
```

This second child has index 1 in the `children` array. In this example, the second child is created as an `IlvText` object. It is defined with the ID `Label` in a separate rule.

Once there is a further child, you need to specify which layout to use. In this case, the Attachment Layout is appropriate because it is the most general. You also need to add a constraint to specify how the second child is attached to the first. The index of the `constraints` array must correspond to the index of the `children` array.

For an Attachment Constraint, you must specify the following properties in a separate rule:

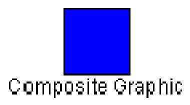
♦ `hotSpot`, which is the point on the subsequent child object by which it is attached

♦ `anchor`, which is the point on the first child to which the subsequent child is attached

At this stage, the `MyNode` node would be displayed by the SDM engine as shown in figure *A composite node with only the first child element*.

Note that an Attachment can be shared and so the CSS construct `@=` is used in the `MyNode` rule. Child graphics cannot be shared and so they are created with the CSS construct `@+`.

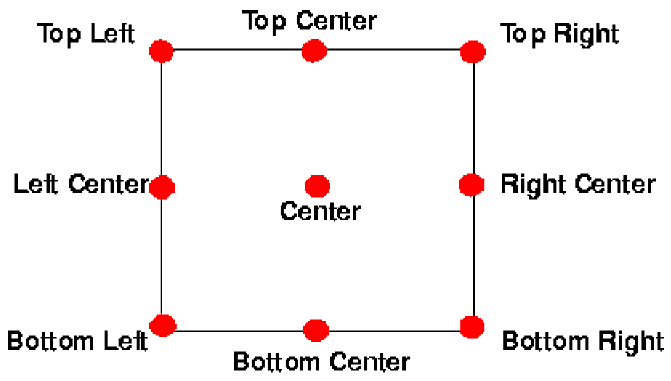
Warning: You must NOT use the @# construct to define composite graphics. Only the @+ and @= constructs are allowed. Using @# may lead to a wrong or inconsistent layout.



A composite node with two child elements

The top center of the second child (`IlvText` object) is anchored to the bottom center of the first child (`IlvRectangle` object).

The Attachment Layout provides nine possible attachment locations, as shown in the following figure.



Possible attachment locations

4. Add an Aligned Nested Composite Graphic

   To add the three little rectangles at the top right corner of the blue rectangle, you need to add a child which is itself a composite graphic.

   First define the three rectangles with IDs `Rect1`, `Rect2`, and `Rect3`, as shown in the following code example. These are the simple graphic objects which make up the nested composite graphic.

```
Subobject#Rect1 {
   class:'ilog.views.graphic.IlvRectangle';
   width:5;
   height:5;
   background:red;
   fillOn:true;
}
Subobject#Rect2 {
   class:'ilog.views.graphic.IlvRectangle';
   width:5;
   height:5;
   background:yellow;
   fillOn:true;
}
Subobject#Rect3 {
   class:'ilog.views.graphic.IlvRectangle';
   width:5;
   height:5;
   background:green;
   fillOn:true;
}
```

   Next add another child element to the `MyNode` rule as shown in the following code example.

```
node.MyNode {
```

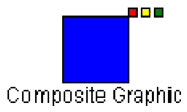
```
...
   children[2]:@+Rectangles;
...
   constraints[2]:@=attachmentRectangles;
}

...

Subobject#Rectangles {
   class:'ilog.views.sdm.graphic.IlvSDMCompositeNode';
   layout:@+StackerLayout;
   children[0]:@+Rect1;
   children[1]:@+Rect2;
   children[2]:@+Rect3;
}
Subobject#attachmentRectangles {
  class  : "ilog.views.graphic.composite.layout.IlvAttachmentConstraint"
 ;
   hotSpot:BottomLeft;
   anchor:TopRight;
}
Subobject#StackerLayout {
   class:'ilog.views.graphic.composite.layout.IlvStackerLayout';
}
```

This third child has index 2 in the `children` array. It is composite graphic, that is, an `IlvSDMCompositeNode`. object and it is defined with the ID `Rectangles` in a separate rule.

In the `Subobject#Rectangles` rule, you need to specify which layout to use for the three children. In this case, the Stacker Layout is appropriate because a Stacker Layout is used to align graphics, either horizontally or vertically. The graphics to be aligned horizontally in this example are the children of the composite node.

You also need to add a constraint to the `MyNode` rule to specify how the third child is attached to the first. In this example, the bottom left of the third child (`IlvSDMCompositeNode` object) is anchored to the top right of the first child (`IlvRectangle` object).

At this stage, the `MyNode` node would be displayed by the SDM engine as shown in the following figure.



A composite node with three child elements, one a composite

5. Add a Centered Nested Composite Graphic

   Finally, add an information balloon to display the `alarm` attribute.

To add the information balloon with its text, you need to add another child which is itself a composite graphic.

First define the ellipse shape and the label with IDs `Ellipse`, and `BalloonLabel`, as shown in the following code example. These are the simple graphic objects which make up the nested composite graphic.

```
Subobject#Ellipse {
   class:'ilog.views.graphic.IlvEllipse';
   background:yellow;
   fillOn:true;
}
Subobject#BalloonLabel {
   class:'ilog.views.graphic.IlvText';
   label:@alarm;

}
```

Next add another child element to the `MyNode` rule as shown in the following code example.

```
node.MyNode {
   ...
   children[3]:@+Balloon;
  ...
   constraints[3]:@=attachmentBalloon;
}
...
Subobject#Balloon {
   class:'ilog.views.sdm.graphic.IlvSDMCompositeNode';
   layout:@+CenteredLayout;
   children[0]:@+Ellipse;
   children[1]:@+BalloonLabel;

}
Subobject#CenteredLayout {
   class:'ilog.views.graphic.composite.layout.IlvCenteredLayout';
   insets:5,5,5,5;
}

Subobject#attachmentBalloon {
   class:'ilog.views.graphic.composite.layout.IlvAttachmentConstraint';
   hotSpot:BottomCenter;
   anchor:TopLeft;
}
```

This fourth child has index 3 in the `children` array. It is a composite graphic, that is, an `IlvSDMCompositeNode`. object and it is defined with the ID `Balloon` in a separate rule.

In the `Subobject#Balloon` rule, you need to specify which layout to use for the two children. In this case, the Centered Layout is appropriate because a Centered Layout is used to center one graphic on another. A Centered Layout displays an outer graphic (the child at index 0) around an inner graphic (the child at index 1). The four `insets` specify the pixel distances from each edge of the bounding box of the outer graphic. Note that the two child objects maintain their relative position when the composite graphic is resized.

You also need to add a constraint to the `MyNode` rule to specify how the fourth child is attached to the first. In this example, the bottom center of the fourth child (`IlvSDMCompositeNode` object) is anchored to the top left of the first child (`IlvRectangle` object).

At this stage, the `MyNode` node would be displayed by the SDM engine as shown in figure *The example composite graphic*, as required.

Building composite graphics in Java

You can also build composite graphics in Java™ code using the JViews Framework classes in the `ilog.views.graphic.composite` package.

To see the same example as the one in *Building composite nodes in CSS* but built with the JViews Framework classes instead, see Composite Graphics in *The Essential JViews Framework*.

Printing

Printing facilities are fully predefined in the class `IlvDiagrammerApplication` . The use of these facilities in an application is explained in Printing the diagram from an IlvDiagrammerApplication instance in *Using the Designer*.

If you require more sophisticated printing facilities, you can retrieve the view, the grapher, and the printing controller for the diagram component and customize the printing framework. Note that the view and grapher are available at the SDM engine level. Therefore, proceed as follows:

1. Retrieve the SDM engine for the diagram component. You do this with the method `getEngine()`:

```
myEngine = myDiagrammer.getEngine();
```

2. Retrieve the view and the grapher from the engine. You do this with the methods `getReferenceView()` and `getGrapher()`:

```
myView = myEngine.getReferenceView();
myGrapher = myEngine.getGrapher();
```

3. Retrieve the printing controller for the diagram component. You do this with the method `getPrintingController()`.

```
myDiagrammer.getPrintingController();
```

4. You can now customize the printing framework work as described in Printing framework for manager content in the *Advanced Features of JViews Framework*.

Using JViews products in Eclipse RCP applications

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic shows you how to display diagrams and dashboards embedded in an SWT window.

Installing the JViews runtime plugin

IBM® ILOG® JViews Diagrammer provides jar files in the form of a pre-packaged Eclipse plugin. The name of this package is `ilog.views.eclipse.diagrammer.runtime`.

In order to install the IBM® ILOG® JViews 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.

This installation also installs some examples. See Installing and using Eclipse samples for more information.

In your applications, you need the `ilog.views.eclipse.diagrammer.runtime` plugin and its dependencies:

- `ilog.views.eclipse.diagrammer.runtime`
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

Providing access to class loaders

Many services in JViews need to look up a resource. Since the classical way to provide access to resources is a classloader, JViews uses classloaders for this purpose. But in Eclipse/RCP applications, each plugin corresponds to a classloader, and the JViews classloader sees only its own resources, not the application resources. To fix this problem, you can register plugin classloaders with JViews through the IlvClassLoaderUtil.registerClassLoader function. Each resource lookup then considers the registered classloaders and, if the plugins are configured accordingly, also considers the dependencies of the registered classloaders.

The code for doing this is usually located in a plugin activator class. For example:

```
public class MyPluginActivator extends AbstractUIPlugin
{
    /**
     * This method is called upon plugin activation
     */
    public void start(BundleContext context) throws Exception {
      super.start(context);
      IlvClassLoaderUtil.registerClassLoader(getClass().getClassLoader());
    }

    /**
     * This method is called when the plugin is stopped
     */
    public void stop(BundleContext context) throws Exception {
      super.stop(context);
      IlvClassLoaderUtil.unregisterClassLoader(getClass().getClassLoader());
    }

  }
```

The overriding of stop() is necessary so that, when the plugin gets unloaded, JViews gets notified about the plugin that is going to stop and can drop references to its resources or instances of its classes. The activator plugin is usually also the place where IlvProductUtil.registerApplication is called. See section Before you start deploying an application for an example.

The bridge between AWT/Swing and SWT

The bridge between the AWT/Swing windowing system and the SWT windowing system consists of an IlvSwingControl class that encapsulates a Swing JComponent in an SWT widget. This class allows you to use IlvDiagrammer, IlvJScrollManagerView, or IlvJManagerViewPanel objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object:

```
Composite parent = ...;
IlvDiagrammer diagrammer = new IlvDiagrammer();
ControlSWTdiagrammer = new IlvSwingControl(parent, SWT.NONE, diagrammer);
```

At the JViews Framework level, the bridge between the AWT/Swing windowing system and the SWT windowing system consists of an IlvSwingControl class that encapsulates a Swing JComponent in an SWT widget. This class allows you to use IlvManager or IlvJManagerViewPanel objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object at the JViews Framework level:

```
Composite parent = ...;
IlvManagerView mgrView = ...;
```

```
IlvJManagerViewPanel jmgrView = new IlvJManagerViewPanel(mgrView);
ControlSWTview = new IlvSwingControl(parent, SWT.NONE, jmgrView);
```

Using `IlvSwingControl` instead of the native SWT_AWT class has the following benefits:

♦ Simplicity: it is easier to use, since you do not have to worry about the details of the `Component` hierarchy (see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*).

♦ Portability: `IlvSwingControl` also works on platforms that do not have SWT_AWT, like X11/Motif® and MacOS® X 10.4.

♦ Less flickering: on Linux®/Gtk, flickering is reduced.

♦ Popup menus: popup menus can be positioned on each `Component` inside the AWT component hierarchy. For details of components, see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*.

♦ Better size management: the size management between SWT and AWT (`LayoutManager`) is integrated.

♦ Focus: it provides a workaround for a focus problem on Microsoft® Windows® platforms.

Note: The `IlvSwingControl` bridge is not supported on all platforms. It is only supported on Windows, UNIX® with X11 (Linux, Solaris™, AIX®, HP-UX®), and MacOS X 10.4 or later.

The `IlvSwingControl` bridge does not support arbitrary JComponents. Essentially, components that provide text editing are not supported. See `IlvSwingControl` for a precise description of the limitations.

Threading modes

You can handle the SWT-Swing user interface events in one or two threads.

Note: Single-thread mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT `Dialogs`, Swing `JDialogs`, or modal `JInternalFrames` in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

♦ Two-thread mode

The SWT events are handled in the SWT event thread and AWT/Swing events are handled in the AWT/Swing event thread. This is the default mode.

You can switch between the two threads by using the SWT method `Display.asyncExec()` and the AWT method `EventQueue.invokeLater()`.

If your application uses this mode, you must be careful to:

- Make API calls on SWT widgets only in the SWT event thread. Otherwise, you will get `SWTExceptions` of type `ERROR_THREAD_INVALID_ACCESS`.
- Make API calls on JComponents, which include `IlvDiagrammer`, `IlvJScrollManagerView`, and `IlvJManagerViewPanel`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

  At the JViews Framework level, make API calls on JComponents, which include `IlvManager` and `IlvJManagerViewPanel`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

♦ Single-thread mode

In single-thread mode, SWT and AWT/Swing events are handled in the same thread.

Single-thread mode reduces the risk of producing deadlocks.

Enable this mode by calling `setAWTThreadRedirect` or `enableAWTThreadRedirect()` early during initialization.

The following example shows how to enable single-thread mode:

```
// Switch single-event-thread mode during a static initialization.
      static {
          IlvEventThreadUtil.enableAWTThreadRedirect();
      }
```

If you are using JComponents other than `IlvDiagrammer`, `IlvJScrollManagerView`, and `IlvJManagerViewPanel` in your application, your JComponents must use the method `isDispatchThread()` rather than *`EventQueue.isDispatchThread()`* or *`SwingUtilities.isEventDispatchThread()`* .

At the JViews Framework level, if you are using JComponents other than `IlvManager` and `IlvJManagerViewPanel` in your application, your JComponents must use the method `isDispatchThread()` rather than `EventQueue.isDispatchThread()` (see *http://java.sun.com/javase/6/docs/api/java/awt/EventQueue.html#isDispatchThread()*) or `SwingUtilities.isEventDispatchThread()` (see *http://java.sun.com/javase/6/docs/api/javax/swing/SwingUtilities.html#isEventDispatchThread()*.)

Performance enhancements

Provides hints to improve the performance of an application.

In this section

Global performance improvements
Explains how to improve the global performance of your application.

Optimizing the performance of JViews Framework
Describes how to optimize the performance of your Framework application.

Optimizing the performance of symbols
Shows how to improve the performance of your symbols.

Optimizing the performance of diagrams
Shows you how to improve the performance of your diagrams.

Optimizing the performance of dashboards
Explains how to improve the efficiency of your dashboards.

Optimizing the performance of Graph Layout
Describes how to optimize the performance of the graph layout algorithm used by your application.

Global performance improvements

IBM® ILOG® JViews is a comprehensive set of Java™ components, tools and libraries for creating diagram-based editing, visualization, supervision and monitoring tools.

To stop memory copy slowing down the Java Virtual Machine for you custom application, the initial heap size must be set to the amount of memory used when the application is running in its cruise state. This maximum heap size is computed either by using the API or through a profiling tool. A list of profiling tools can be found at *http://www.java-source.net/open-source/profilers*.

You set the initial and final heap size using the `-Xmx` and `-Xms` command line parameters for the JVM™ . For example, to set the minimum heap size to 130 megabytes and maximum heap size to 150 megabytes, using the following arguments:

```
java -Xmx150m -Xms130m -jar myApp.jar
```

If you open a dashboard diagram stored in binary format, you do not need to set the heap size of the virtual machine to be so large. For more information see the *Binary dashboard format*.

Optimizing the performance of JViews Framework

Describes how to optimize the performance of your Framework application.

In this section

Introduction
Briefly describes why the application performance would need to be optimized.

Configuring the Java Virtual Machine
Describes how to configure the Java™ Virtual Machine

Minimizing JAR files
Explains how to minimize the JAR files.

Choosing the best graphic object
Explains how to choose the best graphic object depending on the requirements of your application.

Drawing performance in the view
Describes how to speed up the drawing in the view

Events and listeners
Explains how to design events and listeners for better performance.

Interactors
Provides hints to improve the performance related to the use of interactors.

Antialiasing
Provides hints to improve the performance related to the antialiasing mode.

Transparency
Provides hints to improve the performance related to the transparency feature.

Printing
Provides hints to improve the performance related to the printing feature.

Introduction

If your application displays and manipulates only small data sets or graphs, you don't need to worry about performance optimizations. However, an application that handles large data sets must balance between memory, speed and feature set A rich feature set often means large memory and slow speed. Using caching mechanisms may increase the speed but needs more memory. If the application uses too much memory and at the same time reacts too slow for your large data set, then you should consider reducing the features of the data set. JViews Framework includes some features and programming techniques that increase the speed and reduce the memory at the same time, however, not all features can be applied to every application.

Configuring the Java Virtual Machine

IBM® ILOG® JViews allows you to create Java™ applications. As a rule of thumb, it is recommended to run the application with the latest release of the Java Virtual Machine. History has shown that past Java Virtual Machines JRE™ 1.3, JRE 1.4, and JRE 1.5 have increased the performance with each release. Therefore, the latest release is usually the fastest one.

To stop memory garbage collection that slows down the Java Virtual Machine for your custom application:

1. Set the initial heap size to the amount of memory used when the application is running in its cruise state.

   This maximum heap size is computed either by using the API or through a profiling tool. A list of profiling tools can be found at .

2. Set the initial and final heap size using the `-Xmx` and `-Xms` command line parameters for the JVM™ .

   For example, to set the minimum heap size to 130 megabytes and maximum heap size to 150 megabytes, use the following arguments: `java -Xmx150m -Xms130m -jar myApp.jar`

Minimizing JAR files

IBM® ILOG® JViews provides several large JAR files for your application. If your application is an applet, you probably want to reduce the size of the JAR file, because your applet does not require all classes contained in the JAR files.

1. Unpack the JAR file.
2. Remove the unused classes.
3. Create a new JAR file with only the remaining used classes.

   Various commercial tools can help you reduce the JAR files automatically:

   - DashO: *http://www.preemptive.com*
   - JShrink: *http://www.e-t.com/jshrink.html*
   - ProGuard: *http://proguard.sourceforge.net*
   - JoGa: *http://www.nq4.de/*

Choosing the best graphic object

The JViews Framework contains several predefined `IlvGraphic` subclasses. They have different features and memory/speed characteristics. However, in some cases the best choice is to program a custom graphic object for your application, since the predefined graphic objects has a large generic set of features that may not be needed in your application.

Text objects

The classes `IlvLabel`, `IlvZoomableLabel` and `IlvText` can be used to display text. `IlvLabel` is the fastest and lightest text object, but it is not zoomable or rotatable and does not support multiple lines of text. If has very few features. `IlvZoomableLabel` is zoomable, rotatable and supports multiple lines of text. It paints the glyphs with gradient or texture paints. Hence it paints relatively slow and uses plenty of memory. `IlvText` is in many cases much faster than `IlvZoomableLabel` but it only supports uniform colors to draw the text glyphs. It supports attributed text (underline, strike-through, and so on) and multiline-wrapping. In most cases, `IlvText` is better than `IlvZoomableLabel`, although it uses plenty of memory.

Class name	Draw	Memory usage	Zoomable Rotatable	Glyph Fill	Background Rectangle Fill & Stroke	Multiline	Advanced features (attributed text, line wrapping)
`IlvLabel`	Fastest	Very light	No	Color	No	No	No
`IlvZoomableLabel`	Slow	Heavy	Yes	Paint	Yes	Yes	No
`IlvText`	Medium Fast	Heavy	Yes	Color	Yes	Yes	Yes

The drawing speed of `IlvLabel`, `IlvText` and `IlvZoomableLable` is faster if antialiasing is disabled. If the speed is critical, then it is better to switch the antialiasing off:

```
label.setAntialiasing(false);
```

`IlvText` and `IlvZoomableLable` are designed as general purpose classes, hence they have many features and require more memory. If memory is critical and there are many text or label objects, and if only few features of these objects need to be customizable, then it is better to implement a lightweight text class that contains only these few features. The implementation of the lightweight text class can delegate to a temporary allocated `IlvText` on the fly. An example of this mechanism is shown in the code example **<installdir>/jviews-framework86/codefragments/lighttext**.

Shape objects

`IlvGeneralPath` is a generic object that can be used to draw any filled or stroked shape. However, some specialized classes for particular shapes exist and are much faster and use less memory. These specialized classes have less features than `IlvGeneralPath`, for instance they do not support custom strokes.

- `IlvLine` to draw a straight line,

- `IlvPolyline` to draw a sequence of straight line segments,
- `IlvSpline` to draw a sequence of spline segments,
- `IlvRectangle` to draw a filled or stroked rectangle, optionally with rounded corners,
- `IlvPolygon` to draw a filled or stroked polygon,
- `IlvArc` to draw a partial circle, annulus or pie,
- `IlvEllipse` to draw a filled or stroked full circle or ellipse.

If these classes fit the need of your application, we recommend to use them instead of `IlvGeneralPath`.

Icons and images

IBM® ILOG® JViews allows you to load GIF, JPG, PNG images, and, with limitations, also SVG and DXF images. GIF, JPG and PNG are bitmap formats that are loaded into `IlvIcon`. SVG and DXF are vector formats that can be loaded into an `IlvGraphicSet` (for example, in JViews Diagrammer via `get`. When zooming bitmaps, you loose drawing quality. For instance, a magnified bitmap may appear blurred. This effect does not occur with vector formats; they can be zoomed freely. However, vector formats use more memory and may be drawn slower than bitmaps. Assume you have 10000 nodes displaying an image consisting of 3 rectangles and 4 circles. If you load this image as bitmap, it is shared among all 10000 nodes automatically. If you load the image as SVG or DFX file, you have 10000 instances of `IlvGraphicSet`, each containing 3 rectangles and 4 circles: overall 30000 rectangles and 40000 circles. Therefore, implementing these nodes with SVG or DXF files will require much more memory then implementing them with bitmap images. Note that many external SVG tools produce non-optimized SVG code that can be very complex. These tools are suited to create an image that is used one time, but they may not be suitable to produce images that are replicated 10000 times in 10000 nodes. Therefore, if SVG is used inside a large number of nodes, we recommend to fine-tune and to simplify the SVG, or to use bitmap images instead. To overcome the quality loss in zoomed bitmaps, `IlvIcon` has a high quality rendering mode (`setHighQualityRendering`). However, this drawing mode is very time consuming. If you need to display many icons, it is recommended to disable the high quality rendering mode to improve the performance.

Note: For JViews Diagrammer `IlvGeneralNode` uses internally `IlvIcon` when needed, and it sets them by default in slow high quality rendering mode. To disable this effect, call

```
IlvGeneralNode.High_Quality_Icons = false;
```

before the first `IlvGeneralNode` is allocated, that is, at the beginning of your application.

Links

IBM® ILOG® JViews offers many link classes to connect nodes in a diagram. The simplest link class is `IlvLinkImage`, the most complex, feature-rich and slow link class is

IlvGeneralLink from JViews Diagrammer. In terms of features, memory and speed, the following equation holds:

```
IlvLinkImage << IlvPolylineLinkImage < IlvEnhancedPolylineLinkImage <<
IlvSimpleLink (Diagrammer) << IlvGeneralLink (Diagrammer)
```

If your application uses many links, it is recommended to use the lightest possible link that has the features that you need:

♦ IlvLinkImage is suitable as straight link with simple arrowhead and uniform color. Do not use it if you want to apply the automatic graph layout algorithms that reshape links available with JViews Diagrammer.

♦ IlvPolylineLinkImage is a link with bend points. It has a simple arrowhead, a uniform color, and can be used in all graph layouts.

♦ IlvEnhancedPolylineLinkImage has all features of IlvPolylineLinkImage. Additionally, it can display a simple backward arrowhead, has an orthogonal and bundle mode and can display tunnel/bridge crossings.

♦ IlvSimpleLink is an IlvEnhancedPolylineLinkImage that supports property notifications that are sometimes necessary in SDM.

♦ IlvGeneralLink has all features of IlvSimpleLink, but enables arbitrary decorations and labels, and various nonuniform colors.

♦ IlvCompositeLink does not fit into this feature hierarchy as it has not all the features of the previously mentioned links. It can be composed by using one of the previously mentioned links. In terms of memory, IlvCompositeLink is the most heavy link. In terms of speed, it depends on the part it is composed of.

If you use the IlvClippingLinkConnector and want to improve the speed, always use a subclass of IlvPolicyAwareLinkImage (for example, IlvEnhancedPolylineLinkImage, IlvSimpleLink, or IlvGeneralLink) because the link connector can cache the clip points for these classes but not for the other link classes. If the clip points are not cached, they must be recalculated on the fly when drawing the link, and this can be slow.

If the quadtree is enabled, then zoomable links are drawn and manipulated faster than nonzoomable links (see zoomable). If your application uses many links, it is recommended to use zoomable links. Note that IlvGeneralLink is not zoomable, and other links are zoomable only if their end nodes are zoomable. If your application needs to display a huge number of links, it is recommended to use only nodes and link that are zoomable. In particular, if you use tunnel/bridge crossing shapes (of IlvEnhancedPolylineLinkImage, IlvSimpleLink, and IlvGeneralLink) you should use zoomable links because in these cases the crossings of links must be calculated on the fly. The quadtree helps to determine these crossings, but this works only if the quadtree is enabled and the links are zoomable. If the quadtree must be disabled or if you must use a large number of nonzoomable links, it is recommended to disable the tunnel/bridge crossing shapes.

Nested managers and graphers

When managers (IlvManager) or graphers (IlvGrapher) are nested in a hierarchy, the larger the depth of this hierarchy, the smaller the performance. Avoid nested graphs with an extremely high (hundreds or more) nesting depth.

When you have nested graphers, you can have intergraph links, that is, links with end nodes in different graphers. Intergraph links are often less efficient than normal links with end nodes in the same grapher, since when calculating the geometry of the link, the different transformers of the different graphers need to be taken into account. You can optimize the speed by avoiding intergraph links and by structuring your diagram in a way so that only normal links are necessary.

Implementing a custom graphic object

The most effective way to optimize speed and memory usage is to implement a custom graphic class that contains exactly the features that are needed by your application. This reduces the memory, because your custom graphic needs to store only those parameters that are variable in your application. As opposed to the generic predefined graphic classes, all other facilities (such as colors, paints, strokes, and so on) can be hard-wired if they do not need to be variable in your application. Since hard-wired facilities don't need to be stored per object, they need no memory.

There are two strategies to implement a graphic object that has multiple states and that need to be drawn in a different way:

- compose an `IlvGraphicSet` or `IlvCompositeGraphic` with multiple subobjects such as `IlvRectangle`, `IlvEllipse`, `IlvText` and rearrange these subobjects depending on the state. It may make subobjects visible or invisible depending on the state.
- create a custom graphic that simply displays different bitmap images depending on the state. There is only one bitmap image needed per state.

If you have many objects but only few states, the second strategy uses in many cases less memory because the images can be shared among all the objects, while the first strategy requires individual subobjects for each main object. The predefined class `IlvMultipleIcon` helps to implement the second strategy.

When implementing a custom graphic, you should optimize the following methods for speed as they are used quite frequently:

- `boundingBox`
- `contains`
- `draw`
- `zoomable`

If heavy computations are needed inside `boundingBox(IlvTransformer)`, consider implementing a cache that maps the transformer to the bounding box. The bounding box cache must be invalidated whenever the real bounds of the object change, for instance when any object parameter changes by affecting the size or position of the object. If the transformer is not in the cache, the bounding box must be recomputed. However, if the transformer is already in the cache, the bounding box can be retrieved from the cache, which speeds up the operation.

For more information on how to implement custom graphic objects see Creating a new graphic object class. The sample **<installdir>/jviews-framework86/samples/graphics** shows how to create a custom graphic object. The code example **<installdir>/jviews-framework86/codefragments/lighttext** shows how to create a light custom text object.

Drawing performance in the view

Sometimes the drawing performance is still not sufficient despite a careful selection of object classes. This may in particular happen if you display a huge number of objects in multiple views. Since after each change, each view must be drawn, the drawing performance may be a bottleneck. In general, the best way to improve the drawing performance is to display only one view. However, a very common scenario is to have a main view and an overview.

Redraw sessions

Whenever any object change, the view must be redrawn, to display the objects correctly. If many objects change at the same time, it is better to perform only one redraw at the end instead of many redraw after each object. This can be achieved with a redraw session:

```
manager.initReDraws();
   try {
      ...many objects change, but redraw is supressed now ...
   } finally {
     // redraw the views
     manager.reDrawViews();
   }
```

For details, see Optimizing drawing tasks.

Buffering the view

You can use one of the predefined view buffering techniques. A buffered view is drawn faster than each individual object contained in the view. On the other hand, updating the buffer may be slow, therefore buffering is better if objects change only rarely.

♦ Double buffering (`setDoubleBuffering`) should be used to buffer the entire view. It improves scrolling performance in optimized view translation mode (`setOptimizedTranslation`). For details, see Managing double buffering.

♦ Triple buffering (`setTripleBufferedLayerCount`) can be used to buffer the first few layers of a view. This is useful if moving objects are drawn in front of a static background. The layers 0 to n may contain the background, and the layers n+1, n+2 and so on contain the moving objects. It improves the overall drawing performance since the layer 1 to n are quickly copied from the buffer bitmap. However, the memory consumption increases by one additional in-memory bitmap of the size of the view. For details, see Triple buffering layers.

♦ Layer caches (`setLayerCached`) can be used for the same reason, if the static layers that never change are not in a continuous range from 1 to n. Each cached layer requires an additional in-memory bitmap. It can be combined with triple buffering: if the static layers are 0, 1, 2, 3, 5, and 7, then use triple buffering for layer 0-3, and a layer cache for layer 5 and layer 7. If you use individual layer caches for layer 0-3, it would require more bitmap buffers than when combining these layers by triple buffering. For details, see Caching layers.

Optimizing the overview

The overview is a secondary view that is typically be used to navigate in the main view. The overview usually displays the entire manager in a demagnified way. The overview is a performance bottleneck because:

- it needs to all many objects
- it needs to fit the view whenever the manager contents changes

When displaying a huge number of objects, the overview may consume so much time that the application becomes unresponsive. On the other hand, the display of the overview in many situations does not need to be very precise. This can be used to improve the performance:

- the overview does not need to be refreshed as often as the main view. Use the repaint skip mechanism (`setRepaintSkipThreshold`) to refresh the overview only every couple of seconds.
- you may consider to disable blinking for the overview (`setBlinkingMode`). Blinking objects require periodical drawing of the view, which is the more time consuming the more objects are displayed.
- if you implement your own graphic objects, then modify the draw method (`draw`) to draw only small rectangles when the view is demagnified. Drawing rectangles is faster than drawing nodes in full detail. At large demagnification, these details are anyway not visible. By checking the zoom factor of the input transformer of the draw method, you can easily decide whether the object needs to be drawn in full detail or only approximately as rectangle. An example of this technique can be found in the code example **<installdir>/jviews-framework86/codefragments/lighttext**.

Blinking

IBM® ILOG® JViews supports blinking objects, blinking colors and blinking actions (see Blinking of graphic objects). An object that blinks needs to periodically be drawn. Here are several hints to improve the performance:

- Do not use any kind of blinking if it is not necessary.
- If you have blinking objects, ensure that all objects use the same blinking timing. In this case, all objects are drawn at the same time, which is more efficient than when different objects are periodically drawn at different times.
- Enable blinking only in the main view. Disable blinking for secondary views such as the overview. See `setBlinkingMode`.
- Blinking actions can modify objects periodically in an arbitrary way, which can be very inefficient. Avoid blinking actions that modify the bounding box of the graphic objects.
- Consider using the `IlvExpensiveDrawingRepaintManager` in combination with blinking. Blinking periodically repaints regions of the view. The default `RepaintManager` has a simple strategy to combine regions that need repaint. The `IlvExpensiveDrawingRepaintManager` has an advanced strategy that is optimized in case the repaint of individual graphic objects is very expensive.

Grid and background pattern

When the grid is enabled in a manager view, it consumes a lot of graphic resources. To improve performance do not enable the grid in your view. You disable the grid by calling `setGrid`.

Views can also have background pattern. Similar to the grid, they consume a lot of graphic resources. You disable background pattern by calling `setBackgroundPatternLocation`.

The drawing of the view is most efficient if the background of the view is drawn just with a uniform color, without grid or background pattern.

Events and listeners

IBM® ILOG® JViews supports various types of events and allows applications to add listeners to react on events. In many cases, these custom event listeners are the performance bottleneck, not the IBM® ILOG® JViews internal code. Therefore it is necessary to design the listener code carefully and to avoid to install too many listeners.

Adjusting sessions

Most IBM® ILOG® JViews events support an adjusting flag: large changes in the data may trigger a sequence of events, and the adjusting flag indicates that all events belong to the same sequence. This allows you to implement listeners so that inexpensive operations are done for each single event, and expensive operations are only done at the end of the sequence of events. All IBM® ILOG® JViews internal listeners are designed in this manner, and it is recommended that you use the same design principle for your own listeners if possible.

If your application performs large changes in the manager, then we recommend to use adjusting sessions in the following way:

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

All events fired inside this adjusting session have the adjusting flag enabled, hence all listeners supporting the adjusting flag will be executed in optimized way. For more information, see Listener for the content of the manager.

If your application selects or deselects a large set of objects, use selection adjusting sessions in a similar way:

```
manager.setSelectionAdjusting(true);
try {
  ... select or deselect a lot of objects...
} finally {
  manager.setSelectionAdjusting(false);
}
```

Note: It is very common to combine adjusting sessions and redraw sessions:

```
manager.setContentsAdjusting(true);
manager.initReDraws();
try {
  ... add, remove, move a lot of objects...
} finally {
  manager.setContentsAdjusting(false);
  manager.reDrawViews();
}
```

If you use JViews Diagrammer, you don't need to use the low level API `IlvManager.setContentsAdjusting` but should use the higher-level API instead: `IlvDiagrammer.setAdjusting` and `IlvSDMEngine.setAdjusting`. These higher level API will automatically manipulate the contents adjusting and selection adjusting flag of the manager.

Implementing listeners for adjusting sessions

If you need to implement an event listener, it is recommended to perform quick tasks inside adjusting sessions and slow tasks only at the end of adjusting sessions. This ensures that the slow tasks are not performed too often and don't slow down the entire application. Here is a typical scheme for a listener:

```
listener = new ManagerContentChangedListener() {
  int numAdditions = 0;
  int numRemovals = 0;
  public void contentsChanged(ManagerContentChangedEvent evt) {
    if (evt.isAdjusting()) {
      // inside an event series: quick operation, e.g. updating a counter
      switch (evt.getType()) {
        case ManagerContentChangedEvent.OBJECT_ADDED:
          numAdditions++;
          break;
        case ManagerContentChangedEvent.OBJECT_REMOVED:
          numRemovals++;
          break;
      }
    } else (evt.getType() == ManagerContentChangedEvent.ADJUSTMENT_END) {
      // at the end of the series: can do a slow operation
      reorganizeSomeData(numAdditions, numRemovals);
      numAdditions = numRemovals = 0;
    } else {
     // outside a series of events, need to do the full operation immediately

      switch (evt.getType()) {
        case ManagerContentChangedEvent.OBJECT_ADDED:
          reorganizeSomeData(1, 0);
          break;
        case ManagerContentChangedEvent.OBJECT_REMOVED:
          reorganizeSomeData(0, 1);
          break;
      }
    }
  }
```

Dispatch listeners

If you implement your own graphic object, then sometimes you want to perform some code when the graphic object itself is selected. You could register the graphic object itself as a selection listener, but when you add 1000 objects of this kind to a manager, the manager will have 1000 selection listeners. In this case it is more efficient to use a dispatch listener: one listener for all 1000 objects, that dispatches the event to the selected object. IBM® ILOG® JViews contains an abstract auxiliary class `IlvManagerSelectionDispatcher` that is suitable to implement such a dispatch listener.

Interactors

Opaque interaction mode

The select interactor (`IlvSelectInteractor`) and some other interactors that are able to move or reshape objects (for example, `IlvPolyPointsEdition`, `IlvReshapeSelection`, `IlvMoveRectangleInteractor` and its subclasses) have an opaque mode. The opaque mode means that the users sees the full graphic object while it moves or changes. If the opaque mode is switched off, the user sees only a ghost drawing of the object until the interaction is finished (for example, until the user releases the mouse button when dragging). Drawing ghosts is more efficient than drawing opaque objects, in particular if a large number of objects moves. To improve performance, call `setOpaqueMove(false)` or `setOpaqueMode (false)` on any interactor that supports opaque modes.

Hit-test

A very frequent operation is to determine which object is located at a given position. It is used by the select interactor to find the selected object. However, sometimes it is also used during every mouse movement. One example is when graphic objects have tooltips. The tooltip manager must determine during mouse movements which object is under the mouse pointer, in order to display the correct tooltip. Another example is when the objects have object interactors, and the system must determine which object interactor should receive the events.

The quadtree is a data structure to optimize the hit-test. Each manager layer can have a quadtree. The quadtree is enabled by default in optimized mode, and an application that contains a large number of graphic objects should not disable the quadtree nor switch to unoptimized mode.

The quadtree optimizes the hit-test only for zoomable objects, that is, `zoomable` returns true (see Zoomable and nonzoomable objects). If your application uses a large number of objects, we strongly recommend that you use zoomable objects. The hit test for nonzoomable objects is considerably slower than for zoomable objects.

Tooltips

Tooltips are handled by the `IlvToolTipManager` central manager. When a view is registered at the tooltip manager, the tooltip manager analyses for every mouse movement which tooltip must appear. This can slow down all mouse movements. If you use many objects and have nonzoomable objects or have the quadtree disabled, the mouse movement may become too slow. In this case, try avoid using tooltips. The following code example shows how to do this on a specific manager view:

```
javax.swing.SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        IlvToolTipManager.unregisterView(dashDiag.getView());
      }
    });
```

Antialiasing

Antialiasing changes the pixels along the edges of a line into varying shades of gray or in-between color in order to make the edge appear smoother. In JViews Framework, antialiasing is available for graphic objects containing text, such as labels and scales. Disable antialiasing to improve performance at the cost of a small reduction in presentation level. This can be done by calling `setAntialiasing(false)` in the following classes and their subclasses

- `IlvReliefLabel`
- `IlvScale`
- `IlvShadowLabel`
- `IlvLabel`
- `IlvTextPath`
- `IlvFilledLabel`
- `IlvZoomableLabel`
- `IlvText`
- `IlvCircularScale`
- `IlvRectangularScale`
- `IlvManagerView`

Transparency

Manager layers can be drawn transparent. The following classes (and their subclasses) support transparency per object.

- `IlvIcon`
- `IlvGeneralPath`
- `IlvGraphicSet`
- `IlvFullZoomingGraphic`
- `IlvHalfZoomingGraphic`
- `IlvEnhancedPolylineLinkImage`

Transparency has two potential problems:

- The drawing speed is considerably slower, therefore transparency should be avoided if there are plenty of objects.
- The spool size when printing the contents of a view may become very large.

If these problems affect your application, we recommend to avoid transparency. If the drawing speed is fast enough and only the large spool size causes problems, one possible solution is to disable transparency only during printing. In this case, the printout might differ from the display in the view, but the spool size remains small. If you cannot disable transparency but only very few objects are transparent, consider drawing the transparent region into an opaque offscreen buffer and print the opaque buffer instead of the transparent objects to reduce the spool size. Refer to *Printing* for further performance hints related to printing.

Printing

IBM® ILOG® JViews printing capabilities relies on the printing mechanism of the Sun™ JVM™ . In some situations, the Sun JVM generates big spool files that reduce printing performance drastically. (The problem was identified and registered as bug 4314221 in the Sun bugs database). Large printer spool sizes are usually the result of the Java 2D™ printing API using the raster pipeline instead of the default PDL pipeline. The printing API makes this switch automatically if any rendering on the page requires bitmap operations. This includes transparency and gradient paints. Therefore, the developer should avoid the following rendering features to ensure that the default PDL printing pipeline is used and spool sizes are reduced:

♦ Avoid transparency: draw only shapes and text with no alpha component. Do not draw images that have an alpha channel.

♦ Avoid gradients and bitmap paints: use only solid color paints.

JavaSoft™ has documented additional tips and ideas for optimizing the printing in:

♦ The JavaSoft Java2D tutorial at *http://java.sun.com/docs/books/tutorial/2d/printing/index.html*.

♦ The Java2D FAQ at *http://java.sun.com/products/java-media/2D/forDevelopers/java2dfaq.html*.

Some of these optimizations are:

♦ Do not use a PCL or a PostScript printer. If you have to use transparencies or gradients, try to print your document to a printer which is not a PCL or a Post Script?.

♦ Reduce the resolution of the printer, for example, use 300 DPI instead of 600 DPI.

♦ If you must use alpha, use this technique to avoid full-page rasterization:

   1. Draw the non-opaque colors into an opaque off-screen image.
   2. Redraw the image with alpha into an opaque image.
   3. Draw the opaque image into a PrinterGraphics.

   For example, in the case of a transparent bitmap you will have better performance if the actual bitmap drawn on the printer is an opaque bitmap built from the background and the transparent original bitmap (that is, build the transparent result before hand and send that result to the printer).

Optimizing the performance of symbols

Light symbols use the minimum amount of objects, parameters, conditions and interactors necessary to display an understandable message. Use the following guidelines to create symbols with the smallest memory footprint:

- ♦ Use as few objects as possible.
- ♦ Use a static background.
- ♦ Use light (PNG) images.
- ♦ Use the visibility option.
- ♦ Avoid transparency.
- ♦ Avoid complicated gradients.
- ♦ Avoid thick lines.
- ♦ Turn antialiasing off.
- ♦ Turn fraction metrics off.

Rendering operations take up a lot of CPU time. The fastest and lightest symbols are made by changing the visibility setting for a collection of PNG images.

Optimizing the performance of diagrams

Shows you how to improve the performance of your diagrams.

In this section

The Overview pane
Describes the Overview pane and its use in optimizing diagrams.

Grids in a diagram
Describes the use of grids in a diagram.

Rendering Done mode
Describes the Rendering Done mode in the SDM engine.

Composite renderer
Shows how to disable composite rendering in the SDM engine.

Load on demand
Shows how to set the load-on-demand mode in the SDM engine.

Content on demand
Shows how to set the content-on-demand mode in the SDM engine.

Adjusting modes
Shows how to use adjusting modes in the SDM engine.

Detail level
Shows how to set the levels of detail for displaying nodes and links in the SDM engine.

Better SDM/Java transition
Shows how to disconnect SDM to improve performance.

The Overview pane

Each time a change happens in a diagram, only the area in the diagram that has been changed is refreshed. However, the whole diagram in the Overview pane has to be repainted, causing a large performance reduction. The overview pane is represented by an `IlvDiagrammerOverview` object. This pane is to be used when you design your Diagrammer or Dashboard application. Using the Overview pane in a run time application can reduce performance. This is particularly true when you use dynamic symbols.

Grids in a diagram

When the grid is enabled in JViews Diagrammer applications, it consumes a lot of graphic resources. To improve performance do not enable the grid in your diagram. In the Designer GUI, the grid can be enabled or disabled using the View menu. You disable the grid by calling clearing the grid in the `IlvManagerView`. The following code example shows how to clear the grid:

```
IlvSDMEngine e = dashDiag.getEngine();
e.getReferenceView().setGrid(null);
```

Rendering Done mode

In the SDM engine, the rendering process is controlled by a CSS style sheet, which lets you tell the SDM engine how you want each particular kind of data object to be displayed in the grapher. When the data model is loaded, the SDM engine explores it and creates graphic objects representing the nodes and links defined by the data model in the grapher. When the state of an object in the data model changes, the SDM engine updates the graphic object representing the modified data object. The object state may change due to an external application event or after a direct edit of an object property by the user. A property is a named characteristic of a graphic object to which you can assign values.

To improve performance while changing a model property, disable rendering done mode. This is especially beneficial for renderers that perform post-processing; such as link layout.

The following code example shows how to disable rendering done mode in the SDM engine.

```
IlvSDMEngine e = dashDiag.getEngine();
int old = e.getRenderingDoneMode();
try {
  e.setRenderingDoneMode(e.NEVER);
  // modify model here, without set adjusting
  ...
} finally {
  e.setRenderingDoneMode(old);
}
```

Note: Setting rendering done mode to `IlvSDMEngine.NEVER` does not prevent the graphic from being customized.

Composite renderer

Composite rendering manages interaction and smart link connections, that is, the link ports, and the link connection rectangle for composite graphic and symbols. When dealing with basic non-interactive symbols, disabling composite rendering relieves the renderer toolchain and improves performance.

To disable composite rendering, set the `Composite` option in the SDM section of the initial stylesheet to false.

```
SDM {
   Map : "false" ;
   LinkLayout : "true" ;
   Composite : "false" ;
   GraphLayout : "false" ;
   HalfZooming : "false" ;
}
```

Load on demand

Using load on demand mode in the `IlvSubGraphRenderer` class, subnodes of a collapsed subgraph are created only when the subgraph is first expanded. This technique usually gives better startup times, at the expense of a slower expansion. The following code example shows how to set load-on-demand mode.

```
IlvDiagrammer diagrammer;

...

IlvSDMEngine engine = diagrammer.getEngine();
IlvSubGraphRenderer renderer = (IlvSubGraphRenderer)
  IlvRendererUtil.getRenderer(engine, IlvRendererUtil.SubGraph);
renderer.setLoadOnDemand(true);
```

Content on demand

The content on demand feature allows an SDM model to delay the loading of its object content in order to save resources. This assumes that empty model objects have low memory and time footprints, and are rendered with cheap graphic objects. Content on demand allows you to fill them when required and empty them when they are no longer needed.

The classes of the content on demand feature are located in the package `ilog.views.sdm.modeltools`. The entry point is the class `IlvContentController`.

Typically, content on demand is associated with a zoom listener. This allows the supervision of a map that contains a huge number of empty objects. Starting at a certain threshold, when the user zooms in, only the visible objects are filled. Objects outside the area of interest remain unloaded.

A content on demand sample is available in **<installdir>/jviews-diagrammer86/samples/diagrammer/content-on-demand/index.html**. For more information, see *Content on demand*.

Adjusting modes

IBM® ILOG® JViews Diagrammer applications use the SDM to calculate changes in the model. By default, after changes are calculated and validated in the SDM, the graphic objects effected by these changes are recalculated.

To render this process in the `IlvSDMEngine` more efficient, call `IlvSDMEngine.setAdjustingMode(true)`. The SDM will then process all the modifications at once. You open an adjusting sequence by calling `IlvSDMEngine.setAdjustingMode(true)`. To close an adjusting sequence, call `IlvSDMEngine.setAdjustingMode(false)`.

Alternatively, you can close the adjusting sequence using `IlvSDMEngine.clearAdjusting()`. This method also validates the model changes, but prevents SDM from processing the modifications. Use it only if you are sure the model changes do not impact the display in any way.

The following code example shows how to use adjusting modes:

```
IlvSDMEngine engine = ... ;

try {
  engine.setAdjusting(true);

  // perform changes on diag model, i.e. engine.getModel()

} finally {
  // commit changes
  engine.setAdjusting(false);
  // or commit silently:
  //engine.clearAdjusting();
}
```

Detail level

Style rules allow you to define three different levels of detail for displaying the nodes and links in a diagram: high (default), medium, low.

For example, you could define the following information to be displayed for each detail level:

- level: the nodes have a fill and a label
- medium level: the nodes have no fill but they have a label
- low level: the nodes have no fill and no label

A low level of detail allows you to speed the rendering of the diagram. The following code example shows how to set the level of detail to low.

```
IlvDiagrammer diagrammer;

...

IlvSDMEngine engine = diagrammer.getEngine();
engine.setDetailLevel(engine.LOW_DETAIL_LEVEL);;
...
```

Better SDM/Java transition

SDM manages the graphic objects according to the model state and style specification. To improve the performance or to execute tasks that are outside the SDM scope, you may need to tackle the grapher directly. In most cases, you will need to disconnect SDM so that it does not interfere and undo your actions.

To disconnect SDM, invoke the following method:

```
IlvSDMEngine.plugAllListeners(false);
```

This call will remove all SDM listeners set on the model, the view, and the manager. Without listeners, SDM does not react to modifications and you can safely perform actions on graphic objects and on the SDM model. The mapping between the SDM model and the graphic objects still works (`IlvSDMEngine.getGraphic()`) as well as most of the other SDM utility functions.

You can reconnect the SDM engine using `IlvSDMEngine.plugAllListeners(true)`, but be aware that the engine will not catch up automatically the modifications that occurred while it was disconnected. To synchronize again the view with the model, each modified object should be refreshed manually.

For a finer control over SDM listeners, there are specific methods for each type of listener. See

♦ `plugManagerListener`

♦ `plugViewListener`

♦ `plugSelectionListener`

♦ `plugModelListener`

Optimizing the performance of dashboards

Explains how to improve the efficiency of your dashboards.

In this section

The Overview pane
Describes the Overview pane in the Dashboard Editor.

Binary dashboard format
Explains the two dashboard file formats.

The Overview pane

The Overview pane in Dashboard Editor displays all symbols and background objects present in the current dashboard diagram. Each time a change happens in a dashboard diagram, only the area in the diagram that has been changed is refreshed. However, the whole diagram in the Overview pane has to be repainted, causing a large performance reduction.

The overview pane is represented by an `IlvDiagrammerOverview` object. To improve performance, do not add an `IlvDiagrammerOverview` in your Dashboard Editor application.

Note: In the DashboardEditor sample application, the Overview pane is added when you call `DashboardEditor.createRightArea()`.

Binary dashboard format

Dashboard Editor supports two different dashboard file formats:

- XML - stored in `.idbd` files.
- Binary - stored in `.idbin` files.

A dashboard in XML format is 8 times the size of a binary format file and takes 5 times as long to load. Using binary format decreases the amount of parsing and data conversion to be performed by Dashboard Editor.

A binary file is read into Dashboard Editor without using an XML parser or a document object model, which reduces memory use significantly.

Optimizing the performance of Graph Layout

Describes how to optimize the performance of the graph layout algorithm used by your application.

In this section

Introduction
Briefly describes how the application performance can be optimized.

Use layout only when needed
Explains how to avoid unnecessary executions of the layout to improve performance.

Layout of huge graphs
Provides some hints on how to deal with huge graphs.

Speed of layout algorithms
Describes the speed of the different layout algorithms.

Introduction

If your application displays and manipulates only small graphs, you don't need to worry about performance optimizations. Some layout algorithms are designed for medium sized graphs, and only few algorithms are available for huge graphs. Graph layout is in general a complex task that often uses heuristics to solve NP-hard problems (problems that cannot be easily solved from a computational point of view). Different heuristics have different speed characteristics.

Graph layout places the nodes and routes the links. If moving nodes and reshaping links is slow, graph layout cannot be fast. Therefore, it is important to know the performance hints described in *Optimizing the performance of JViews Framework* in addition to those hints that are related to graph layout.

Use layout only when needed

The graph layout algorithm is most likely the most complex part of your application. Hence it might also be the slowest part of your application. Therefore it is useful to design the application so that graph layout is used only when needed. For instance, the application could offer a button that triggers layout, so that the graph layout does not need to run continuously during all interactions.

Graph layout places the nodes and routes the links. If moving nodes and reshaping links is slow, graph layout cannot be fast. Therefore, it is important to know the performance hints described in the *Optimizing the performance of JViews Framework* in addition to those hints that are related to graph layout.

Orthogonal links without link layout

If your application requires orthogonal link shapes, you might be tempted to use a link layout in automatic layout mode. This has the effect that the layout is triggered whenever a node moves. However, if you have too many links, a full automatic link layout will be too slow. An alternative way to ensure orthogonal links is to use the orthogonal mode of `IlvEnhancedPolylineLinkImage` (and its subclasses). This mode will ensure that the link shape remains orthogonal, without analyzing all links to reduce link crossings and overlaps. Therefore it can be more efficient than running link layout in autolayout mode. To enable the orthogonal mode on a link, call:

```
enhancedLinkImage.setOrthogonal(true);
```

Automatic layout

If you must use automatic layout (`setAutoLayout`), be aware that layout is triggered whenever any event is fired indicating a change of the graph. In this case, it is important to optimize the events by using adjusting sessions as explained in *Events and listeners*. This avoids that a sequence of changes triggers many layouts and ensures that the layout is only called once at the end of the sequence of changes.

Layout of huge graphs

Different graph layout algorithms can handle different graph sizes. As a rule of thumb:

- ♦ `IlvGridLayout`, `IlvBusLayout`, and `IlvTreeLayout` can handle huge graphs.
- ♦ `IlvHierarchicalLayout` can handle medium sized graphs that don't have too many links.
- ♦ `IlvUniformLengthEdgesLayout` is the slowest algorithm and is not suitable for huge graphs.

More details are given in *Speed of layout algorithms* .

Don't show all links

The tree layout can handle very huge graphs. If you have a very huge graph with links, the recommended way is to show only a spanning tree of the graph and hide the other links. The spanning tree can be laid out with the tree layout.

Design your application to use interactions to make the user is aware of the hidden links. For example, selecting one node may highlight all nodes that are reachable from this node via hidden or visible links. This is even more ergonomic than displaying all the links at the same time, since the user's eyes cannot trace links if too many are displayed at the same time.

The code example **<installdir>/jviews-diagrammer86/codefragments/graphlayout/filterlinks/nocss** shows how this strategy can be implemented.

Cluster into subgraphs and collapse

Sometimes, graphs have meaningful cluster information. For instance, a graph of people can be clustered according to nationality, or according to families. Each cluster can be represented as a nested subgraph (`IlvGrapher`) that can be collapsed and expanded.

The advantage of subgraphs is having a faster layout when they are collapsed, since a faster layout does not need to lay out the inner of the clusters. The diagram also becomes more comprehensible if only those details of interest are shown and the less interesting subgraphs are collapsed. On the other hand, if all subgraphs are expanded, the layout may become slow if the nesting depth of subgraphs is too high. When dealing with very large graphs, a carefully designed clustering into nested subgraphs can help to improve the user experience of the application.

Speed of layout algorithms

The speed of the graph layout algorithms depend on the type of graph and on the layout parameters. Some layouts are very slow for specific types of graphs. Some layouts are fast in general but become slow for specific layout parameter settings. The speed of all layouts depends also on the performance of the nodes and links that are used. If you use nodes classes with a very inefficient implementation for moving the nodes, or link classes with a very inefficient implementation for reshaping the links, then the layout algorithm will also be slow, since it moves nodes and reshapes links.

Layout customization interfaces

Several layout algorithms support customization interfaces such as `IlvNodeBoxInterface`, `IlvNodeSideFilter`, `IlvLinkConnectionBoxInterface` or `IlvLinkClipInterface`. When you use these interfaces, the layout algorithm jumps into your code. Be careful when implementing these interfaces, so that they do not decrease the performance of the layout algorithm.

Tree layout

Tree Layout is in general very fast and can handle huge graphs, as long as none of the automatic tip over modes are used.

- ♦ The layout mode FREE and LEVEL are the fastest modes.
- ♦ The radial layout modes are a bit slower, but usually still fast enough for very large graphs.
- ♦ The tip over layout modes are slow and should only be used for small graphs.

For details, see Tree Layout (TL).

Hierarchical layout

The speed of hierarchical layout depends on the density of the graph. Hierarchical layout can handle very large graphs that have only few links, but it might be too slow for smaller graphs that have a huge amount of links.

The speed and quality of the hierarchical layout also depends on the number of constraints: The fewer constraints, the more freedom the layout has to place nodes and the faster the layout is. In particular, you should avoid unsatisfiable constraint conflicts, because detecting these conflicts is very slow.

For details, see Hierarchical Layout (HL).

Grid layout

Grid layout is in general very fast and can handle huge graphs. However, in layout mode `TILE_TO_MATRIX`, the speed depends on the grid mesh size (the distance between gridlines). The smaller the grid mesh size, the slower the layout. In the other layout modes, the grid mesh size has no influence on the performance.

For details, see Grid layout (GL).

Link layout

The speed of link layout depends on the number of links. It is suitable for small and medium size graphs. If the graph has too many links, see section Orthogonal links without link layout.

In layout mode `LONG_LINKS` (or when using the `IlvLongLinkLayout`) the layout speed depends on the grid mesh size (the distance between gridlines). The smaller the grid mesh size, the slower the layout. Additionally, the exhaustive search mode (`setExhaustiveSearching`) of `IlvLongLinkLayout` should be avoided, because it is very slow.

For details, see Link layout (LL).

Uniform length edges layout

To fit a variety of needs, the algorithm provides three optional modes: incremental, non-incremental and fast multilevel. The later is the fastest for medium and large graphs. For more details on these modes, see Layout mode .

Index

Symbols

!important token **119**

A

abstract methods
- unused **69**

accessing
- renderer **237**

adding
- border in SDM general link **153**
- decorations to general node **150**
- renderer **219**

adding a decoration **150**
adding categories **264**
addObject method
- IlvSDMModel interface **59**

addSDMModelListener method
- IlvSDMModel interface **59**

addSDMPropertyChangeListener method
- IlvSDMModel interface **59**

aligning graphics **280**
alternateColor property **157**
anchor **278**
animation
- SDM general link **165**

anti-aliasing
- label **147**

applying
- CSS to Java objects **121**

approach
- lower-level **46**

area events, content on demand **101**
arrow
- by a decoration **162**
- decoration **162**
- drawing **161**

attachment
- anchor **278**
- hotspot **278**

Attachment Constraint **278**
Attachment Layout **278**
Attachment layout **276**
attribute matching
- CSS2 syntax **135**

autowrap
- label **146**

B

basic graphic objects **47**
Bean
- creating new **128**
- library **125**
- property **123, 175, 227**

Bezier curve **155**
Blinking renderer **184**
border
- adding in SDM general link **153**

C

cascading
- CSS syntax **119**
- CSS2 syntax **135**

categories **256**
- adding to palette **264**

Centered Layout **281**
Centered layout **276**
centering graphics **281**
class declaration in CSS **125**
class loader **286**
class property name **125**
clear method **76**
- IlvSDMModel interface **59**

collapsed nodes **212**
color
- label **147**

Coloring renderer **180**
comparing

attribute patterns **135**
composite
Attachment constraint **278**
Attachment Layout **278**
Attachment layout **276**
Centered layout **276**
nested **280**
Stacker **280**
Stacker layout **276**
composite graphics facility **50**
composite layout **276**
Composite renderer **198**
configuring
renderer by code **235**
content on demand **100, 106**
area events **101**
cache size **101**
lock, load, unload **101**
multithreading **102**
notifications **102**
set events **101**
tiling systems **108**
controlling
SDM general node icon **148**
SDM general node label **146**
SDM general node shape **141, 143, 152, 157, 158, 161, 164**
converting
data model to graphic object **24**
createChildren method **77**
createLink method
IlvSDMModel interface **59**
createNode method
IlvSDMModel interface **59**
creating
new Bean **128**
renderer **219**
SDM engine **24**
SDM general node **139**
creating palettes **263**
CSS
applying to Java objects **121**
cascading syntax **119**
class declaration **125**
class property name **125**
classes and ID **124**
declaration **117**
engine in SDM **123**
inheritance **120**
origins **111**
parent-child construct **82**
priority **118**
recursion **128**
selector **115**
sharing an object **278**
subobject **128**
use **16**
XML elements **124**
CSS class in the data model **82**
CSS engine
CSS class **82**
CSS for Java
composite graphics **276**
CSS for Java engine **124**
CSS syntax
divergences from CSS2 **135**
CSS2 syntax **135**
attribute matching **135**
cascading **135**
empty string **137**
enhancement **136**
null value **136**
pseudo-classes **126, 135**
pseudo-elements **126**
pseudoelements **135**
CSSclass attribute name **82**
CSSclass property **124**
curves
SDM general link **155**
custom functions
in expressions **130**
registering **133**
custom SDM model
JavaBeans **67**
custom shape **141**
customizing
renderer **175**
SDM general link **151**

D

dashes
SDM general link **156**
data model
dynamic **59**
editable **59**
indirection **126**
information **58**
listeners **58, 59**
navigating **58**
not editable **76**
writing **60**
data to graphics **24**
declaration
CSS syntax **117**
decoration
adding **150**
as arrow **162**
SDM general node **150**
Decoration renderer **182**
default interactor **243**
deploying applications **38**

diagram component **12**
DOM tree **95**
drawing
    arrow **161**
DrillDown renderer **193**
dynamic data model **59**

E

Eclipse Rich Client Platform **286**
    class loader **286**
    runtime plugin **286**
editable data model **59**
elements of a composite graphic **50**
empty string
    CSS2 syntax **137**
`enableAWTThreadRedirect` method
    `IlvEventThreadUtil` class **288**
end cap **154**
errors in CSS **130**
`EventQueue` class
    `isDispatchThread` method **288**
example
    link, adding a border **153**
    link, alternating colors **157**
    link, drawing an arrow **161**
    link, major appearance **152**
    link, mixing decoration options **158**
    link, modifying curve **155**
    link, modifying dash specification **156**
    link, setting a wave effect **164**
    setting shape size **141**
    using a renderer **174**
ExpandCollapse renderer **212**
expressions **129**
    css **130**
    inverting **133**

F

font
    label **147**
functionList property **133**
functions
    custom **130**
    standard **130**

G

general node
    label position **146**
    label string **146**
    multiline labels **146**
getChildren method
    IlvSDMModel interface **58**
getFilteredRenderer method
    IlvFilterSDMRenderer class **237**
getFrom method
    IlvSDMModel interface **58**
getID method
    IlvSDMModel interface **58, 124, 126**
getObject method
    IlvSDMModel interface **58**
getObjectProperty method
    IlvSDMModel interface **58**
getObjectPropertyNames method
    IlvSDMModel interface **58**
getObjects method **76**
    IlvSDMModel interface **58**
getParent method
    IlvSDMModel interface **58**
getRenderer method
    IlvRendererUtil class **237**
    IlvSDMEngine class **237**
getTag method
    IlvSDMModel interface **58, 124, 126**
getTo method
    IlvSDMModel interface **58**
GrapherPin renderer **187**
graphic object
    creating **47**
graphic objects
    basic **47**
GraphLayout renderer **175, 176, 189, 205**

H

HalfZooming renderer **195**
hotSpot **278**

I

icon
    general node **148**
ID
    and CSS classes **124**
    of model object **58**
IlvBasicSDMModel class **54, 87, 88**
IlvColoringRenderer class **180**
IlvDecorationRenderer class **182**
IlvDefaultSDMLink class **54**
IlvDefaultSDMModel class **54, 60, 61**
IlvDefaultSDMNode class **54**
`IlvEventThreadUtil` class
    `enableAWTThreadRedirect` method **288**
    `setAWTThreadRedirect` method **288**
IlvFilterSDMRenderer class **220, 225, 237**
    getFilteredRenderer method **237**
IlvGrapherPinRenderer class **187**
IlvGraphic class **124, 150**
IlvGraphicFactories class **125**
IlvGraphLayout class **190, 191**
IlvHalfZoomingRenderer class **195**
IlvHierarchicalLayout class **190**
IlvInfoBalloonRenderer class **197**
IlvInteractorRenderer class **200**
IlvJavaBeanSDMModel class **54, 60**
IlvLabelInterface interface **158**

IlvLabelLayoutRenderer class **201**
IlvLegendRenderer class **202**
IlvLinkLayoutRenderer class **175, 205, 214**
IlvMakeSDMLinkInteractor class **242, 244**
IlvMakeSDMNodeInteractor class **242, 244**
IlvMapRenderer class **206**
IlvObjectInteractor class **243**
IlvPalette **256**
IlvPalette class **263**
IlvPaletteCategory **264**
IlvPaletteCategory class **256**
IlvPaletteManager class **256**
IlvPaletteSymbol class **256**
IlvPolylineLinkImage class **154**
IlvRect class **125**
IlvRendererUtil class **237**
    getRenderer method **237**
IlvSDMEngine class
    getRenderer method **237**
    readDOM method **94**
    readXML method **94**
    writeDOM method **95**
    writeXML method **95**
IlvSDMModel interface
    addObject method **59**
    addSDMModelListener method **59**
    addSDMPropertyChangeListener method **59**
    clear method **59**
    createLink method **59**
    createNode method **59**
    getChildren method **58**
    getFrom method **58**
    getID method **58, 124, 126**
    getObject method **58**
    getObjectProperty method **58**
    getObjectPropertyNames method **58**
    getObjects method **58**
    getParent method **58**
    getTag method **58, 124, 126**
    getTo method **58**
    isAdjusting method **59**
    isLink method **58**
    removeObject method **59**
    removeSDMModelListener method **59**
    removeSDMPropertyChangeListener method **59**
    setAdjusting method **59**
    setFrom method **59**
    setID method **59**
    setObjectProperty method **59**
    setTo method **59**
IlvSDMRenderer class **220**
IlvShadowLabel class **150**
IlvStyleSheetRenderer class **208, 237**
IlvSubGraphRenderer class **212**
`IlvSwingControl` class **286**
`IlvSwingUtil` class
    `isDispatchThread` method **288**
IlvXMLConnector class **96**
IlvZoomableLabel class **158**
import statement **119**
InfoBalloon renderer **197, 243**
information on model objects **58**
inherit token **119**
inheritance
    CSS syntax **120**
    of declarations **120**
insets in composite graphic **281**
interactor
    default **243**
    renderer **243**
Interactor graphic property **247**
Interactor renderer **247**
inverting expressions **133**
isAdjusting method
    IlvSDMModel interface **59**
`isDispatchThread` method
    `EventQueue` class **288**
    `IlvSwingUtil` class **288**
`isEventDispatchThread` method
    `SwingUtilities` class **288**
isLink method
    IlvSDMModel interface **58**

J

JApplet (Swing) **40**
JavaBeans **175, 190**
JavaBeans (lower-level) **48, 49**
JavaBeans conventions **65**
join **154**
JTree **75**

L

label
    anti-aliasing **147**
    autowrap **146**
    color **147**
    font **147**
    general node **146**
    truncation **147**
    zoom **147**
label position **146**
label string **146**
LabelLayout renderer **201**
Lane Renderer **216**
layers
    in composite graphics **276**
layout
    of composite **276**
Layout Managers
    in composite graphics **276**

Legend renderer **202**
LINEAR_GRADIENT mode **143**
lineStyle property **156**
link **58**
  end cap **154**
  join **154**
  methods **58**
LinkLayout renderer **175, 205, 237, 238**
links
  striped **157**
listeners **58, 59**
literal
  css declaration **123**
loading palettes **258**
loading, content on demand **101**
locking, content on demand **101**
lower-level approach **46**

M

Map renderer **206**
Map StyleSheet renderer **210**
maps integration **32**
methods
  clear **76**
  createChildren **77**
miter join **154**
model indirection **126**
model object
  ID **58, 124**
  object tag **58**
  properties **58**
model property name **126**
modes
  SDM general link **152**
modifying **238**
multiline label **146**
multithreading, content on demand **102**

N

navigating
  data model **58**
nested composite graphic **280**
node **58**
  custom shape **141**
notification, content on demand **102**
null value
  CSS2 syntax **136**

O

object
  data model to graphic **24**
object interactors **242**
object tag
  of model object **58**

P

palette **256**
  creating **263**
  events **268**
  loading **258**
  parameters **266**
  saving **260**
palette manager **256, 269**
palette manager events **270**
palette symbols **256**
PaletteEvent **268**
parent-child construct in CSS **82**
printing controller **285**
printing framework
  customizing **285**
priority
  CSS syntax **118**
priority in display **46**
properties
  CSSclass **124**
  in a sheet **34**
  in a table **35**
  of model object **58**
  renderer **176**
  SDM general link **167**
pseudo-classes **135**
  CSS2 syntax **126, 135**
pseudo-elements
  CSS2 syntax **126, 135**
pseudoclass **176**

R

Radial mode (Tree layout) **82**
RADIAL_GRADIENT mode **143**
RCP **286**
readDOM method
  IlvSDMEngine class **94**
reading
  custom XML formats **96**
  custom XML formats using XSLT **96**
  XML data file **93**
readXML method
  IlvSDMEngine class **94**
recursion
  CSS **128**
registering
  custom functions **133**
removeObject method
  IlvSDMModel interface **59**
removeSDMModelListener method
  IlvSDMModel interface **59**
removeSDMPropertyChangeListener method
  IlvSDMModel interface **59**
renderer **238**
  accessing **237**
  adding a new **219**
  configuring by code **235**
  customizing **175**

modifying **238**
properties **176**
setting new **239**
using **173**
resizing
general node **149**
SDM general node **149**
resolving
URL **126**
root of the tree **76**
rule
specificity **118**
style **114**
runtime plugin **286**

S

saving palettes **260**
Scalable Vector Graphics (SVG)
supported/unsupported CSS properties **288**
SDM engine
creating **24**
SDM general link **167**
adding border **153**
animation **165**
curves **155**
customizing **151**
dashes **156**
modes **152**
stroke **154**
summary of properties **167**
wave effect **164**
zoom **165**
SDM general node
automatic resizing **149**
controlling icon **148**
controlling label **146**
controlling shape **141, 143, 152, 157, 158, 161, 164**
creating **139**
decorations **150**
SDM model **76**
selector
CSS syntax **115**
set events, content on demand **101**
setAdjusting method
IlvSDMModel interface **59**
`setAWTThreadRedirect` method
`IlvEventThreadUtil` class **288**
setFrom method
IlvSDMModel interface **59**
setID method
IlvSDMModel interface **59**
setObjectProperty method
IlvSDMModel interface **59**
setting
new renderer **239**
setTo method
IlvSDMModel interface **59**
shape
general node **141, 143, 152, 157, 158, 161, 164**
shared object in CSS **278**
sharing **137**
specificity **118**
Stacker Layout **280**
Stacker layout **276**
standard functions
in expressions **130**
Standard Widget Toolkit **286**
striped links **157**
stroke
SDM general link **154**
style rule **114**
style sheet **16**
changing properties **208**
StyleSheet renderer **172, 237**
subclassing a graphic object **47**
subclassing the XML connector **96**
SubGraph renderer **212**
subgraphs **212**
subobject **128**
swim lanes **214**
SwimLanes renderer **214**
Swing JApplet **40**
Swing JTree **75**
`SwingUtilities` class
`isEventDispatchThread` method **288**
symbols
in CSS **273**
syntax enhancement
CSS2 syntax **136**

T

tag **26**
TEXTURE mode **143**
tiling systems, content on demand **108**
tokens
cascading priority **119**
Tree layout **82**
TreeSDMNode class **77**
truncation
label **147**

U

unloading, content on demand **101**
URL
resolving **126**
user-defined type **26**
example in CSS **277**

V

view interactors **242**

W

wave
  in SDM general link **164**
web deployment **42**
writeDOM method
  IlvSDMEngine class **95**
writeXML method
  IlvSDMEngine class **95**
writing
  custom XML formats **96**
  custom XML formats using XSLT **96**
  data model **60**
  XML data file **95**

X

XML data file
  custom formats **96**
  custom formats using XSLT **96**
  reading **93**
  subclassing the XML connector **96**
  writing **95**
XML element name in CSS **124**
XSLT format **96**

Z

zoom
  label **147**
  SDM general link **165**