IBM ILOG JViews Diagrammer V8.6

Using graph layout algorithms

Table of contents

Conventions and Bibliography

Conventions

In CSS means that you create or modify a Cascading Style Sheet file that is used in your diagram component or SDM component as a style sheet. A CSS file does not contain Java™ code, it contains style rules.

Layout parameter names in the `GraphLayout`, `LinkLayout`, and `LabelLayout` sections always start with a lowercase letter. Layout parameter names in the node or link rules always start with an uppercase letter.

In Java means that you write Java™ code.

Accessors and Modifiers

Very often, you can set and retrieve a property of a class by using a pair of modifier/accessor methods, such as:

```
setFlowDirection(int direction);
int getFlowDirection();
setIncrementalMode(boolean mode);
boolean isIncrementalMode();
```

This document uses the standard Java naming scheme for the modifiers and accessors, that is, the *set* and *get/is* methods. However, when explaining the Java API, it often mentions only the *set* method. Please refer to the For a detailed list of all the *get/is* methods, see the *Java API Reference Documentation* at `index`.

Books

Several books dedicated to graph layout have been published:

Di Battista, Giuseppe, Peter Eades, Roberto Tammassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*, Prentice Hall, 1999. See:

http://www.cs.brown.edu/people/rt/gdbook.html

or

http://www.mypearsonstore.com/bookstore/product.asp?isbn=0133016153.

Kaufmann, Wagner (Eds.): *Drawing Graphs*, Lecture Notes in Computer Science Vol. 2025, Springer 2001. See:

http://link.springer.de/link/service/series/0558/tocs/t2025.htm.

Graph layout is closely related to graph theory, for which extensive literature exists. See:

Clark, John and Derek Allan Holton. *A First Look at Graph Theory*. World Scientific Publishing Company, 1991.

For a mathematics-oriented introduction to graph theory, see:

Diestel, Reinhard, *Graph Theory*, 2nd ed., Springer-Verlag, 2000.

A more algorithmic approach may be found in:

Gibbons, Alan. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

Gondran, Michel and Michel Minoux. *Graphes et algorithmes*, 3rd ed., Eyrolles, Paris, 1995 (in French).

Bibliography

A comprehensive bibliographic database of papers in computational geometry (including graph layout) can be found at:

The Geometry Literature Database

http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html.

The recommended bibliographic survey paper is the following:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. “Algorithms for Drawing Graphs: an Annotated Bibliography.” *Computational Geometry: Theory and Applications* 4 (1994): 235-282 (also available at

http://www.cs.brown.edu/people/rt/gd-biblio.html.

Journals

The following are electronic journals:

Journal of Graph Algorithms and Applications

http://jgaa.info/

Algorithmica

http://link.springer-ny.com/link/service/journals/00453/

Computational Geometry: Theory and Applications

http://www.elsevier.com/locate/comgeo

Journal of Visual Languages and Computing

http://www.elsevier.com/locate/jvlc

The following journals occasionally publish papers on graph layout:

Information Processing Letters

http://www.elsevier.com/locate/ipl

Computer-aided Design

http://www.elsevier.com/locate/cad

IEEE Transactions on Software Engineering

http://www.computer.org/tse/

Many papers are presented at conferences in Combinatorics and Computer Science.

Conferences

An annual Symposium on Graph Drawing has been held since 1992. The proceedings are published by Springer-Verlag in the *Lecture Notes in Computer Science* series.

The 2008 Symposium on Graph Drawing was held in Heraklion, Crete, Greece:

http://gd2008.org/

The 2009 Symposium will be held in Chicago, USA.

Introducing graph layout

Describes the IBM® ILOG® JViews graph layout package and its features.

In this section

What is IBM® ILOG® JViews graph layout?

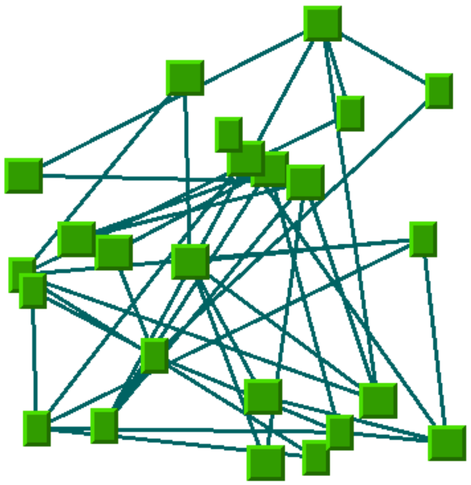
Graph layout is a set of algorithms that optimize the display of nodes and links with respect to each other in graphs such as network topologies for telecommunications networks and systems management applications.

Many types of complex business data can best be visualized as a set of nodes and interconnecting links, more commonly called a graph or a network. Examples of graphs include business organization charts, workflow diagrams, telecom network displays, and genealogical trees. Whenever these graphs become large or heavily interconnected, it becomes difficult to see the relationships between the various nodes and links (the "edges"). This is where IBM® ILOG® JViews graph layout algorithms help.

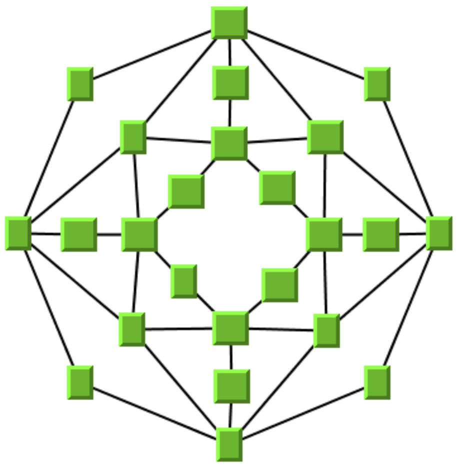
Graph layout provides high-level, ready-to-use relationship visualization services. It allows you to take any "messy" graph and apply a sophisticated graph layout algorithm to rearrange the positions of the nodes and links. The result is a more readable and understandable picture.

Take a look at two sample drawings of the same graph.

Here, no formal layout algorithm was used. The nodes were placed randomly when the graph was drawn.



Using one of the layout algorithms provided with the product, the following drawing was obtained.



In the second drawing, the layout algorithm has distributed the nodes quite uniformly, avoiding overlapping nodes and showing the symmetries of the graph. This drawing presents a much more readable layout than the first drawing.

The concept of graph layout

Simply speaking, a graph is a data structure that represents a set of entities, called nodes, connected by a set of links. A node can also be referred to as a vertex. A link can also be referred to as an edge or a connection. In practical applications, graphs are frequently used to model a very wide range of things: computer networks, software program structures, project management diagrams, and so on. Graphs are powerful models because they permit applications to benefit from the results of graph theory research. For instance, efficient methods are available for finding the shortest path between two nodes, the minimum cost path, and so on.

Layout of a graph

Graph layout is used in graphical user interfaces of applications that need to display graph models. To lay out a graph means to draw the graph so that an appropriate, readable representation is produced. Essentially, this involves determining the location of the nodes and the shape of the links. For some applications, the location of the nodes may already be known (for example, based on the geographical positions of the nodes). However, for other applications, the location is not known (a pure “logical” graph) or the known location, if used, would produce an unreadable drawing of the graph. In these cases, the location of the nodes must be computed.

What is meant by an “appropriate” drawing of a graph? In practical applications, it is often necessary for the graph drawing to observe certain quality criteria. These criteria may vary depending on the application field or on a given standard of representation. It is often difficult to tell what a good layout consists of. Each end user may have different, subjective criteria for qualifying a layout as “good”. However, one common goal exists behind all the criteria and standards: the drawing must be easy to understand and provide easy navigation through the complex structure of the graph.

What is a good layout?

To deal with the various needs of different applications, many classes of graph layout algorithms have been developed. A layout algorithm addresses one or more quality criteria, depending on the type of graph and the features of the algorithm, when laying out a graph.

The most common criteria are:

- Minimizing the number of link crossings
- Minimizing the total **area** of the drawing
- Minimizing the number of **bends** (in orthogonal drawings)
- Maximizing the smallest **angle** formed by consecutive incident links
- Maximizing the display of **symmetries**

How can a layout algorithm meet each of these quality criteria and standards of representation? If you look at each individual criteria, some can be met quite easily, at least for some classes of graphs. For other classes, it may be quite difficult to produce a drawing that meets the criteria. For example, minimizing the number of link crossings is relatively simple for trees (that is, graphs without cycles). However, for general graphs, minimizing the number of link crossings is a mathematical NP-complete problem (that is, with all known algorithms, the time required to perform the layout grows very fast with the size of the graph).

Moreover, if you want to meet several criteria at the same time, an optimal solution may not exist with respect to each individual criteria because many of the criteria are mutually contradictory. Time-consuming trade-offs may be necessary. In addition, it is not a trivial task to assign weights to each criteria. Multicriteria optimization is, in most cases, too complex to implement and much too time-consuming. For these reasons, layout algorithms are often based on heuristics and may provide less than optimal solutions with respect to one or more of the criteria. Fortunately, in practical terms, the layout algorithms will still often provide reasonably readable drawings.

Methods for using layout algorithms

Layout algorithms can be employed in a variety of ways in the various applications in which they are used. The most common ways of using an algorithm are the following:

♦ *Automatic layout*

The layout algorithm does everything without any user intervention, except for perhaps the choice of the layout algorithm to be used. Sometimes, a set or rules can be coded to choose automatically (and dynamically) the most appropriate layout algorithm for the particular type of graph being laid out.

♦ *Semiautomatic layout*

The end user is free to improve the result of the automatic layout procedure by hand. In some cases, the end user can move and “pin” nodes at desired locations and perform the layout again. In other cases, a part of the graph is automatically set as “read-only” and the end user can modify the rest of the layout.

♦ **Static layout**

The layout algorithm is completely redone (“from scratch”) each time the graph is changed.

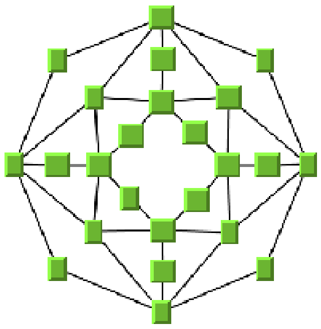
♦ **Incremental layout**

When the layout algorithm is performed a second time on a modified graph, it tries to preserve the stability of the layout as much as possible. The layout is not performed again from scratch. The layout algorithm also tries to save CPU time by using the previous layout as an initial solution. Some layout algorithms and layout styles are incremental by nature. For others, incremental layout may be impossible.

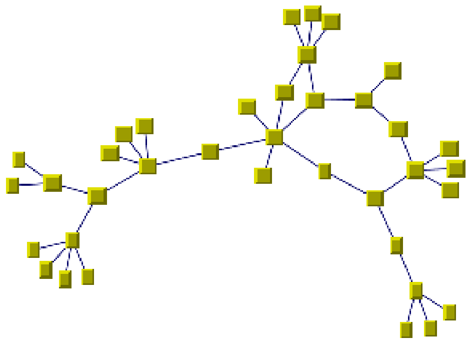
The graph layout algorithms

The graph layout package provides numerous ready-to-use layout algorithms. They are shown below with sample illustrations. In addition, you can develop new layout algorithms using the generic layout framework.

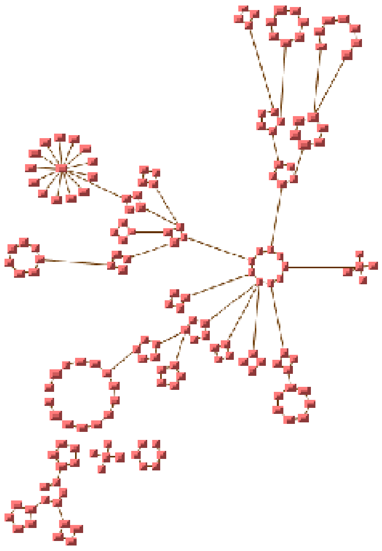
Topological Mesh Layout (TML)



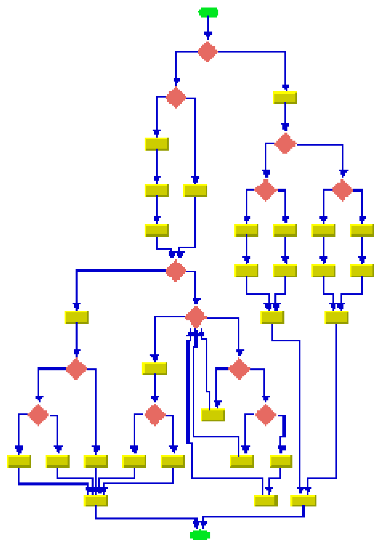
Force-directed or Uniform Length Edges Layout (ULEL)



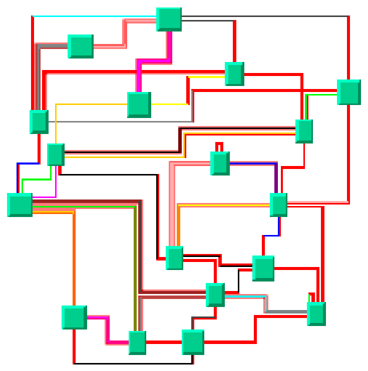
Circular layout (CL)
(Ring/Star)



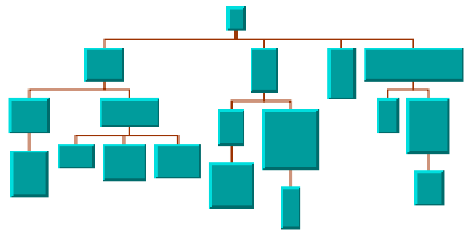
Hierarchical Layout (HL)



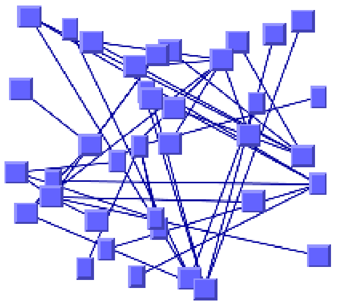
Link layout
(LL)



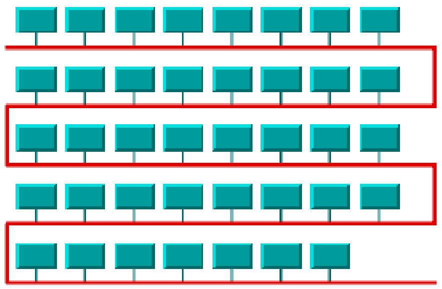
Tree Layout
(TL)



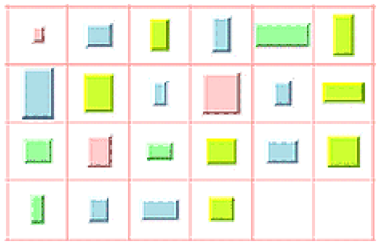
Random layout
(RL)



Bus layout
(BL)



Grid layout
(GL)



Structure of the graph layout API

The IBM® ILOG® JViews graph layout API is composed of:

- *The generic graph layout package*
- *The layout algorithm packages*
- *The label layout package*
- *The Swing components*

The generic graph layout package

`ilog.views.graphlayout`: A high-level, generic framework for the graph layout services provided by IBM® ILOG® JViews.

The layout algorithm packages

- `ilog.views.graphlayout.bus`: A layout algorithm designed to display bus network topologies (that is, a set of nodes connected to a bus node).
- `ilog.views.graphlayout.circular`: A layout algorithm that displays graphs representing interconnected ring and/or star network topologies.
- `ilog.views.graphlayout.grid`: A layout algorithm that arranges the disconnected nodes of a graph in rows, in columns, or in the cells of a grid.
- `ilog.views.graphlayout.hierarchical`: A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.
- `ilog.views.graphlayout.link`: A layout algorithm that reshapes the links of a graph without moving the nodes.
  - `ilog.views.graphlayout.link.longlink`: For long orthogonal links.
  - `ilog.views.graphlayout.link.shortlink`: For short links.
- `ilog.views.graphlayout.multiple`: A facility that combines multiple layout algorithms and treat them as one algorithm object.
- `ilog.views.graphlayout.random`: A layout algorithm that moves the nodes of the graph at randomly computed positions inside an user-defined region.
- `ilog.views.graphlayout.recursive`: A layout algorithm that can be used to control the layout of nested graphs (containing subgraphs and intergraph links).
- `ilog.views.graphlayout.topologicalmesh`: A layout algorithm that can be used to lay out cyclic graphs.
- `ilog.views.graphlayout.tree`: A layout algorithm that arranges the nodes of a tree horizontally or vertically, starting from the root of the tree. A radial layout mode allows you to arrange the nodes of a tree on concentric circles around the root of the tree.

- `ilog.views.graphlayout.uniformlengthedges`: A layout algorithm that can be used to lay out any type of graph and allows you to specify the length of the links.

The label layout package

`ilog.views.graphlayout.labellayout`: A layout algorithm for automatic placement of labels.

- `ilog.views.graphlayout.labellayout.annealing`: For close label positioning.
- `ilog.views.graphlayout.labellayout.random`: For random placement.

The Swing components

`ilog.views.graphlayout.swing`: Swing components useful for creating applications mixing IBM® ILOG® JViews Diagrammer graph layout and Swing.

Common features of graph layout algorithms

The graph layout algorithms share the following features:

- **Programmable:** All graph layout algorithms can be tailored through code. You do not need a diagram component to be able to use the graph layout algorithms. The algorithms can be attached to a grapher (class `IlvGrapher`) directly.
- **Adaptable to any graph data structure:** Not even an IBM® ILOG® JViews grapher is required. You can program your own graph data structures and apply an IBM® ILOG® JViews Diagrammer graph layout algorithm to them.
- **Extensible:** you can easily use the generic framework to implement your own graph layout algorithm, or to combine smaller layout algorithms into a larger one.
- **Suitable for nested graphs:** The graph layout framework provides capabilities to lay out graphs that contain other graphs as nodes. It can even route intergraph links that run between different subgraphs of a nested graph.
- **Economic and automatic:** The graph layout framework has capabilities to perform a layout only when needed, i.e. when a parameter or a detail of the graph has changed. Furthermore, the framework has the capability to react automatically to such a change.
- **Selective:** You can apply different layout algorithms to different parts of a graph. For instance, you can apply a layout only to the nodes and links that are on user-defined layers of the graph, or only to parts that meet user-defined conditions.
- **Time controlled:** All layout algorithms can be set to stop automatically when a time has elapsed. Some layout algorithms can even be interrupted during runtime.
- **Stylable:** All graph layout algorithms can be used in a diagram component and all important graph layout settings can be controlled by Cascading Style Sheets (CSS).

Getting started with graph layout

Provides information to help you start using the IBM® ILOG® JViews Graph Layout functionality.

In this section

Using layout algorithms

Explains how to use layout algorithms.

In this section

Different ways to use layout algorithms

You can use graph layout algorithms with or without a diagram component. There are several ways to use the graph layout algorithms:

- By specifying a style sheet for graph layout in a diagram component
- By using the graph layout API in a diagram component
- By using the graph layout API in a graphics framework application

The diagram component (class `IlvDiagrammer`) allows you to configure the graph layout entirely through style sheets. Internally, it uses the graph layout API and the Graphics Framework, but it simplifies their usage by a high-level API and by the expressiveness of the CSS language.

It is not mandatory to use a diagram component (class `IlvDiagrammer`) to perform graph layout. The graph layout algorithms work also on graphers (class `IlvGrapher`).

The diagram component uses the grapher infrastructure internally, adds style sheets and simplifies the usage. But if style sheets are not required by the application, you can just use graph layout algorithms without any diagram component.

Running a graph layout application with a diagram component

To use the layout algorithms provided by the graph layout package inside a diagram component, you usually perform the following steps:

1. Create a style sheet (CSS file) that specifies the graph layout. You can use the JViews Diagrammer Designer to create the style sheet interactively, or a text editor to create the style sheet by editing the CSS text.
2. Create an `IlvDiagrammer` diagram component and fill it with data from the data model.
3. Load the style sheet into the diagram component. Graph layout is automatically performed when the style sheet is loaded or changed.

Sample CSS file for graph layout

You can use the sample style sheet provided to get started with the layout algorithms of the graph layout package in an IBM® ILOG® JViews Diagrammer application. It illustrates how to specify a layout algorithm and the layout parameters in a CSS file. The example uses the Tree Layout, but most of the principles apply to any of the other layouts.

The complete style sheet is named `Sample.css` and is located in **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample1/data/Sample.css**

Since the Tree Layout applies link reshaping as well as laying out the nodes, an additional link layout is not necessary. Therefore, the link layout is set to `false`. Besides the layout style, the GraphLayout section of the style sheet specifies the global layout parameters of the layout style. Parameters that are not specified take the default value.

```
SDM {
   GraphLayout : "true";
   LinkLayout : "false";
}

node {
   class : "ilog.views.sdm.graphic.IlvGeneralNode";
   ...
}

link {
   class : "ilog.views.sdm.graphic.IlvGeneralLink";
   ...
}

GraphLayout {
   graphLayout : "Tree";
   flowDirection : "Bottom";
   layoutMode : "FREE";
   globalLinkStyle : "ORTHOGONAL_STYLE";
   globalAlignment : "CENTER";
   connectorStyle : "EVENLY_SPACED_PINS";
   siblingOffset : "15";
   branchOffset : "30";
   parentChildOffset : "20";
   position : "200,20";
}
```

Specifying graph layout in detail

The graph layout algorithm in the sample CSS file is configured according to the CSS specification for graph layout.

The `SDM` style rule specifies that a graph layout renderer is created. An SDM renderer is a pluggable object that controls the rendering of the graph. The graph layout renderer calls a layout algorithm. By default, a layout is in enabled mode and, therefore, is applied whenever the diagram changes.

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

The renderer corresponds to the following Java™ class:

`IlvGraphLayoutRenderer`

The parameters of the graph layout renderer and of the graph layout algorithm are specified in the `GraphLayout` rule.

```
GraphLayout {
  graphLayout : "Tree";
  flowDirection : "Bottom";
  ...
}
```

The `GraphLayout` rule contains the following declarations:

♦ The first declaration tells the graph layout renderer to use a tree layout algorithm, which corresponds to the Java class `IlvTreeLayout`. This declaration calls the method `setGraphLayout(ilog.views.graphlayout.IlvGraphLayout)`. You can pass the name of any subclass of `IlvGraphLayout`. The class name can be abbreviated, for example, `Tree`, or you can pass the full class name.

♦ The second declaration tells the tree layout algorithm to lay nodes out from top to bottom. This declaration calls the method `setFlowDirection(int)`.

Other graph layout or renderer parameters can be specified in a similar way. If a parameter is not specified then its default value is used. When the graph layout renderer is enabled (the default), it reapplies the layout whenever an object is changed or moved.

Note: When the graph layout renderer is enabled and a hierarchical layout or tree layout is in use, it is not necessary to use the link layout renderer, because the graph layout renderer has full control over the layout of nodes and links. Other graph layouts may control only the node layout, in which case the link layout renderer can be used to position the links.

Running the sample IBM® ILOG® JViews Diagrammer application

When reading a style sheet, the application automatically performs the graph layout as specified in the style sheet.

The source code of the application is named `Sample1.java` and is located in **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample1/src/Sample1.java.**

To compile and run sample 1:

1. Go to the `sample1` directory. On Microsoft® Windows® systems, you must open a Command Prompt window.
2. Set the `CLASSPATH` variable to the IBM® ILOG® JViews Diagrammer library and the current directory.

On Microsoft Windows systems	`.;<installdir>\jviews-diagrammer86\lib\jviews-diagrammer-all.jar;<installdir>\jviews-framework86\lib\jviews-framework-all.jar;<installdir>\jlm`
On UNIX® systems	`.:<installdir>/jviews-diagrammer86/lib/jviews-diagrammer-all.jar:<installdir>/jviews-framework86/lib/jviews-framework-all.jar:<installdir>/jlm`

   The `jlm` directory, which contains the license keys (the file `keys.jlm`), must be in the class path.

3. Compile the application as follows:

```
javac -d . src/Sample1.java
```

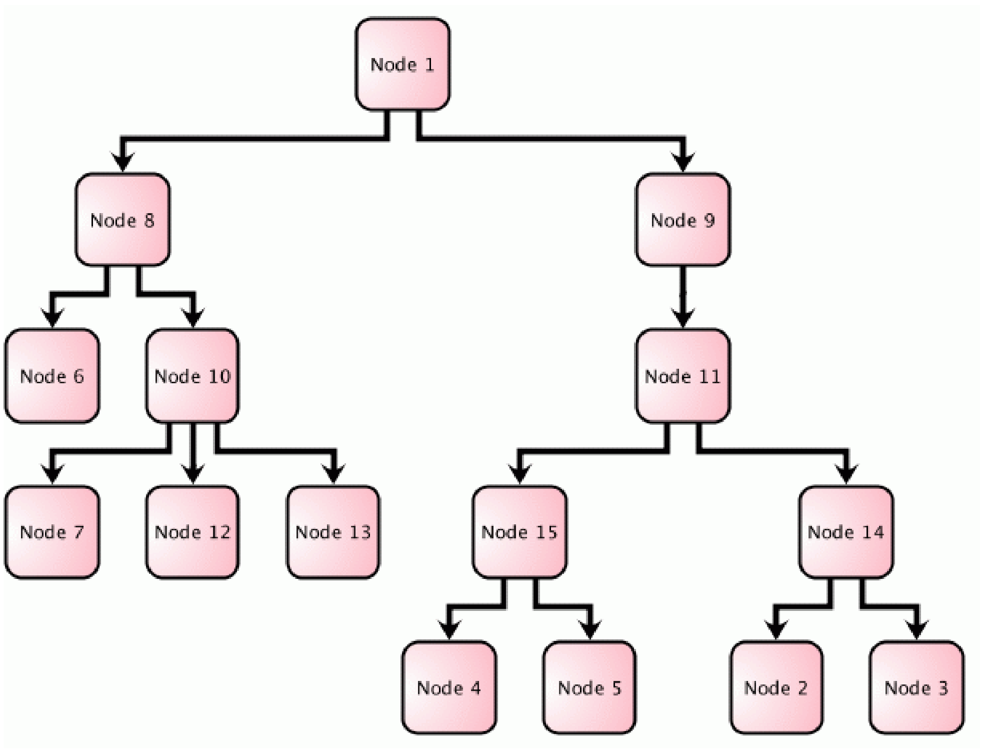
4. Run the application as follows:

```
java Sample1
```

The `Sample1.java` file contains the following code:

```
// the Diagrammer Framework
import ilog.views.diagrammer.*;
// the Java AWT package
import java.awt.*;
// The Java Net package
import java.net.*;
// the Java Swing package
import javax.swing.*;

public class Sample1
{
  public static void main(String[] arg)
  {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        // Create the diagrammer component
        IlvDiagrammer diagrammer = new IlvDiagrammer();

        // Change diagrammer parameters.
        diagrammer.setSelectMode(false);
        diagrammer.setScrollable(false);
        diagrammer.setEditingAllowed(true);
        diagrammer.getView().setBackground(Color.white);
        diagrammer.getView().setForeground(SystemColor.windowText);

        // The name of the XML file containing the model data
        String xmlFileName = "data/Sample.xml";

        // The name of the CSS file containing the style sheet
        String cssFileName = "data/Sample.css";

        // Load the sample data file
        try {
          diagrammer.setDataFile(new URL("file:" + xmlFileName));
        } catch (Exception e) {
          System.out.println("could not read " + xmlFileName);
          return;
        }

        // Load the style sheet.
        // Since layout is fully specified in the style sheet, loading the
        // style sheet performs the layout.
        try {
          diagrammer.setStyleSheet(new URL("file:" + cssFileName));
        } catch (Exception e) {
          System.out.println("could not read " + cssFileName);
          return;
        }

        // A Swing Frame to display
        JFrame frame = new JFrame("Layout Sample");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);


        // Put the manager view inside the Swing Frame
        frame.getContentPane().add(diagrammer);

        frame.setSize(600, 600);
        frame.setVisible(true);
      }
    });
  }
}
```

The sample style sheet and sample application produce the graph shown in *Output from Sample Java™ Application*:



Output from Sample Java™ Application

Running a graph layout application with link layout

The diagram component (class `IlvDiagrammer`) distinguishes between node layout and link layout. The node layout is specified as a normal graph layout. The link layout executes after the node layout. It keeps the calculated node positions and only reshapes the links.

A typical use of the link layout is when the node layout should only be applied on demand (for example, by clicking on a “perform layout” button), but the link layout should be applied automatically. When the user moves the nodes interactively, only the link layout should be applied.

To set up link layout to work automatically:

1. Create a style sheet (CSS file) that specifies the node layout and link layout. In this style sheet, disable the graph layout and enable the link layout. You can use the JViews Diagrammer Designer to create the style sheet interactively, or a text editor to create it by editing the CSS text.
2. Create an `IlvDiagrammer` component and fill it with data from the data model.
3. Load the style sheet into the diagram.
4. Create a button that allows the user to perform the graph layout on demand.

Sample CSS file for link layout

The complete style sheet is named `Sample.css` and is located in **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample1a/data/Sample.css**.

In this example, a hierarchical node layout is used, and a link layout is specified in addition. The graph layout renderer is not permanently enabled, which means that the node layout is not automatically applied. Only the link layout is applied automatically whenever the diagram changes.

```
SDM {
  GraphLayout : "true";
  LinkLayout : "true";
}

node {
  class : "ilog.views.sdm.graphic.IlvGeneralNode";
  ...
}

link {
  class : "ilog.views.sdm.graphic.IlvGeneralLink";
  ...
}

GraphLayout {
  graphLayout : "Hierarchical";
  flowDirection : "Bottom";
  enabled : "false";
}

LinkLayout {
  hierarchical: "true";
}
```

Specifying link layout in detail

The link layout algorithm in the sample CSS file is configured according to the CSS specification for graph layout.

The SDM style rule specifies that both a link layout renderer and a graph layout renderer are created.

```
SDM {
  GraphLayout : "true";
  LinkLayout :  "true";
}
```

The link layout renderer routes links in a logical way. It corresponds to the following Java™ class:

`IlvLinkLayoutRenderer`

The graph layout renderer, on the other hand, is disabled, since graph layout should not be applied automatically but only on demand.

```
GraphLayout {
  ...
  enabled : "false";
}
```

The link layout renderer is enabled by default. The LinkLayout style rule allows you to specify parameters of the link layout renderer and of the link layout.

```
LinkLayout {
  hierarchical : "true";
}
```

The parameter `hierarchical` tells the link layout renderer to use a hierarchical layout algorithm to route links. This declaration calls the method `setHierarchical(boolean)`.

Applying graph layout on demand

The application that reads the style sheet into a diagram component is very similar to the previous example. The source code of the application is named `Sample1a.java` and is located in **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample1a/src/Sample1a.java.**

The link layout is applied automatically, but the node layout is applied only on demand. To perform a node layout, call the method

```
diagrammer.layoutAllNodes()
```

To implement a button that allows the user to request a node layout, you need to define a Swing action that calls the method `layoutAllNodes`.

You can derive your application from `IlvDiagrammerApplication`. This is an application that encapsulates an `IlvDiagrammer` component. It contains already a toolbar with several standard buttons. There is already a built-in action in IBM® ILOG® JViews Diagrammer that calls `layoutAllNodes`, so all you need to do is add this action to the toolbar, using:

```
toolbar.addAction(IlvDiagrammerAction.layoutAllNodes)
```

When you move the nodes of a graph without the graph layout renderer being enabled, the link layout renderer rearranges the links accordingly.

When you click the graph layout button, the graph layout renderer redraws the graph according to the requested graph layout.

Running a graph layout application with dynamic layout parameters

The sample application allows you to load a style sheet and to apply graph layout, but not to change the layout parameters dynamically in the program. All layout parameters are defined statically in the style sheet.

The diagram component allows you to specify multiple style sheets to apply to the data. You can use this mechanism to change parameters dynamically.

Another way to implement dynamically changing parameters is to call the API of the graph layout class directly.

To specify multiple style sheets:

1. Create a main style sheet (CSS file) that specifies all the static layout parameters that will not be changed by the application.
2. Create an `IlvDiagrammer` component and fill it with data from the data model.
3. Load the main style sheet into the diagram component.
4. Add as second style sheet, that is, a mutable style sheet (class `IlvMutableStyleSheet`).

   A mutable style sheet is a memory representation of a style sheet that is suitable to hold dynamic non-persistent layout parameters.

After specifying multiple style sheets, you can use the API of the mutable style sheet to change the dynamic parameters.

For example, to change the flow direction of a Tree layout from bottom to right, call the API of the mutable style sheet to replace the CSS rule that defines the flow direction `Bottom` by a rule that specifies the flow direction `Right`. Graph layout is automatically performed when the style sheet changes.

All CSS specifications of graph layout parameters have a corresponding API in the graph layout class.

For example, the CSS specification:

```
GraphLayout {
   flowDirection : "Bottom";
}
```

corresponds to the following API call in a tree layout:

```
IlvTreeLayout treeLayout =
    (IlvTreeLayout) diagrammer.getEngine().
          getNodeLayoutRenderer().getGraphLayout();
treelayout.setFlowDirection(IlvDirection.Bottom);
```

To use direct API calls to change parameters dynamically:

1. Create a style sheet (CSS file) that specifies all the static layout parameters that will not be changed by the application.
2. Create an `IlvDiagrammer` component and fill it with data from the data model.

3. Load the main style sheet into the diagram component.
4. Access the layout instance of the graph layout renderer to change the layout parameters dynamically.

When changing graph layout parameters in this way, the graph layout is not automatically performed. You must explicitly perform the layout by calling the corresponding API on the diagram component (`diagrammer.layoutAllNodes()`).

Running the sample application that uses mutable style sheets

When the mutable style sheet changes, the application automatically performs the graph layout. The source code of the application is named `Sample2.java` and is located at **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample2/src/Sample2.java**.

To compile and run sample 2:

1. Go to the `sample2` directory. On Microsoft® Windows® systems, you must open a Command Prompt window.
2. Set the `CLASSPATH` variable to the IBM® ILOG® JViews Diagrammer library and the current directory.

On Microsoft Windows systems	`.;<installdir>\jviews-diagrammer86\lib\jviews-diagrammer-all.jar;<installdir>\jviews-framework86\lib\jviews-framework-all.jar;<installdir>\jlm`
On UNIX® systems	`.:<installdir>/jviews-diagrammer86/lib/jviews-diagrammer-all.jar:<installdir>/jviews-framework86/lib/jviews-framework-all.jar:<installdir>/jlm`

   The `jlm` directory, which contains the license keys (the file `keys.jlm`), must be in the class path.

3. Compile the application:

```
javac -d . src/Sample2.java
```

4. Run the application:

```
java Sample2
```

The `Sample2.java` file contains the following code:

```
// the JViews SDM Utilities
import ilog.views.sdm.util.*;
// the Diagrammer Framework
import ilog.views.diagrammer.*;
// the Java AWT package
import java.awt.*;
// The Java Net package
import java.net.*;
// the Java Swing package
import javax.swing.*;

public class Sample2
{
  public static void main(String[] arg)
```

```
{
  SwingUtilities.invokeLater(new Runnable() {
    public void run() {
      // Create the diagram component
      IlvDiagrammer diagrammer = new IlvDiagrammer();

      // Create the mutable style sheet for temporary layout
      // parameter settings
      IlvSDMMutableStyleSheet styleSheet =
        new IlvSDMMutableStyleSheet(
          diagrammer.getEngine(), true, false);

      // Change diagram parameters.
      diagrammer.setSelectMode(false);
      diagrammer.setScrollable(false);
      diagrammer.setEditingAllowed(true);
      diagrammer.getView().setBackground(Color.white);
      diagrammer.getView().setForeground(SystemColor.windowText);

      // The name of the XML file containing the model data
      String xmlFileName = "data/Sample.xml";

      // The name of the CSS file containing the main style sheet
      String cssFileName = "data/Sample.css";

      // Load the main style sheet
      try {
        diagrammer.setStyleSheet(new URL("file:" + cssFileName));
      } catch (Exception e) {
        System.out.println("could not read " + cssFileName);
        return;
      }

      // Cascade the main style sheet with the mutable style sheet.
      // The mutable style sheet holds the temporary style changes
      // that are not statically stored in the main style sheet.
      try {
        diagrammer.getEngine().setStyleSheets(1,
           styleSheet.toString());
      } catch (Exception e) {
        System.out.println("could not load the style sheet");
      }

      // Load the sample data file
      try {
        diagrammer.setDataFile(new URL("file:" + xmlFileName));
      } catch (Exception e) {
        System.out.println("could not read " + xmlFileName);
        return;
      }

      // Change some layout parameters in the mutable style sheet
      styleSheet.setAdjusting(true);
      try {
          // enable graph layout
          styleSheet.setDeclaration("GraphLayout", "enabled", "true");
          // use Tree Layout
          styleSheet.setDeclaration("GraphLayout", "graphLayout", "Tree");
          // use flow direction towards bottom
          styleSheet.setDeclaration("GraphLayout", "flowDirection", "Bottom")
;
          // use orthogonal link style
          styleSheet.setDeclaration("GraphLayout", "globalLinkStyle",
            "ORTHOGONAL_STYLE");
        } finally {
          // This completes the adjusting session: it validates the new
          // declarations and performs layout as neceesary
          styleSheet.setAdjusting(false);
        }

        // A Swing Frame to display
        JFrame frame = new JFrame("Layout Sample");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Put the manager view inside the Swing Frame and show it.
        frame.getContentPane().add(diagrammer);

        frame.setSize(600, 600);
        frame.setVisible(true);
      }
    });
  }
}
```

Note: The bold font indicates the differences between `Sample2` and `Sample1`.

If you need to change layout parameters inside the program, call the method `setDeclaration`. For example, to set the flow direction to `Bottom`, call:

```
styleSheet.setDeclaration("GraphLayout", "flowDirection", "Bottom");
```

This is equivalent to changing the style sheet declaration to:

```
GraphLayout {
  flowDirection : "Bottom";
}
```

Whenever the declaration of the style sheet changes outside an adjustment session, or when an adjustment session ends, the layout is automatically applied.

An adjustment session encapsulates a sequence of declaration changes as follows:

```
styleSheet.setAdjusting(true);
```

```
try {
  ... declaration changes ...
} finally {
  styleSheet.setAdjusting(false);
}
```

These changes are delayed until the method `styleSheet.setAdjusting(false)` is called. Only then will the layout be applied. Adjustment sessions are an efficient way to apply a large number of declaration changes.

Running the sample application that uses the graph layout API

Using the graph layout API directly may be necessary in advanced applications for temporary parameter settings.

In this case, unlike the case with mutable style sheets, the diagram component forgets all direct parameter settings when it triggers a reload of the style sheet. Also, when the layout parameters are changed through the layout API, the layout is not performed automatically and must be triggered explicitly.

An application that accesses the graph layout instance directly is illustrated below. It loads a style sheet that specifies a tree layout (see *Sample CSS file for graph layout*) and accesses the tree layout instance to change layout parameters. The source code of the application is named `Sample3.java` and it is located at **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample3/src/Sample3.java**.

To compile and run sample 3:

1. Go to the `sample3` directory. On Microsoft® Windows® systems, you must open a Command Prompt window.
2. Set the `CLASSPATH` variable to the IBM® ILOG® JViews Diagrammer library and the current directory.

On Microsoft Windows systems	`.;<installdir>\jviews-diagrammer86\lib\jviews-diagrammer-all.jar;<installdir>\jviews-framework86\lib\jviews-framework-all.jar;<installdir>\jlm`
On UNIX® systems	`.:<installdir>/jviews-diagrammer86/lib/jviews-diagrammer-all.jar:<installdir>/jviews-framework86/lib/jviews-framework-all.jar:<installdir>/jlm`

   Note that the `jlm` directory, which contains the license keys (the file `keys.jlm`), must be in the class path.

3. Compile the application:

```
javac -d . src/Sample3.java
```

4. Run the application:

```
java Sample3
```

The `Sample3.java` file contains the following code:

```
// the JViews Graphic Framework
import ilog.views.*;
// The JViews Tree Layout
import ilog.views.graphlayout.tree.*;
```

```
// the Diagrammer Framework
import ilog.views.diagrammer.*;
// the Java AWT package
import java.awt.*;
// The Java Net package
import java.net.*;
// the Java Swing package
import javax.swing.*;

public class Sample3
{
  public static final void main(String[] arg)
  {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        // Create the diagram component
        IlvDiagrammer diagrammer = new IlvDiagrammer();

        // Change diagram parameters.
        diagrammer.setSelectMode(false);
        diagrammer.setScrollable(false);
        diagrammer.setEditingAllowed(true);
        diagrammer.getView().setBackground(Color.white);
        diagrammer.getView().setForeground(SystemColor.windowText);

        // The name of the XML file containing the model data
        String xmlFileName = "data/Sample.xml";

        // The name of the CSS file containing the main style file
        String cssFileName = "data/Sample.css";

        // Load the style sheet
        try {
          diagrammer.setStyleSheet(new URL("file:" + cssFileName));
        } catch (Exception e) {
          System.out.println("could not read " + cssFileName);
          return;
        }

        // Load the sample data file
        try {
          diagrammer.setDataFile(new URL("file:" + xmlFileName));
        } catch (Exception e) {
          System.out.println("could not read " + xmlFileName);
          return;
        }

        // A Swing Frame to display
        JFrame frame = new JFrame("Layout Sample");

        // Put the manager view inside the Swing Frame and show it.
        frame.getContentPane().add(diagrammer);

        frame.setSize(600, 600);
        frame.setVisible(true);

        // The style sheet specifies Tree layout, therefore the current
        // layout instance has type IlvTreeLayout
        IlvTreeLayout layout = (IlvTreeLayout) diagrammer.getEngine().
          getNodeLayoutRenderer().getGraphLayout();

        // Change some layout parameters on the layout instance directly.
        layout.setFlowDirection(IlvDirection.Bottom);
        layout.setGlobalLinkStyle(IlvTreeLayout.ORTHOGONAL_STYLE);

        // Changing the layout parameters in this way does not perform
        // a layout automatically. Therefore layout is called
        // explicitely.
        IlvDiagrammer fdiagrammer = diagrammer;
        if (fdiagrammer.isNodeLayoutAvailable()) {
          // Perform the layout of all nodes
          fdiagrammer.layoutAllNodes();
          // Fit the view to show the entire graph
          fdiagrammer.fitToContents();
        }
      }
    });
  }
}
```

Note: The bold font indicates the differences between `Sample3` and `Sample1`.

Using layout algorithms through the graph layout API

Explains how to use layout algorithms through the graph layout API.

In this section

Using layout algorithms on graphers

Graphers (class `IlvGrapher`).store nodes and links and provide the minimal infrastructure that is necessary to perform a graph layout.

To use the layout algorithms provided by the graph layout package:

1. Create a grapher object (`IlvGrapher`) and fill it with nodes and links.
2. Create an instance of the layout algorithm (any subclass of `IlvGraphLayout`).
3. Declare a handle for the corresponding layout report. The layout report is an object in which the layout algorithm stores information about its behavior. For details, see *Performing a layout*.
4. Attach the grapher to the layout instance.
5. Modify the default settings for the layout parameters, if necessary.
6. Call the `performLayout` method inside a `try` block.
7. Read and display information from the layout report.
8. Catch the exceptions.
9. When the layout instance is no longer needed, detach the grapher from it.

Running the sample application that uses the graph layout API

You can use this application as an example to get started with the layout algorithms of the graph layout package.

The example uses the Tree layout, but most of the principles apply to any of the other layouts. The source code of the application is named `Sample4.java` and it is located at **<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample4/src/Sample4.java**.

To compile and run sample 4:

1. Go to the `sample4` directory. On Microsoft® Windows® systems, you must open a Command Prompt window.
2. Set the `CLASSPATH` variable to the IBM® ILOG® JViews Diagrammer library and the current directory.

On Microsoft Windows systems	`.;<installdir>\jviews-diagrammer86\lib\jviews-diagrammer-all.jar;<installdir>\jviews-framework86\lib\jviews-framework-all.jar;<installdir>\jlm`
On UNIX® systems	`.:<installdir>/jviews-diagrammer86/lib/jviews-diagrammer-all.jar:<installdir>/jviews-framework86/lib/jviews-framework-all.jar:<installdir>/jlm`

   The `jlm` directory, which contains the license keys (the file `keys.jlm`), must be in the class path.

3. Compile the application:

```
javac -d . src/Sample4.java
```

4. Run the application:

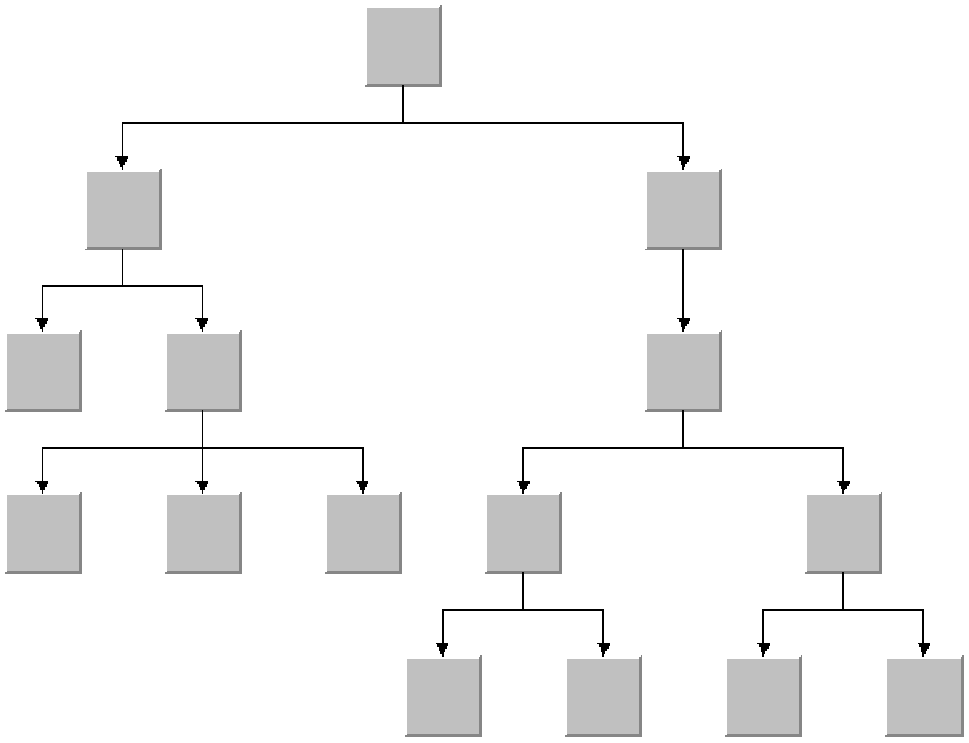
```
java Sample4
```

The `Sample4.java` file contains the following code:

```
// the  JViews Graphic Framework
import ilog.views.*;
// the JViews Graph Layout Framework
import ilog.views.graphlayout.*;
// the  JViews Tree Layout
import ilog.views.graphlayout.tree.*;
// the Java AWT package
import java.awt.*;
// the Java Swing package
import javax.swing.*;
```

```
public class Sample4
{
  public static void main(String[] arg)
  {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
        // Declare a handle for the layout instance
        IlvTreeLayout layout = new IlvTreeLayout();

        // Create the grapher instance
        IlvGrapher grapher = new IlvGrapher();

        // Create the manager view instance
        IlvManagerView view = new IlvManagerView(grapher);

        // Change view parameters.
        view.setBackground(Color.white);
        view.setKeepingAspectRatio(true);

        // The name of the IVL file containing the graph data
        String fileName = "data/Sample.ivl";

        // Fill the grapher with nodes and links from a JViews IVL file.
        // Alternatively, the nodes and links could be created
        // programmatically.
        try {
          grapher.read(fileName);
        } catch (Exception e) {
          System.out.println("could not read " + fileName);
          return;
        }

        // A Swing Frame to display
        JFrame frame = new JFrame("Layout Sample");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

        // Put the manager view inside the Swing Frame and show it.
        frame.getContentPane().add(view);

        frame.setSize(600, 600);
        frame.setVisible(true);

        // perform the layout in the AWT thread
        performLayout(grapher, view, layout);
      }
    });
  }

  /**
   * Perform the layout.
   */
  private static void performLayout(IlvGrapher grapher, IlvManagerView view,
IlvTreeLayout layout)
  {
    // Attach the grapher to the layout instance
    layout.attach(grapher);

    // Change some layout parameters
    layout.setFlowDirection(IlvDirection.Bottom);
    layout.setGlobalLinkStyle(IlvTreeLayout.ORTHOGONAL_STYLE);

    try {
      // Perform the layout and get the layout report
      IlvGraphLayoutReport layoutReport = layout.performLayout();

      int code = layoutReport.getCode();

      // Print information from the layout report (optional)
      System.out.println("Done in " +
                         layoutReport.getLayoutTime() +
                         " millisec., return code = " +
                         code + " (" +
                         layoutReport.codeToString(code) + ")");
    }

    // Catch the exceptions
    catch (IlvGraphLayoutException e) {
      System.out.println(e.getMessage());
    }

    finally {
      // Fit the view to show the entire graph
      view.fitTransformerToContent();
      // Redraw the grapher
      grapher.reDraw();
    }

    // Detach the grapher from the layout instance
    layout.detach();

  }
}
```

The sample Java™ application sample 4 produces the following graph.


Output from sample 4

Basic concepts

Explains some basic concepts and background information to help in using the Graph Layout algorithms.

In this section

Graph layout in IBM® ILOG® JViews Diagrammer

In IBM® ILOG® JViews, graphs are instances of the class `IlvGrapher` These instances are called graphers. Nodes, which are instances of `IlvGraphic`, and links, which are instances of `IlvLinkImage`, "know" how to draw themselves. Nodes can be "placed" interactively or by code. To lay out a grapher to obtain a readable drawing, just compute and assign appropriate coordinates to the nodes. In some cases, you may also need to modify the shape of the links. The main task of the Graph Layout algorithms is to provide support for computing node and link coordinates—that is, laying out the graph—fully automatically.

IBM® ILOG® JViews Diagrammer provides a Swing component that encapsulates a grapher and the view that displays the grapher. JViews Diagrammer uses a model-view architecture, that is, the application objects must be provided as an SDM model, and the grapher is filled with corresponding nodes and links that are displayed in the view. The nodes and links are styled according to a *style file*. A style file is a CSS file, where CSS stands for Cascading Style Sheets. It has the file name extension `.css`. A style file specifies which instances of `IlvGraphic` and `IlvLinkImage` must be created for the diagram, and which layout algorithms and layout parameters must be used to calculate the coordinates of the nodes and links.

There are various ways to use the Graph Layout algorithms:

- **In a diagram component:** You specify the graph layout in CSS format. The diagram component loads this specification and automatically applies the graph layout when necessary.

- **In an application that uses IBM® ILOG® JViews graphers:** Instead of using style files, you access the API of the graph layout classes directly. This is suitable for applications that do not need the model-view architecture or styling. Using the graph layout classes directly is more powerful because you have access to internal details of the graph layout classes that are not available in style files. However, using the graph layout API directly is slightly more complex. For example, the layout algorithm is generally not applied automatically, you have to call it explicitly.

- **With your own graph data structures:** It is not required to use a diagram component to display a diagram, or to use an IBM® ILOG® JViews grapher to represent a graph. If you have implemented your own data structures or if you use some other third-party data structures that represent a graph, it is still possible to use graph layout algorithms by providing an adapter between your data structures and the graph model. This more complex application of the graph layout package is explained in *Laying out a non-JViews grapher*.

This documentation shows you how to style graph layouts in a diagram component by using style sheets (CSS) and how to program graph layout classes in Java™ , which is necessary when you cannot use a diagram component with styling capabilities. In most cases, both ways are illustrated in parallel even though, practically, the most common situation is to use either one or the other.

Here are two examples in ascending order of difficulty.

Setting the offset between nodes

In CSS
Specify:

```
GraphLayout {
     horizontalNodeOffset: "20";
}
```

In Java
Call:

```
layout.setHorizontalNodeOffset(20);
```

Setting the position of a node

Example of setting the position of a node
In CSS
Specify:

```
GraphLayout {
    position          : "10,10";
    rootPosition      : "false";
}
```

In Java
Call the `setPosition` method as follows:

```
layout.setPosition(new IlvPoint(10, 10), false);
```

These examples mean two things:

♦ If you implement an application that uses IBM® ILOG® JViews Diagrammer or the SDM engine and therefore is based on styling, you should specify the graph layout parameters in a style sheet (CSS file), and you can set the position of the top left corner by adding the above lines to your CSS file.

♦ If you implement an application that does not use JViews Diagrammer or the SDM engine, for example an application that works directly on instances of `IlvGrapher`, or on your own graph data structures, then you cannot use style sheets and you must access the graph layout instances directly as illustrated in the Java code above.

In the first case, you specify a CSS file, in the second, you write Java code.

However, it may be necessary to mix style sheets and Java code in some cases, for example when you implement your own graph layout renderer. But this case should be left to expert users who are familiar with the entire SDM and JViews Framework architectures. See Using and adding renderers *in JViews Diagrammer SDK* and *Writing a new layout renderer to clip links*.

Using graph layout in a diagram component

In a diagram component, operations such as attaching or detaching a graph layout instance take place automatically.

Controlling layout renderers by style sheets

A diagram component (an instance of `IlvDiagrammer`) contains various renderers that control the graphic display of SDM model objects. The following three renderers are relevant for graph layout:

- **The Graph Layout Renderer** is also called the *Node Layout Renderer* because in some configurations, it only places the nodes without routing the links.
- **The Link Layout Renderer** routes the links, unless the Graph Layout Renderer does this already.
- **The Label Layout Renderer** places labels at nodes and links.

The renderers and the general CSS format are documented in Using CSS syntax in the style sheet

The style sheet controls all the renderers. For each renderer, there is a section in the style sheet.

The following is a summary:

```
GraphLayout {
...//settings relevant for the Graph Layout Renderer...
}
LinkLayout {
...//settings relevant for the Link Layout Renderer...
}
LabelLayout {
...//settings relevant for the Label Layout Renderer...
}
```

You can also specify layout parameters for individual nodes and links by specifying a rule that selects the node or link. Here are a few examples:

```
#obj13 {
... //settings relevant for the application object with ID "obj13" ...
}
node.participant {
... //settings relevant for the node of type participant ...
}
node.selected {
... //settings relevant for all selected nodes ...
}
node.participant[abc="true"] {
... //settings relevant for the node of type participant whose property "abc"
```

```
 is true ...
}
```

Note: Layout parameter names in the GraphLayout, LinkLayout, and LabelLayout sections always start with a lowercase letter. Layout parameter names in the node or link rules always start with an uppercase letter.

Loading a style file

A CSS specification file becomes active when it is loaded into the SDM engine. At this point, the graph layout is performed.

Example of loading a style file
For example, if you have a diagram component of type `IlvDiagrammer` use the following:

In Java™

```
try {
      diagrammer.setStyleSheet(new URL("file:styleSheet.css"));
} catch (Exception e) {
      ...
}
```

When the style sheet is loaded and the layout renderers are enabled, any change to the data model automatically updates the layout to reflect the changes.

Accessing the layout renderers

The layout renderers can be accessed from the diagram component either in Java or in CSS.

Example of accessing the layout renderers
In CSS
Use the following syntax:

```
GraphLayout {
    enabled:true;
    ...
```

In Java
The call

```
diagrammer.getEngine().getNodeLayoutRenderer()
```

returns the renderer controlled by the `GraphLayout` section of the style sheet.

The call

```
diagrammer.getEngine().getLinkLayoutRenderer()
```

returns the renderer controlled by the `LinkLayout` section of the style sheet.

```
diagrammer.getEngine().getLabelLayoutRenderer()
```

returns the renderer controlled by the `LabelLayout` section of the style sheet.

You can enable or disable each individual renderer. If the renderer is disabled, changes to the data model will have no effect on the renderer. For example, after the call

```
diagrammer.getEngine.getNodeLayoutRenderer().setEnabled(false);
```

a change to the data model will not trigger any new layout.

Performing layout explicitly

When a style sheet is loaded, you can also trigger a layout explicitly.

Node layout

The call

```
diagrammer.performNodeLayout();
```

performs a graph layout through the node layout renderer.

Important: Some layout algorithms place only nodes, while others place nodes and links at the same time. If the style sheet specifies a layout that places nodes and links at the same time, this call routes the links as well.

Link layout

The call

```
diagrammer.performLinkLayout();
```

performs a link layout through the link layout renderer. The link layout does not move any node, it only reshapes the links. Separating between node layout and link layout is useful for those layout algorithms that cannot do both tasks.

Label layout

The call

```
diagrammer.performLabelLayout();
```

performs a label layout through the label layout renderer. It positions the labels of graphic nodes and graphic links that are subclasses of `IlvGeneralNode`, `IlvGeneralLink` `IlvSDMCompositeNode`, and `IlvSDMCompositeLink`.

Accessing graph layout instances

Graph layout instances can be accessed from the layout renderers by the following methods:

```
nodeLayoutRenderer.getGraphLayout();
```

```
linkLayoutRenderer.getGraphLayout();
```

```
labelLayoutRenderer.getLabelLayout();
```

Writing a new layout renderer to clip links

This example shows how to implement and install a new layout renderer in order to perform link clipping. This involves combining Java code and style rules that fit together, as follows:

♦ Create a new graph layout renderer in Java code.

♦ Install the new graph layout renderer by adding style rules in the style sheet.

See
<installdir>/jviews-diagrammer86/codefragments/graphlayout-diagrammer/graphlayout4.

In Java code

To implement the link clipping example, you need to write a subclass of `IlvGraphLayoutRenderer` called `LinkClippingRenderer`. The class declaration for the new renderer is as follows:

```
public class LinkClippingRenderer extends IlvGraphLayoutRenderer
```

The new renderer has a property called `clipping` which enables or disables link clipping.

The clipping property and its methods shows the property declaration and the methods to test it and set it.

The clipping property and its methods

```
  private boolean clipping = false;

  public boolean isClipping()
  {
    return clipping;
  }

  public void setClipping(boolean clipping)
  {
    this.clipping = clipping;
  }
```

To enable the link-clipping feature, call the method `setLinkClipInterface(ilog.views.graphlayout.IlvLinkClipInterface)`. In the new renderer, this method is called in an overridden version of the method `prepareRendering(ilog.views.sdm.IlvSDMEngine)` , which is called before the graph is rendered.

The new code for the prepareRendering method shows the code for the `prepareRendering` method in the `LinkClippingRenderer` class.

The new code for the prepareRendering method

```
public void prepareRendering(IlvSDMEngine engine)
  {
    super.prepareRendering(engine);
    if(clipping && getGraphLayout().supportsLinkClipping()){
     getGraphLayout().setLinkClipInterface(new ShapeLinkClipInterface(engine)
);
    }
  }
```

The link clip interface is set to an instance of a class called `ShapeLinkClipInterface`. This class computes the intersection point of the link on the node shape. Look at the source code of the example to see how the computation works.

The method `removeAll(ilog.views.sdm.IlvSDMEngine)` is also overridden. This method is called after the graph is rendered (and after the graph layout is performed), to clean up the link clip interface.

The new code for the removeAll method shows the code for the `removeAll` method in the `LinkClippingRenderer` class.

The new code for the removeAll method

```
public void removeAll(IlvSDMEngine engine)
  {
    super.removeAll(engine);
    if(getGraphLayout().supportsLinkClipping()){
      getGraphLayout().setLinkClipInterface(null);
    }
  }
```

This method calls the superclass method and then clears the link clip interface.

The Enable/Disable Link Clipping button in the toolbar is bound to an action that calls the `LinkClippingRenderer.setClipping()` method to enable or disable link clipping.

You must re-create the graph after processing the action for the Enable/Disable Link Clipping button. *Re-creating a graph* shows the code line to add.

Re-creating a graph

```
diagrammer.getEngine().loadData();
```

Note that you could also use the `IlvSDMMutableStyleSheet` class to set the layout parameters dynamically.

In CSS

When you have written a new renderer, you need to configure your diagram to use this renderer instead of the supplied graph layout renderer.

Setting a new class to use instead of a supplied renderer shows the extra declaration in the SDM rule in the link-clipping style sheet.

Setting a new class to use instead of a supplied renderer

```
SDM {
  Renderers : "GraphLayout=LinkClippingRenderer";
  GraphLayout : true;
}
```

This declaration tells the SDM engine to use the new class instead of the `IlvGraphLayoutRenderer` class as the GraphLayout renderer.

Style rule that customizes graph layout with link clipping shows the GraphLayout rule with the extra declarations that are necessary for link clipping to work correctly.

Style rule that customizes graph layout with link clipping

```
GraphLayout {
  graphLayout : "Hierarchical";
  flowDirection : "Right";
  globalLinkStyle : "STRAIGHT_LINE_STYLE";
  connectingLinksToShape : "false";
  connectorStyle : "CLIPPED_PINS";
}
```

The GraphLayout rule contains the parameters for the graph layout renderer and for the graph layout algorithm. It contains three additional declarations for link clipping:

♦ The first extra declaration tells the Hierarchical layout not to reshape links orthogonally, so that the effect of link clipping is more visible.

♦ The second extra declaration tells the graph layout renderer to connect links to the center of the nodes, instead of connecting them to the border of the nodes. The link clipping algorithm then clips each link at the crossing point on the border.

♦ The third extra declaration tells the Hierarchical layout to connect the links using clipped pins. This declaration overrides the `connectorStyle` declaration in the graph-layout style sheet.

Further information

You can find information on how to use the Graph Layout with IBM® ILOG® JViews Diagrammer in *Getting started with graph layout*.

Further information is in in The GraphLayout renderer in *JViews Diagrammer SDK*.

You can also find examples of specification in CSS for graph layout in:
<installdir>/jviews-diagrammer86/codefragments/graphlayout/sample1/data/Sample.css;

More graph layout examples are supplied in:
<installdir>/jviews-diagrammer86/codefragments/graphlayout-diagrammer.

A detailed description of the renderer API is available in the online *Java API Reference Documentation*:

- `IlvDiagrammer`
- `IlvGraphLayoutRenderer`
- `IlvLinkLayoutRenderer`
- `IlvLabelLayoutRenderer`

Using the graph layout API

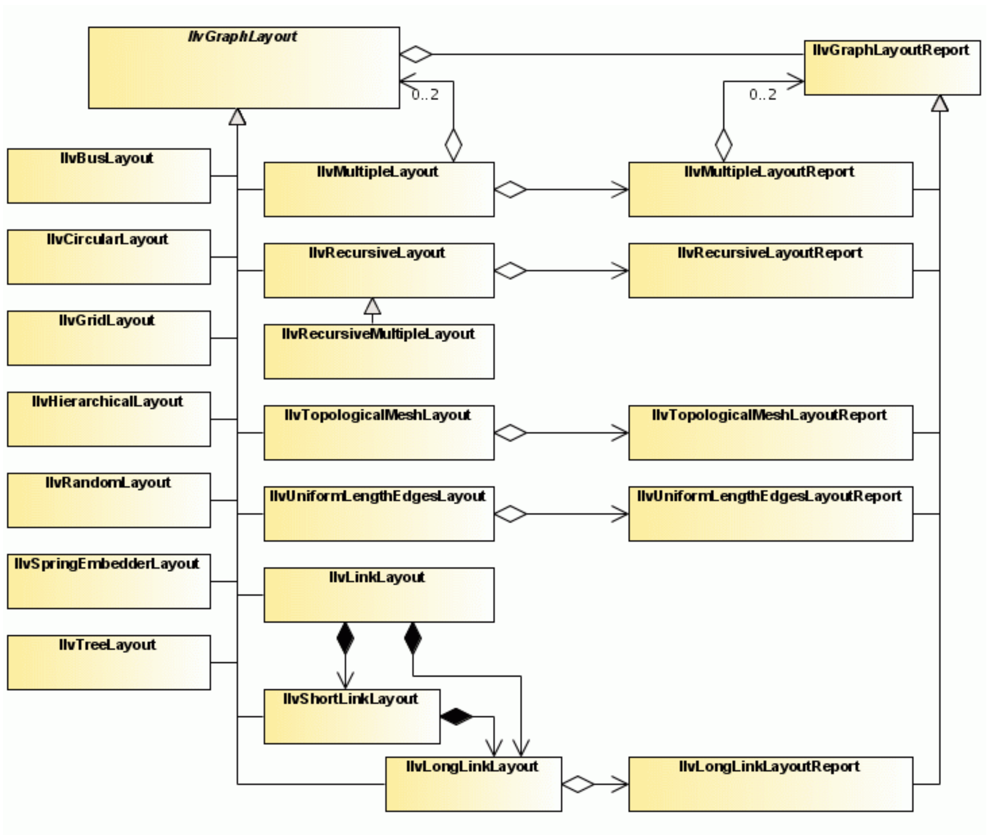
In an application that works directly on an `IlvGrapher` object, operations such as attaching or detaching a graph layout instance must be performed explicitly.

Nodes are instances of the class `IlvGraphic` and links are instances of the class `IlvLinkImage`.

The base class: IlvGraphLayout

The `IlvGraphLayout` class is the base class for all layout algorithms. This class is an abstract class and cannot be used directly. You must use one of its subclasses: `IlvHierarchicalLayout`, `IlvTreeLayout`, `IlvUniformLengthEdgesLayout`, `IlvTopologicalMeshLayout`, `IlvLinkLayout`, `IlvRandomLayout`, `IlvBusLayout`, `IlvCircularLayout`, `IlvGridLayout`. You can also create your own subclasses to implement other layout algorithms. See *Defining your own type of layout*.

Despite the fact that only subclasses of `IlvGraphLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by the subclasses. And, of course, you will need to understand it if you subclass it yourself.



The Class IlvGraphLayout and its subclasses and relationships to layout reports

Instantiating a subclass of IlvGraphLayout

The class `IlvGraphLayout` is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvLinkLayout layout = new IlvLinkLayout();
```

Attaching/detaching a grapher

You must attach the grapher before performing the layout. The following method, defined on the class `IlvGraphLayout`, allows you to specify the grapher you want to lay out:

```
void attach(IlvGrapher grapher)
```

For example:

```
...
IlvGrapher grapher = new IlvGrapher();
/* Add nodes and links to the grapher here */
layout.attach(grapher);
```

The `attach` method does nothing if the specified grapher is already attached. If a different grapher is attached, this method first detaches this old grapher, then attaches the new one. You can obtain the attached grapher using the method `getGrapher()`. If the grapher is attached in this way, a default graph model is created internally. For details on the graph model, see *Using the Graph Model*. The attached graph model can be obtained by:

```
IlvGraphModel graphModel = layout.getGraphModel();
```

Warning: You are not allowed to attach a default model created internally to any other layout instance, nor to use it in any way once it has been detached from the layout instance. For details, see *Using the class IlvGrapherAdapter*.

After layout, when you no longer need the layout instance, you should call the method

```
void detach()
```

If the `detach` method is not called, some objects may not be garbage-collected. This method also performs clean-up operations on the grapher, such as removing properties that may have been added to the grapher objects by the layout algorithm. It also removes layout parameters of nodes and links.

Note: A layout instance should stay attached as long as its layout parameters are relevant for the grapher. Only when the layout parameters, and therefore the entire layout instance, become irrelevant for this grapher should it be detached.

Performing a layout

The `performLayout` method starts the layout algorithm using the currently attached grapher and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the layout algorithm.

```
IlvGraphLayoutReport performLayout()
```

```
IlvGraphLayoutReport performLayout(boolean force, boolean redraw)
```

The first version of the method simply calls the second one with a `false` value for the first argument and a `true` value for the second argument. If the argument `force` is `false`, the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the grapher or any of the parameters have been changed since the last time the layout was successfully performed. A “change” can be any of the following:

- ♦ Nodes or links have been added or removed.
- ♦ Nodes or links have been moved or reshaped.
- ♦ The value of a layout parameter has been modified.
- ♦ The transformer of a manager view (`IlvManagerView`) of the grapher has changed.

Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. Note that if the argument `force` is passed as `true`, the verification is no longer performed.

The argument `redraw` determines whether a redraw of the graph is requested. For details, see *Redrawing the grapher after layout*.

The protected abstract method `layout(boolean redraw)` is then called. This means that the control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the nodes to new positions and/or reshapes the links.

The `performLayout` method returns an instance of `IlvGraphLayoutReport` (or of a subclass) that contains information about the behavior of the layout algorithm. It tells you whether the algorithm performed normally, or whether a particular, predefined case occurred. (For a more detailed description of the layout report, see *Using a graph layout report*.)

Note that the layout report that is returned can be an instance of a subclass of `IlvGraphLayoutReport` depending on the particular subclass of `IlvGraphLayout` you are using. For example, it will be an instance of `IlvTopologicalMeshLayoutReport` if you are using the class `IlvTopologicalMeshLayout`. Subclasses of `IlvGraphLayoutReport` are used to store layout algorithm-dependent information.

You must call the method `performLayout` inside a `try` block because it can throw an exception. The exception can be of the type `IlvGraphLayoutException` or one of its subclasses, `IlvInappropriateGraphException` and `IlvInappropriateLinkException`. The first indicates internal problems in the layout algorithm or an unexpected situation. The second exception indicates that a particular grapher cannot be laid out with the layout algorithm. For example, the Topological Mesh Layout cannot be used on a tree). The third exception indicates that a particular type of link (or link connector) cannot be used for this layout. The recommended type of link is `IlvPolylineLinkImage` or `IlvSplineLinkImage` (or subclasses). For layouts that do not reshape the links by adding intermediate points, the class `IlvLinkImage` can also be used. See *Layout exceptions* for details and solutions.

Further information

You can find more information about the class `IlvGraphLayout` in the following sections:

- ♦ *Base class parameters and features* contains the methods that are related to the customization of the layout algorithms.
- ♦ *Using event listeners* tells you about the layout event listener mechanism.
- ♦ *Defining your own type of layout* tells you how to implement new subclasses.

For details on `IlvGraphLayout` and other graph layout classes, see the Java™ *API Reference Documentation.*

Layout algorithms

Describes the IBM® ILOG® JViews Graph Layout algorithms.

In this section

Force-directed or Uniform Length Edges Layout (ULEL)

Describes the *Force-directed layout* or *Uniform Length Edges Layout* algorithm (class `IlvUniformLengthEdgesLayout` from the package `ilog.views.graphlayout.uniformlengthedges`).

Tree Layout (TL)

Describes the *Tree Layout* algorithm (class `IlvTreeLayout` from the package `ilog.views.graphlayout.tree`).

Hierarchical Layout (HL)

Describes the *Hierarchical Layout* algorithm (class `IlvHierarchicalLayout` from the package `ilog.views.graphlayout.hierarchical`).

Link layout (LL)

Describes the *Link Layout* algorithm (class `IlvLinkLayout` from the package `ilog.views.graphlayout.link`).

Random layout (RL)

Describes the *Random Layout* algorithm (class `IlvRandomLayout` from the package `ilog.views.graphlayout.random`).

Bus layout (BL)

Describes the *Bus Layout* algorithm (class `IlvBusLayout` from the package `ilog.views.graphlayout.bus`).

Circular layout (CL)

Describes the *Circular Layout* algorithm (class `IlvCircularLayout` from the package `ilog.views.graphlayout.circular`).

Grid layout (GL)

Describes the *Grid Layout* algorithm (class `IlvGridLayout` from the package `ilog.views.graphlayout.grid`).

Layout exceptions

Describes the exceptions that may be thrown when a layout is applied to the wrong type of graph or the wrong type of link and explains how to catch them.

Overview of graph layout information

For each layout, the information given includes:

- Code samples
- Which types of graphs the layout may be used for
- The application domains, features, and limitations
- A brief description of the algorithm
- The specification in CSS
- The specification in Java™ code
- The generic features and parameters, as well as the specific parameters of the algorithm

Determining the appropriate layout algorithm

When using the graph layout package, you need to determine which of the ready-to-use layout algorithms is appropriate for your particular needs. Some layout algorithms can handle a wide range of graphs. Others are designed for particular classes of graphs and will give poor results or will reject graphs that do not belong to these classes. For example, a Tree Layout algorithm is designed for tree graphs, but not cyclic graphs. Therefore, it is important to lay out a graph using the appropriate layout algorithm.

The following tables can help you determine which of the layout algorithms is best suited for a particular type of graph.

♦ Across the top of the table are various classifications of different types of graphs.

♦ The layout algorithms appear on the left side of the tables.

♦ Table cells containing illustrations indicate when a layout algorithm is applicable for a particular type of graph.

By identifying the general characteristics of the graph you want to lay out, you can see from the tables whether a layout algorithm is suited for that particular type of graph.

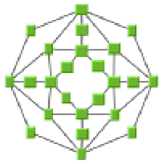
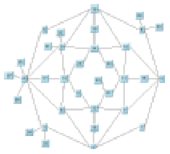
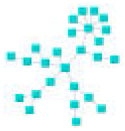
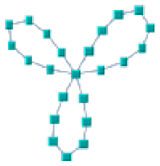
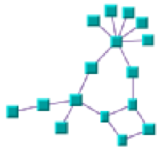
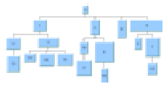
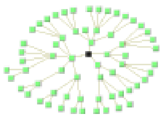
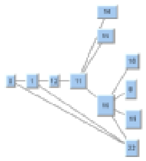
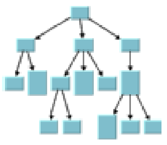
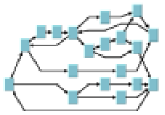
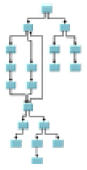
For example, if you know that the structure of the graph is a tree, you can look at the Domain-Independent Graphs/Trees column to see which layout algorithms are appropriate. The Uniform Length Edges Layout, Tree Layout, and Hierarchical Layout could all be used. Use the illustrations in the table cells to help you further narrow your choice.

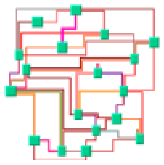
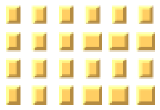
You can use the *Recursive layout* to control the layout of nested graphs (containing subgraphs and intergraph links). This is in particular useful if different layout styles should be applied to different subgraphs. The other layout algorithms such as, Tree Layout, and Hierarchical Layout treat only flat graphs (unless otherwise noted), that is, a specific layout instance is only able to lay out the nodes and links of the attached graph, but not the nodes and links of its subgraphs. The Recursive Layout allows you to specify which flat layout is used for which subgraph, and it traverses the entire nested graph recursively when applying the layout. As a result, the entire nested graph is laid out.

You can use the *Multiple layout* to combine several different layouts into one instance. In this case, they become *sublayouts* of the Multiple Layout instance.

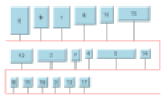
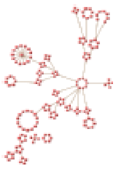
This is useful in particular for nested graphs when used in combination with the Recursive Layout. The Multiple Layout ensures that the normal layout, the routing of the intergraph links, and the layout of labels are applied in the correct order to a nested graph.

Layout algorithms and common types of graphs

Layout	Domain-Independent Graphs		
	Trees	Cyclic Graphs	Any Graph
Topological Mesh Layout			 Requires (semi)manual refinements
Uniform Length Edges Layout		 Preferable to avoid heavily interconnected graphs (large number of links)	
Tree Layout	 		
Hierarchical Layout			

Layout	Domain-Independent Graphs		
	Trees	Cyclic Graphs	Any Graph
Link Layout			
Grid Layout			 Note that the algorithm does not take into account the links between the nodes.
Recursive Layout			Nested graphs.
Multiple Layout			Combination of multiple different layout algorithms on the same graph (in particular for nested graphs).

Telecom-Oriented Representations

Layout	Telecom-Oriented Representations
Bus Layout	 For bus topologies
Circular Layout	 For interconnected ring/star topologies

Overview: layout algorithms in CSS

In a diagram component, the graph layout algorithm is usually specified by CSS.

The layout algorithms fall into two categories: node layout and link layout. There is also a label layout facility, which technically is not a graph layout algorithm. Label layout is described in section *Automatic label placement*.

Node layout algorithms

The node layout algorithms are presented in the following table.

Node layout algorithms

Bus layout	**Class name:** ilog.views.graphlayout.bus.IlvBusLayout
	
	Example: *BL - CSS Sample*
	
	Short description: A layout algorithm designed to display bus network topologies (that is, a set of nodes connected to a bus node)
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘Bus’;`
	`}`

Circular layout	**Class name:** ilog.views.graphlayout.circular.IlvCircularLayout
	
	Example: *CL samples*
	
	Short description: A layout algorithm that displays graphs representing interconnected ring and/or star network topologies
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘Circular’;`
	`}`

Grid layout	**Class name:** ilog.views.graphlayout.grid.IlvGridLayout **Example** GL Sample *In CSS*

	Short description: A layout algorithm that arranges the disconnected nodes of a graph in rows, in columns, or in the cells of a grid
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘Grid’;`
	`}`

Hierarchical layout	**Class name:** ilog.views.graphlayout.hierarchical.IlvHierarchicalLayout
	
	Example: *HL samples*
	
	Short description: A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘Hierarchical’;`
	`}`

Topological Mesh layout	**Class name:** ilog.views.graphlayout.topologicalmesh.IlvTopologicalMeshLayout
	
	Example: *TML samples*
	
	Short description: A layout algorithm that arranges cyclic (2-connected) undirected graphs in a regular fashion
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘TopologicalMesh’;`
	`}`

Tree layout	**Class name:** ilog.views.graphlayout.tree.IlvTreeLayout **Example:** *TL samples* **Short description:** A layout algorithm that arranges the nodes of a tree horizontally or vertically, starting from the root of the tree. The radial layout mode allows you to

	arrange the nodes of a tree in concentric circles around the root of the tree.
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘Tree’;`
	`}`

Uniform Length Edges layout	**Class name:** ilog.views.graphlayout.uniformlenghtedges.IlvUniformLengthEdgesLayout
	
	Example: *ULEL samples*
	
	Short description: A layout algorithm that can be used to lay out any type of graph and allows you to specify the length of the links. It applies a force-directed physical simulation that moves the nodes until all links have approximately the same length.
	
	CSS specification:
	
	`SDM {`
	`   GraphLayout: “true”;`
	`}`
	`GraphLayout {`
	`   graphLayout: ‘UniformLengthEdges’;`
	`}`

Besides these layout algorithms, there are many other layout facilities such as Random Layout, Multiple Layout, and Recursive Layout. The Random Layout is only a testing facility and plays no role in real applications. The Multiple Layout and the Recursive Layout cannot be specified in CSS, but they are used internally in the implementation of the graph layout Renderer.

Link layout algorithm

The link layout algorithm is presented in the following table.

Link Layout Algorithm

Link layout	**Class name:** ilog.views.graphlayout.link.IlvLinkLayout **Example:***LL samples* **Short description:** A layout algorithm that reshapes the links of a graph without moving the nodes. Several link modes are available, most importantly the orthogonal link mode. The Link Layout is not controlled by the graph layout renderer, but by the specialized link layout renderer. **CSS specification**:

	`SDM {`
	`LinkLayout: “true”;`
	`}`
	`LinkLayout {`
	`... further options...`
	`}`

Typical ways to choose a layout

Explains possible ways to choose a graph layout algorithm.

In this section

Choosing a layout algorithm

The choice of the appropriate algorithm for a graph can be done either by the end user at run time or by the programmer when he develops the application. This process can be *semiautomatic*, when the user is involved, or *automatic*, when the application does everything with no user intervention.

As a programmer of applications, you can choose *Semiautomatic layout* to involve the end user in the choice of the layout, or *Automatic layout*, in which case the application does everything with no end user action.

Semiautomatic layout

For applications using a semiautomatic layout, the choice of the layout algorithm is done by the end user. The application can provide a menu or some other way to select the layout algorithm.

In some cases, this may be an iterative process. The user may try different layout algorithms with different values for the parameters and/or may apply manual refinements to find the best layout. The application may possibly provide some help using textual explanations or by automatically checking the graph to find out to which class it belongs. For example, to detect whether the graph that has been attached to a layout instance is a tree, the `IlvGraphLayoutUtil` class provides the method:

```
static boolean IsTree(IlvGraphLayout layout, Object startNode)
```

For details on this method, see `IsTree(ilog.views.graphlayout.IlvGraphLayout, java.lang.Object)`. See also *Attaching/detaching a grapher*.

Automatic layout

If an automatic layout is needed, the choice of the layout algorithm can be:

♦ Chosen dynamically at run time by means of heuristics or rules to determine the appropriate layout algorithm depending on the structure and/or size of the graph

♦ Hard-coded if the developer knows what types of graphs will be used and can determine the appropriate layout algorithm.

Choosing the layout algorithm dynamically

If nothing is known about the graphs that the application will need to lay out, the developer can write a routine that automatically chooses the layout algorithm at run time. The following simple rules could be applied:

1. If the nodes of the graph cannot be moved (they are geo-positioned), use the Link Layout.
2. If the graph is a tree, use the Tree Layout.
3. Otherwise, use one of the layout algorithms that are the less restricted to a given graph category, especially the Uniform Length Edges Layout. (The preferred length of the links could also be computed with respect to the size of the nodes.)
4. If the graph is too large, apply a "divide-and-conquer" strategy. Cut the graph into several subgraphs and apply the layout separately to each subgraph. If the graph is disconnected, you can use the built-in support provided by the layout library to perform this task automatically. (See *Layout of connected components*.)
5. If the graph is nested, use the Recursive Layout algorithm that controls which subgraph is laid out by which (flat) sublayout. Use steps 1 to 4 to determine the sublayouts for the subgraphs. The Hierarchical Layout and the Tree Layout also have special modes for nested graphs, see *Recursive mode* and *Recursive layout*.

Hard-coding a layout at programming time

A special case occurs when the application will deal with only a small set of graphs that are known at the time the application is built. In this case, the layout can be performed at programming time. A possible step-by-step procedure may be the following:

1. Create each graph manually with a graph editor or in Java™ .
2. Try different layout algorithms and choose the best for each graph.
3. Apply manual refinements to the layout if needed.
4. Store the result of the layout by saving the coordinates of the nodes in the XML file of the diagram component, or by saving the graphers in `.ivl` files.
5. Provide these files with the application.

When the application is used, the `ivl` files will simply be loaded. There will be no need to perform the layout again since it is already done.

See also

Determining the appropriate layout algorithm

Hard-coding a layout at run time

If the choice of the layout algorithm is hard-coded, but the layout must be performed at run time because the graphs are not known at programming time, one possible step-by-step procedure for the choice of the appropriate layout algorithm may be the following:

1. Look at sample graphs for your domain.
2. Try to determine some generalities about the properties of the structure and the size of the graph (Is the graph cyclic? Is the graph a tree? Is the graph a combination of the two? What is the number of nodes and links in the graph?)
3. Pick an appropriate layout algorithm.
4. Try out the algorithm on one or more samples.

See also

Determining the appropriate layout algorithm

Generic parameters and features

Describes the support for generic features and parameters provided by each layout algorithm.

In this section

Support by algorithms of generic features and parameters

The graph layout generic features and parameters described in *Base class parameters and features* allow you to customize the behavior of the layout algorithms to meet specific needs and to perform useful operations such as saving the layout parameters in a file.

The following table indicates the generic features and parameters that are supported by each layout algorithm. These parameters are defined in the base class for all layout algorithms, `IlvGraphLayout`

Generic parameters supported by layout algorithms

Layout Algorithm Parameters	TML	ULEL	TL	HL	LL	RL	BL	CL	GL	Recursive Layout	Multiple Layout
Allowed Time	Yes	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes
Animation	Yes	Yes			Yes						
Fixed Links			Yes	Yes	Yes						
Fixed Nodes	Yes	Yes	Yes	Yes		Yes	Yes	Yes	Yes		
Layout of Connected Components	Yes	Yes	Yes	Yes			Yes	Yes			Yes
Layout Region	Yes	Yes				Yes	Yes	Yes	Yes		
Link Clipping	Yes	Yes	Yes	Yes			Yes	Yes			
Link Connection Box	Yes	Yes	Yes	Yes	Yes		Yes	Yes			
Spline Routing	Yes		Yes	Yes	Yes						
Memory Savings											
Percentage Complete		Yes		Yes	Yes		Yes			Yes	Yes
Random Generator Seed Value						Yes					
Save Parameters to File	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
Stop Immediately	Yes	Yes	Yes	Yes	Yes	Yes	Yes		Yes	Yes	Yes

Key

TML	Topological Mesh Layout
ULEL	Uniform Length Edges Layout

TL	Tree Layout
HL	Hierarchical Layout
LL	Link Layout
RL	Random Layout
BL	Bus Layout
CL	Circular Layout
GL	Grid Layout

Base class parameters and features

The `IlvGraphLayout` class defines a number of generic features and parameters. These features and parameters can be used to customize the layout algorithms.

Although the `IlvGraphLayout` class defines the generic parameters, it does not control how they are used by its subclasses. Each layout algorithm (that is, each subclass of `IlvGraphLayout`) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own layout algorithm by subclassing `IlvGraphLayout`, you decide whether you want to use the features and the way in which you are going to use them.

The `IlvGraphLayout` class defines the following generic features:

- *Allowed time*
- *Animation (ULEL)*
- *Automatic layout*
- *Coordinates mode*
- *Layout of connected components*
- *Layout region*
- *Link clipping*
- *Link connection box*
- *Spline routing*
- *Memory savings*
- *Percentage of completion calculation*
- *Preserve fixed links*
- *Preserve fixed nodes*
- *Random generator seed value*
- *Save parameters to named properties*
- *Stop immediately*
- *Use default parameters*

Support by algorithms of generic features and parameters provides a summary of the generic parameters supported by each layout algorithm. If you are using one of the subclasses provided with IBM® ILOG® JViews, check the documentation for that subclass to know whether it supports a given parameter and whether it interprets the parameter in a particular way.

Allowed time

Several layout algorithms can be designed to stop computation when a user-defined time specification is exceeded. This may be done for different reasons: as a security measure to avoid a long computation time on very large graphs or as an upper limit for algorithms that iteratively improve a current solution and have no other criteria to stop the computation.

Example of specifying allowed time
To specify that the layout is allowed to run for 60 seconds:

In CSS
Add to the `GraphLayout` section:

```
allowedTime: "60000";
```

In Java™
Call:

```
layout.setAllowedTime(60000)
```

The time is in milliseconds. The default value is 32000 (32 seconds).

If you subclass `IlvGraphLayout`, use the following method to know whether the specified time was exceeded:

```
boolean isLayoutTimeElapsed()
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsAllowedTime()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Animation

Some iterative layout algorithms can optionally redraw the graph after each iteration or step. This may create a pleasant animation effect and may be used to keep the user aware of the evolution of the layout computation by showing intermediate results (as a kind of progress bar). However, this increases the duration of the layout because additional redrawing operations need to be performed.

Example of specifying animation
To specify that the layout animation is enabled:

In CSS
Add to the `GraphLayout` section:

```
animate: "true";
```

In Java
Call:

```
layout.setAnimate(true)
```

Layout animation is disabled by default.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsAnimation()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Note: Layout animation shows the intermediate steps of the layout algorithm. If you need such animation only to show how the graph before layout transforms into the graph after layout, use the animation renderer of SDM instead. In this case, the intermediate steps of the layout algorithm are not shown, but after the layout is completed, the nodes and links are moved smoothly from the old positions to the new positions.

Automatic layout

For some layout algorithms, it may be suitable to have the layout automatically performed again after each change of the graph, that is, when a node or link moves, is added, or is removed. Automatic layout is most useful for link layouts, in a situation where the shape of the links must remain optimal after each editing action of the end-user. It also works well with other layout algorithms that offer an incremental behavior, that is, for which a small change of the graph usually produces only a small change of the layout. Automatic layout is generally not suitable for non-incremental layout algorithms.

Example of automatic layout
To enable automatic layout:

In CSS
Add to the `GraphLayout` section:

```
autoLayout: "true";
```

In Java
Call:

```
layout.setAutoLayout(true);
```

The following hints are important when programming in Java on an `IlvGrapher` instance:

♦ Automatic layout works well if the `IlvGrapher` instance is not attached to other layouts. If multiple layouts are used for the same `IlvGrapher` instance, they may mutually affect each other. In this case, it is recommended to switch off automatic layout.

♦ The following example shows how to perform multiple changes all at the same time in the `IlvGrapher` instance when automatic layout is switched on. Automatic layout is performed only once at the end of all the changes:

```
layout.attach(grapher);
layout.setAutoLayout(true);
   ...
// switch the notification of changes off
grapher.setContentsAdjusting(true);
try {
    // ... perform multiple changes without any automatic layout
    ...
} finally {
    // now the grapher notifies layout about the changes:
    // therefore, only one automatic layout is performed
    grapher.setContentsAdjusting(false);
}
```

For more information about automatic layout, see the method `performAutoLayout()` in the *Java API Reference Documentation*.

Coordinates mode

The geometry, that is, the position and size, of the graphic objects that are used to represent nodes and links in an `IlvGrapher` instance is subject to a transformer (`IlvTransformer`). By default, the layout algorithms consider the geometry of the nodes and links of an `IlvGrapher` in a coordinate space that is appropriate for most cases. In some situations, it can be useful to specify a different coordinate space. For details, see *Choosing the layout coordinate space*.

Example of specifying coordinate space
To specify, for instance, the view coordinate space:

In CSS
Add to the `GraphLayout` section:

```
coordinatesMode: "VIEW_COORDINATES";
```

In Java
Use the method:

```
void setCoordinatesMode(int mode)
```

The valid values for the coordinates mode are:

♦ `IlvGraphLayout.MANAGER_COORDINATES`

The geometry of the graph is computed using the coordinate space of the manager (that is, the attached `IlvGrapher`) without applying any transformation.

Use this mode:

- if you visualize the graph at zoom level 1, or

- if you do not visualize it at all, or
- if the grapher contains only fully zoomable objects.

  In all these cases, there is no need to take the transformer zoom level into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered specified in manager coordinates.

- ♦ `IlvGraphLayout.VIEW_COORDINATES`

  The geometry of the graph is computed in the coordinate space of the manager view. More exactly, all the coordinates are transformed using the current reference transformer.

  This mode should be used if you want the dimensional parameters of the layout algorithms to be considered as being specified in manager view coordinates.

- ♦ `IlvGraphLayout.INVERSE_VIEW_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager view and then applying the inverse transformation. This mode is equivalent to the “manager coordinates” mode if the geometry of the graphic objects strictly obeys the transformer, that is, the objects are fully zoomable. (A small difference may exist because of the limited precision of the computations.)

  On the contrary, if some graphic objects are either nonzoomable or semizoomable (for example, links with a maximum line width), this mode gives different results from the manager coordinates mode. These results are optimal if the grapher is visualized using the same transformer as the one taken into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered specified in manager coordinates.

In CSS, you omit the prefix `IlvGraphLayout` when specifying the value of the coordinates mode (see *Example of specifying coordinate space*).

The default mode is `INVERSE_VIEW_COORDINATES`.

See also *Specifying the mode for layout coordinates*.

Layout of connected components

The base class `IlvGraphLayout` provides generic support for the layout of a disconnected graph (composed of connected components). For details, see *Laying out connected components of a disconnected graph*.

Example of layout
To enable the placement of disconnected graphs:

In CSS
Add to the GraphLayout section:

```
layoutOfConnectedComponentsEnabled: "true";
```

In Java
Call:

```
setLayoutOfConnectedComponentsEnabled(true);
```

> **Note**: Some of the layout classes (`IlvHierarchicalLayout`, `IlvCircularLayout`) have a built-in algorithm for placing connected components. This algorithm is enabled by default and fits the most common situations. For these layout classes, the generic mechanism provided by the base class `IlvGraphLayout` is disabled by default.

When enabled, a default instance of the class `IlvGridLayout` is used internally to place the disconnected graphs. If necessary, you can customize this layout.

Example of customizing layout

To customize this layout:

In CSS

Add to the GraphLayout section:

```
layoutOfConnectedComponents: "@#GridLayout";
```

and add a new section for the definition of the layout used to place the disconnected graphs, including statements for the parameters you want, for instance:

```
Subobject#GridLayout {
    class: "ilog.views.graphlayout.grid.IlvGridLayout";
    layoutMode: "TILE_TO_ROWS";
    topMargin: "20";
}
```

In Java

Call:

```
IlvGridLayout gridLayout = new IlvGridLayout();
gridLayout.setLayoutMode(IlvGridLayout.TILE_TO_ROWS);
gridLayout.setTopMargin(20);

layout.setLayoutOfConnectedComponents(gridLayout);
```

Example for experts

The various capabilities of the class `IlvGridLayout` cover most of the likely needs for the placement of disconnected graphs. However, if necessary, you can write your own subclass of `IlvGraphLayout` to place disconnected graphs and specify it instead of `IlvGridLayout`:

In CSS

Add to the GraphLayout section:

```
layoutOfConnectedComponents: "@#MyGridLayout";
```

and add a new section to define the layout used to place disconnected graphs, including statements for the parameters you want, for instance:

```
Subobject#MyGridLayout {
    class: "mypackage.MyGridLayout";
    // settings for MyGridLayout, if necessary
}
```

In Java
Call:

```
MyGridLayout myGridLayout = new MyGridLayout();

// settings for myGridLayout, if necessary

layout.setLayoutOfConnectedComponents(myGridLayout);
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsLayoutOfConnectedComponents()
```

The default implementation returns `false`. You can write a subclass to override this behavior.

Layout region

Some layout algorithms can control the size of the graph drawing and can take into account a user-defined layout region.

Example of specifying layout region
To specify that the layout is allowed to run for 60 seconds:

In CSS
If you work with style sheets, you can specify the layout region as a rectangle, for instance:

```
layoutRegion: "0,0,100,100";
```

The above CSS statement sets the layout region to the rectangle with the top-left corner at coordinates 0,0 and width and height at 100.

In Java

```
layout.setLayoutRegion(new IlvRect(0,0,100,100));
```

Besides, the method

```
void setLayoutRegion(IlvRect rect)
```

which defines the layout region in manager coordinates, there are two more ways to set the layout region. These ways are only available in Java, not in CSS:

- `setLayoutRegion(ilog.views.IlvManagerView, ilog.views.IlvRect)`

  The rectangle (the argument `rect`) specifies the layout region. The dimensions of the rectangle are given in view coordinates relative to the input `view` argument. This view is usually the view for displaying the grapher.

- `setLayoutRegion(ilog.views.IlvManagerView)`

  The entire visible area of the input view specifies the layout region.

To access the layout region, use the method:

```
IlvRect getSpecLayoutRegion()
```

This method returns a copy of the rectangle that defines the specified layout region. The dimensions of the rectangle are in the manager (grapher) coordinates. Depending on the last method you called, one of the following cases may occur:

- If `setLayoutRegion(IlvRect)` was the last method called, it returns a copy of the rectangle with no transformations.

- If `setLayoutRegion(IlvManagerView, IlvRect)` was the last method called, it returns a copy of the rectangle transformed to the manager coordinates using the transformer of the view. (The transformation to manager coordinates is not done if the coordinates mode is specified as *view coordinates*.)

- If `setLayoutRegion(IlvManagerView)` was the last method called, it returns a rectangle with the attributes `x=0, y=0` and with the attributes `width` and `height` equal to the current width and height of the view, transformed to manager coordinates using the current transformer of the view. (The transformation to manager coordinates is not done if the coordinates mode is specified as *view coordinates*.)

- None of the methods was called. (This is the default behavior.) If at least one manager view is attached to the grapher, it returns a rectangle with the attributes `x=0, y=0` and with the attributes `width` and `height` equal to the current width and height of the first attached view, transformed to manager coordinates using the transformer of the view. (The transformation to manager coordinates is not done if the coordinates mode is specified as *view coordinates*.) If no view is attached, the method returns `null`.

The layout algorithms call a different method:

```
IlvRect getCalcLayoutRegion()
```

This method first tries to use the layout region specification by calling the method `getSpecLayoutRegion()`. If this method returns a non-null rectangle, this rectangle is returned. Otherwise, the method tries to estimate an appropriate layout region according to the number and size of the nodes in the attached graph. If no graph is attached, or the attached graph is empty, it returns a default rectangle (`0`, `0`, `1000`, `1000`).

To indicate whether a subclass of `IlvGraphLayout` supports the layout region mechanism, use the method:

```
boolean supportsLayoutRegion()
```

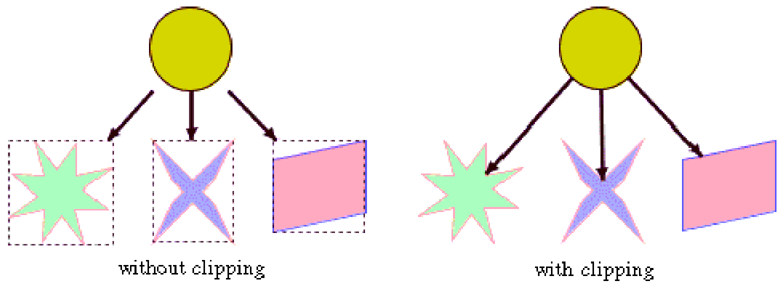
The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Note: The implementation of the method `layout(boolean)` is solely responsible for whether the layout region is taken into account when calculating the layout, and in which manner. For details, refer to the documentation of the layout algorithms.

Link clipping

Some layout algorithms try to calculate the specific connection points of links at the border of nodes and require instances of `IlvFreeLinkConnector` attached to the nodes, while other layout algorithms do not calculate any connection points but simply let the link connectors (any subclass of `IlvLinkConnector`) determine how the links connect to the nodes.

If a layout algorithm calculates specific connection points, then it places the connection points of links by default at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want to place the connection points exactly on the border of the shape. This can be achieved by code by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. Some examples are shown in the following figure.



Effect of link clipping interface

Example of link clipping
To specify the link clip interface:

In CSS
It is not possible to specify the link clip interface in CSS, however the sample in *Writing a new layout renderer to clip links* shows how to integrate a link clip interface into the graph layout renderer.

In Java
Use the method:

```
setLinkClipInterface(ilog.views.graphlayout.IlvLinkClipInterface)
```

You modify the position of the connection points of the links by implementing a class that implements the `IlvLinkClipInterface`. This interface defines the following method:

```
public IlvPoint getConnectionPoint
                          (IlvGraphModel graphModel,
                           Object node,
                           IlvRect currentNodeBox,
                           Object link,
                           IlvPoint proposedConnectionPoint,
                           IlvPoint auxControlPoint,
                           boolean origin)
```

This method `getConnectionPoint(ilog.views.graphlayout.IlvGraphModel, java.lang.Object, ilog.views.IlvRect, java.lang.Object, ilog.views.IlvPoint, ilog.views.IlvPoint, boolean)` allows you to return the corrected connection point when the layout algorithm tries to connect to the proposed connection point. The `auxControlPoint` parameter is the auxiliary control point of the link segment that ends at the proposed connection point. The flag `origin` indicates whether the connection point is the start point or the end point of the link.

One strategy is to calculate the intersection between the ray starting at `auxControlPoint` and going through `proposedConnectionPoint` and the shape of the node. If there is any intersection, we return the one closer to `auxControlPoint`. If there is no intersection, clipping is not possible and we return the proposed connection point.

The following sample shows how to set a link clip interface that clips the connection points at the border of an ellipse or circle node:

```
layout.setLinkClipInterface(new IlvLinkClipInterface() {
    public IlvPoint getConnectionPoint
                                (IlvGraphModel graphModel,
                                 Object node,
                                 IlvRect nodeBox,
                                 Object link,
                                         IlvPoint proposedConnectionPoint,
                                 IlvPoint auxControlPoint,
                                 boolean origin)
    {
      // get the intersections between the line through connect and control
      // point and the ellipse at currentNodeBox.
      IlvPoint[] intersectionPoints = new IlvPoint[2];
      int numIntersections = IlvGraphLayoutUtil.LineIntersectsEllipse(
                                   proposedConnectionPoint, auxControlPoint,
                                   nodeBox, intersectionPoints);
      // choose the result from the intersections
      return IlvGraphLayoutUtil.BestClipPointOnRay(proposedConnectionPoint,
                                                   auxControlPoint,
                                                   intersectionPoints,
                                                   numIntersections);
     }

});
```

The sample in *Writing a new layout renderer to clip links* shows how to integrate a link clip interface into the graph layout renderer.

Note: In addition to the link-clip interface, you can use the class `IlvClippingLinkConnector`. This special link connector clips the links at nonrectangular node shapes and updates the connection points automatically during interactive node movements.

To indicate whether a subclass of `IlvGraphLayout` supports the link clip interface, use the method:

```
boolean supportsLinkClipping()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Link connection box

If a layout algorithm calculates specific connection points, it places the connection points of links by default at the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes (see *Effect of Link Connection Box Interface*). This can be achieved by specifying a link connection box interface. The link connection box interface allows you to specify, for each node, a node box different from the bounding box that is used to connect the links to the node.

Example of link connection box interface
In CSS
It is not possible to specify the link connection box interface in CSS. The diagram component uses some predefined link-connection-box interfaces in combination with nodes of type `IlvGeneralNode`. If you need to use a different link-connection-box interface, you must integrate it in the graph layout renderer in the same way as the link clipping interface (see *Writing a new layout renderer to clip links* for a sample that integrates the link clipping interface).

In Java
To set a link connection box interface in Java, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

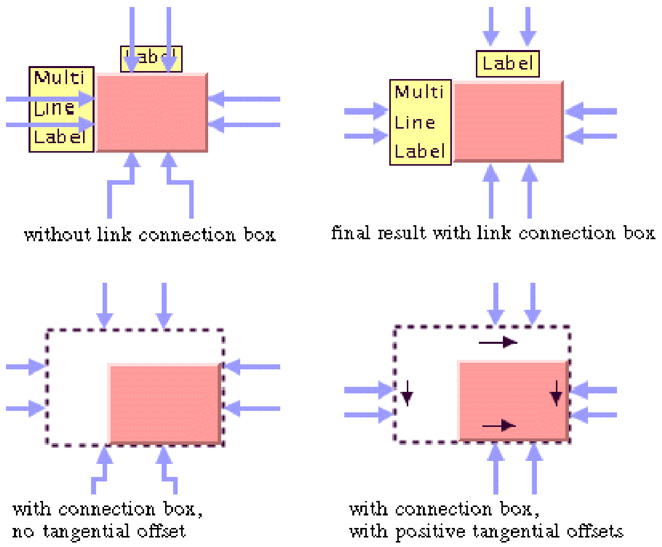
You implement the link connection box interface by defining a class that implements the `IlvLinkConnectionBoxInterface`. This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

A second method defined on the interface allows the connection points to be “shifted” tangentially, in a different way for each side of each node:

```
public float getTangentialOffset(IlvGraphModel graphModel,
                                 Object node, int nodeSide);
```



Effect of Link Connection Box Interface

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes of type `IlvShadowRectangle` and shifts up the connection points on the left and right side of all the nodes, call:

```
layout.setLinkConnectionBoxInterface(new IlvLinkConnectionBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof IlvShadowRectangle) {
            // need a rect that is 4 pixels smaller
            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
    public float getTangentialOffset(IlvGraphModel graphModel,
                                     Object node, int nodeSide) {
        switch (nodeSide) {
          IlvDirection.Left:
          IlvDirection.Right:
            return -10; // shift up with 10 for both left and right side
          IlvDirection.Top:
          IlvDirection.Bottom:
            return 0; // no shift for top and bottom side
    }
});
```

Some layout algorithms allow you to use the link connection box interface and the link clip interface in a combined way. It is specific to each layout algorithm how the interfaces will be used and which connection points are the final result.

To indicate whether a subclass of `IlvGraphLayout` supports the link connection box interface, use the method:

```
boolean supportsLinkConnectionBox()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Spline routing

Some layout algorithms always use straight links, while other layout algorithms can calculate bend points for polyline links. If splines are used instead of polyline links, special control points must be calculated for spline links. There is a generic spline control point optimization available as a postprocessing step.

If a layout algorithm supports multiple link shapes, the spline optimization affects only those links with bends. It does not affect straight links or links that are marked as fixed or non-reshapeable. Furthermore, it affects only those links that are really spline links. If you use customized `IlvGraphic` data structures instead of `IlvSplineLinkImage` or `IlvGeneralLink`, you must set an `IlvSplineLinkFilter` which tells the layout which link classes are splines. By default, only `IlvSplineLinkImage` or `IlvGeneralLink` are recognized as splines.

Example of spline routing

In CSS

If you work with style sheets, add to the `GraphLayout` section, for example:

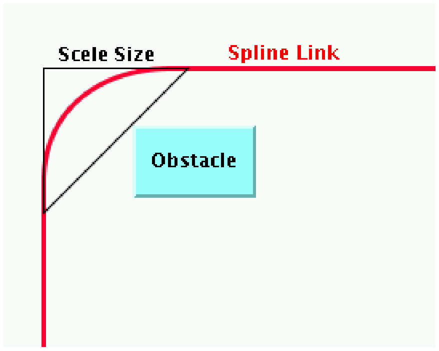
```
splineRoutingEnabled: "true";
minSplineCurveSize: "5";
maxSplineCurveSize: "100";
balanceSplineCurveThreshold: "3";
```

See below the meaning of these parameters.

In Java

To enable the spline routing, call:

```
layout.setSplineRoutingEnabled(true);
```



Spline Routing

When the layout algorithm needs to create a bend, the spline routing tries to determine a triangle at the bend so that the curve of the spline runs inside this triangle. The size of the triangle depends on the available free space and the location of the other nodes, which are considered obstacles for the spline. The scele size of the triangle is controlled by two parameters:

- `layout.setMinSplineCurveSize(min)`
- `layout.setMaxSplineCurveSize(max)`

The algorithm tries to find a triangle with a scele size between `min` and `max`. If a lot of free space is available, it chooses a triangle at `max` size. If no free space is available, it chooses a triangle at `min` size, even if this will cause an overlap of the spline with neighbor nodes. Therefore it is recommended to set the minimal spline curve size to a very small value.

The algorithm chooses isoscele triangles whenever possible, because the shape of a spline link looks more balanced if the curves run inside isoscele triangles. However, if there is no available space, then isoscele triangles are impossible and triangles with different scele lengths are choosen. A threshold determines how small a triangle can be before non-isoscele triangles are choosen:

`layout.setBalanceSplineCurveThreshold(threshold)`

A spline link filter is a subclass of `IlvSplineLinkFilter` that determines which links are splines. The base class `IlvSplineLinkFilter` simply tests the method `IlvGraphic.isSpline`. Currently, `IlvSplineLinkImage`, `IlvGeneralLink` and `IlvCompositeLink` return true when certain link parameters are set so that they behave like splines. You can set your own spline link filter that is adapted to your `IlvGraphic` data structures if needed. Call:

```
layout.setSplineLinkFilter(filter);
```

Memory savings

The computation of a layout on a large graph may require a large amount of memory. Some layout algorithms optionally use two ways to store data: one which gives the priority to speed (this is the default case), the other which consumes less memory and is usually slower. The amount of memory savings depends, of course, on the implementation of the subclass of

`IlvGraphLayout`. No matter which option you choose for memory savings, the resulting layout should be the same.

Example of memory savings
To enable memory savings:

In CSS
Add to the `GraphLayout` section:

```
memorySavings: "true";
```

In Java
Use the method:

```
void setMemorySavings(boolean option)
```

Memory savings is disabled by default.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsMemorySavings()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Percentage of completion calculation

Some layout algorithms can provide an estimation of how much of the layout has been completed. This estimation is made available as a percentage value that is stored in the graph layout report. When the algorithm starts, the percentage value is set to 0. The layout algorithm calls the following method from time to time to increase the percentage value by steps until it reaches 100:

```
void increasePercentageComplete(int newPercentage);
```

The percentage value can be accessed from the layout report using the following:

```
int percentage = layoutReport.getPercentageComplete();
```

To see an example of how to read the percentage value during the running of a layout, see *Graph layout event listeners*.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsPercentageComplete()
```

The default implementation returns `false`. A subclass can override this method to return `true` to indicate that this mechanism is supported.

Preserve fixed links

At times, you may want some links of the graph to be “pinned” (that is, to stay in their current shape when the layout is performed). You need a way to indicate the links that the layout algorithm cannot reshape. This makes sense especially when using a semi-automatic layout (the method where the end user fine-tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the shape of the links is modified after the layout has been performed, and then the layout is performed again).

Example of fixing links

To specify that a link is fixed:

In CSS

1. Create a rule that selects the link, for instance:

   ```
   #link1 {
     Fixed: "true";
   }
   ```

2. Add this CSS statement to the `GraphLayout` section:

   ```
   preserveFixedLinks: "true";
   ```

In Java

Use the method:

```
void setFixed(Object link, boolean fixed)
```

If the `fixed` parameter is set to `true`, it means that the link is fixed. To obtain the current setting for a link:

```
boolean isFixed(Object link)
```

The default value is `false`.

To remove the fixed attribute from all links in the grapher, use the method:

```
void unfixAllLinks()
```

The fixed attributes on links will be taken into consideration only if you additionally call the following statement:

```
layout.setPreserveFixedLinks(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsPreserveFixedLinks()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Preserve fixed nodes

At times, you may want some nodes of the graph to be “pinned” (that is, to stay in their current position when the layout is performed). You need a way to indicate the nodes that the layout algorithm cannot move. This makes sense especially when using a semi-automatic layout (the method where the end user fine-tunes the layout by hand after the layout is completed) or when using an incremental layout (the method where the graph and/or the position of the nodes is modified after the layout has been performed, and then the layout is performed again).

Example of fixing nodes

To specify that a node is fixed:

In CSS

1. Create a rule that selects the node, for instance:

   ```
   #node1 {
     Fixed: "true";
   }
   ```

2. Add this CSS statement to the `GraphLayout` section:

   ```
   preserveFixedNodes: "true";
   ```

In Java

Use the method:

```
void setFixed(Object node, boolean fixed)
```

If the `fixed` parameter is set to `true`, it means that the node is fixed. To obtain the current setting for a node:

```
boolean isFixed(Object node)
```

The default value is `false`.

To remove the fixed attribute from all nodes in the grapher, use the method:

```
void unfixAllNodes()
```

The fixed attributes on nodes will be taken into consideration only if you also call:

```
layout.setPreserveFixedNodes(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsPreserveFixedNodes()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this mechanism is supported.

Random generator seed value

Some layout algorithms use random numbers (or randomly chosen parameters) for which they accept a user-defined seed value. For example, the Random Layout uses the random generator to compute the coordinates of the nodes. The Uniform Length Edges Layout uses the random generator to compute some internal variables.

Subclasses of `IlvGraphLayout` that are designed to support this mechanism allow the user to choose one of three ways of initializing the random generator:

♦ With a default value that is always the same.

♦ With a user-defined seed value that can be changed when re-performing the layout.

♦ With an arbitrary seed value, which is different each time. In this case, the random generator is initialized based on the system time.

The user chooses the initialization option depending on what happens when the layout algorithm is performed again on the same graph. If the same seed value is used, the same layout is produced, which may be the desired result. In other situations, the user may want to produce different layouts in order to select the best one. This can be achieved by performing the layout several times using different seed values.

Example of random generator seed value
To specify that the layout is allowed to run for 60 seconds:

In CSS
You can specify for instance the seed value 25 of the random generator by adding the following statements to the `GraphLayout` section:

```
seedValueForRandomeGenerator: "15";
useSeedValueForRandomGenerator: "true";
```

The first statement defines the seed value, and the second statement specifies that the seed value must be used.

In Java
This example shows how this parameter can be used in Java in combination with the `java.util.Random` class in your implementation of the method `IlvGraphLayout.layout()`:

```
Random random = (isUseSeedValueForRandomGenerator()) ?
```

```
  new Random(getSeedValueForRandomGenerator()) :
  new Random();
```

To specify the seed value in Java, use the method:

```
void setSeedValueForRandomGenerator(long seed)
```

The default seed value is `0`.

The user-defined seed value is used only if you call additionally

```
layout.setUseSeedValueForRandomGenerator(true);
```

To indicate whether a subclass of `IlvGraphLayout` supports this parameter, use the method:

```
boolean supportsRandomGenerator()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this parameter is supported.

Save parameters to named properties

There are many ways to store your graph and your parameters:

♦ The diagram component uses XML files for the data and CSS files for the rendering parameters.

♦ The diagram component can also use a database.

♦ The IBM® ILOG® JViews grapher can be stored in `.ivl` files.

The base class `IlvGraphLayout` provides support for saving the layout parameters (such as `isAnimate` or `isMemorySavings`) to `.ivl` files or to transfer the parameters to named properties. This is an advanced mechanism that is explained in detail in *Saving layout parameters and preferred layouts*. If you use XML files, CSS files, or databases, there is no point using this advanced mechanism.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsSaveParametersToNamedProperties()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Stop immediately

Several layout algorithms can stop computation when an external event occurs, for instance when the user hits a “Stop” button. In Java, to stop the layout, you can call:

```
boolean stopImmediately();
```

This method is typically called in a multithreaded application from a separate thread that is not the layout thread. The method returns `true` if the stop was initiated and `false` if the algorithm cannot stop. The method returns immediately, but the layout thread usually needs some additional time after initiating the stop to clean up data structures.

The following code fragment illustrates the usage.

You start the layout in a separate thread:

```
Thread layoutThread = new Thread(new GraphLayoutPerformer(layout, grapher));
layoutThread.start();
```

The class `GraphLayoutPerformer` is an implementation of the interface `Runnable` that performs layout. The following is a sketch of this class:

```
class GraphLayoutPerformer implements Runnable
{
  ...
  public void run()
  {
    // from now we are busy
    busy = true;
    try {
      // perform the layout
      layout.performLayout(true, true);
    }
    catch (IlvGraphLayoutException e) {
      ... // handle the excepction
    }
    finally {
      // we are not busy anymore
      busy = false;
    }
  }
}
```

The Stop button operates outside the layout thread and simply calls the method `stopImmediately` of the running layout instance:

```
Button stopButton = new Button("Stop Layout");
stopButton.addActionListener(new ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (busy) layout.stopImmediately();
    }
});
```

Note: A detail has been omitted from the previous code fragment. A multitasking operation requires that the layout thread calls the `yield()` or `sleep(t)` methods from time to time. A good place to do this is by using a graph layout event listener. Event listeners are explained in *Using event listeners*.

The consequences of stopping a layout process depend on the specific layout algorithm. Some layout algorithms have an iterative nature. Stopping the iteration process results in a slight loss of quality in the drawing, but the layout can still be considered valid. Other layout algorithms have a sequential nature. Interrupting the sequence of the layout steps may not result in a valid layout. Usually, these algorithms return to the situation before the start of the layout process.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, use the method:

```
boolean supportsStopImmediately()
```

The default implementation returns `false`. You can write a subclass to override this method in order to return `true` to indicate that this mechanism is supported.

Use default parameters

All the generic parameters have a default value. After modifying parameters, you may want the layout algorithm to use the default values. Then, you may want to return to your customized values. IBM® ILOG® JViews keeps the previous settings when selecting the default values mode. In Java, you can switch between the default values mode and the mode for your own settings using the method:

```
void setUseDefaultParameters(boolean option)
```

To obtain the current value:

```
boolean isUseDefaultParameters()
```

The default value is `false`. This means that any setting you make will be taken into consideration and the parameters that have not been specified will have their default values.

Layout characteristics

It is often useful to know how certain settings will affect the resulting layout of the graph after the layout algorithm has been applied. The following table provides additional information about the behavior of the layout algorithms.

Layout characteristics of layout algorithms

Layout algorithm	Do the initial positions of the nodes affect the layout?	How do I get a different layout of the same graph when I perform the layout a second time?
Topological Mesh Layout (TML)	No	You can completely change the layout by using the starting node, outer cycle, and fixed nodes parameters. To change only the dimensions of the graph, use the layout region parameter. See *Outer cycle (TML)*, *Using fixed nodes (TML)*, and *Layout region (TML)*.
Force-directed or Uniform Length Edges Layout (ULEL)	Yes	In incremental mode, you can completely change the layout by changing the initial positions of the nodes. To change only the dimensions of the graph, use the preferred length of the links or size of the layout region. See *Preferred length (ULEL)*.
Tree Layout (TL)	Yes (if incremental mode is switched on)	In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can change the layout by selecting a different *Root node (TL)*. To change only the dimensions of the graph, use the various offset parameters.
Hierarchical Layout (HL)	Yes (if incremental mode is switched on)	In incremental mode, you can change the layout by changing the initial positions of the nodes. Furthermore, you can use specified node level indices to change the level structure. See *Level index parameter (HL)*. You can use specified node position indices to change the node order within the levels. See *Position index parameter (HL)*. You can change the layout by changing the link priorities. See *Link priority parameter (HL)*. To change only the dimensions of the graph, use the various offset parameters.
Link layout (LL)	Yes	Link Layout routes the links depending on the node positions. It does not move the nodes. You can change the link style option and the dimensional parameters, such as

Layout algorithm	Do the initial positions of the nodes affect the layout?	How do I get a different layout of the same graph when I perform the layout a second time?
		the link offset and final segment length. You can also specify the rules for computing the connection points of the links.
Random layout (RL)	No	This is the default behavior when using the default parameter settings (the random generator is initialized differently each time).
Bus layout (BL)	No, except in incremental mode	You change the dimensions of the graph by using the various dimensional parameters.
Circular layout (CL)	No	You can completely change the layout by using clustering settings and the root clusters parameter. You can change the dimensions of the graph by using the dimensional parameters.
Grid layout (GL)	Yes (if incremental mode is switched on)	You can change various dimensional parameters, layout mode, and so on.
Recursive layout	Depends on the behavior of the sublayouts applied to the subgraphs.	Depends on the behavior of the sublayouts applied to the subgraphs. You can change the parameters of the sublayouts individually.
Multiple layout	Depends on the behavior of the sublayout that is applied first.	Depends on the behavior of the sublayouts of the Multiple Layout instance. You can change the parameters of the sublayouts individually.

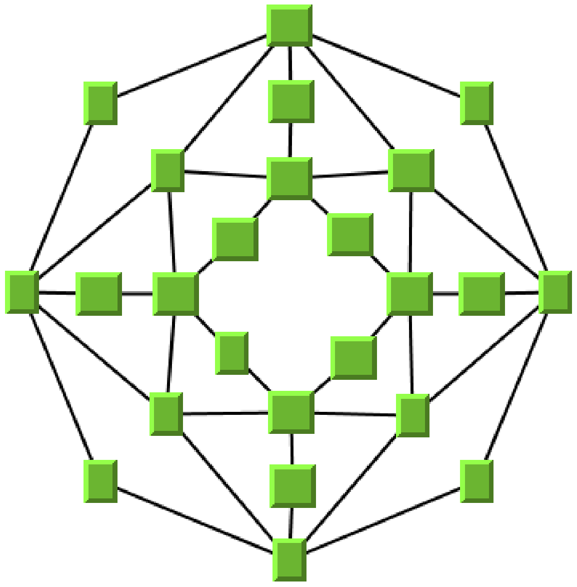
Topological Mesh Layout (TML)

Gives information on the *Topological Mesh Layout (TML)* algorithm (class `IlvTopologicalMeshLayout` from the package `ilog.views.graphlayout.topologicalmesh`).

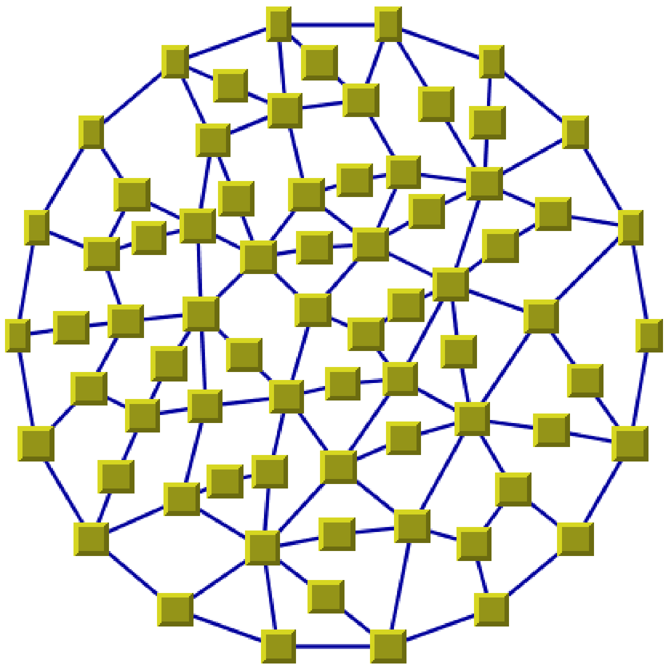
In this section

TML samples

The following sample drawings were produced with TML.



Small cyclic graph drawing produced with TML



Large cyclic graph drawing produced with TML

What types of graphs suit the TML?

- Cyclic (2-connected graph) graphs. (Preferably without cut-nodes or cut-edges; otherwise, manual adjustments are necessary.)
- Cyclic (2-connected) graphs plus only a few branches. (You may need to make manual adjustments for the branches.)
- Both planar graphs and nonplanar graphs.

Application domains for the TML

Application domains of the Topological Mesh Layout include:

- Database and knowledge engineering (semantic networks, qualitative reasoning and other artificial intelligence diagrams)

Features and limitations of the TML

Features

- Most of the time, produces planar drawings of planar graphs, and drawings with a small number of link crossings for nonplanar graphs.
- Produces a nice layout for most small- and medium-size graphs relatively quickly. (The maximum cyclomatic number of the graph is about 30-50, but the number of nodes and links can be a lot higher.)
- Most of the time, produces symmetrical drawings of symmetrical graphs.
- The computation time for one iteration depends on the cyclomatic number of the graph, which is smaller than the number of nodes or links.
- The user can obtain several layouts of the same graph easily and quickly by simply changing a parameter (especially the starting node and the outer cycle) or by applying manual refinements to the layout. The best layout can then be selected from the resulting layouts.

Limitations

- The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to quickly solve this problem for any graph size. Therefore, the algorithm uses heuristics that cannot always obtain a layout with the theoretical minimum number of link crossings.
- The computation time required to obtain an appropriate drawing grows relatively quickly with the cyclomatic number and the layout process may become very time-consuming for large graphs. Again, this is because the minimization of the number of link crossings is mathematically NP-complete in the general case.
- The algorithm cannot automatically produce appropriate drawings of some types of graphs:
  - For graphs containing branches and graphs containing cut-nodes or cut-edges, manual adjustments are necessary. (See *Refining a graph layout (TML)*.)
  - For disconnected graphs, the connected component layout feature should be used. (See *Layout of connected components*)
- The layout algorithm often produces a drawing with no overlapping nodes. Nevertheless, overlapping nodes cannot always be avoided. When overlapping occurs, you can try to increase the size of the layout region parameter or to change the outer cycle (see the method `setExteriorCycleId(int)`). You can also use manual adjustments to correct the problem.

The TML algorithm

TML is a heuristical approach for the layout of cyclic graph, either planar graphs or nonplanar graphs. TML is very simple to use. However, to use all the functionality of TML, you should understand its basic concepts.

When laying out a general graph, producing a drawing with a minimum number of link crossings is a mathematically NP-complete problem. The search space (and time) grows exponentially with the graph size. Traditionally, most of the existing layout algorithms use node coordinates from the beginning, searching for a coordinate set to minimize the cost function, which is mainly the number of link crossings. These coordinates can be constrained on a grid, but the number of combinations to explore is still enormous.

In contrast, TML uses a **two-step approach** that drastically reduces the number of combinations to explore. The first step of TML deals only with the pure topology (that is, the connectivity) of the graph without taking into consideration the node coordinates. This first step is called **topological optimization.** It chooses one of the cycles of the graph to be used in the second step.

In the second step, called **node placement**, the result of the first step is used to compute the coordinates of the nodes using a deterministic, high-speed barycenter algorithm. Of course, the problem still remains NP-complete and large graphs cannot be processed. In practice, however, you will often get better results for “mesh” graphs with TML than with many other algorithms.

Step 1: Topological optimization

Input

The topology of the graph (its connectivity or the neighborhood relationships between nodes).

Output

A set of possible outer cycles, ordered decreasingly by their lengths. The length of a cycle is the number of nodes in the cycle.

Explanation

This step determines what cycles of the graph, if used as an outer cycle for drawing the graph during nodes placement, will allow a drawing with a minimum number of link crossings. An optimization algorithm tries to minimize a heuristic cost function that estimates the number of link crossings for each solution, based on pure topology (graph connectivity)

Step 2: Node placement

Input

The output of topological optimization and the graph.

Output

A set of coordinates for the nodes. The coordinates are assigned to the nodes to obtain the graph drawing.

Explanation

This step is a variant of the “barycentric” layout algorithm. It takes a cycle from the output of topological optimization and draws it as a regular polygon. Then, it iteratively moves each node (except those on the regular polygon) at the “barycenter” of its neighbors (the nodes to which it is connected). This procedure always converges, and the final result is a graph drawing where the number of link crossings is dependent only on the choice of the outer cycle.

Example of TML

In CSS

Below is an example of specification in CSS using the Topological Mesh Layout algorithm. Since the Topological Mesh Layout places nodes and reshapes the links, it is usually not necessary to specify an additional link layout in CSS. The specification in CSS can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see Graph Layout in IBM® ILOG® JViews Diagrammer).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

 GraphLayout {
    enabled                               : "true";
    graphLayout                           : "TopologicalMesh";
    allowedOptimizationTime               : "20000";
    allowedNumberOfOptimizationIterations : "5";
    allowedNodesPlacementTime             : "20000";
    nodesPlacementAlgorithm               : "SLOW_GOOD";
     linkStyle                                        : "STRAIGHT_LINE_STYLE";

         }
```

In Java™

Below is a code sample using the `IlvTopologicalMeshLayout` class. This code sample shows how to perform a Topological Mesh Layout on a grapher directly without using a diagram component or any style sheet:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.topologicalmesh.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*   Fill in the grapher with nodes and links here */

IlvTopologicalMeshLayout layout = new IlvTopologicalMeshLayout();
layout.attach(grapher);
try {
        IlvTopologicalMeshLayoutReport layoutReport =
              (IlvTopologicalMeshLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

It is possible to enable the link layout additionally, and in this case, the link layout determines the shape of the links.

Important: All explanations in the subsequent sections regarding the shape of the links in Topological Mesh Layout are valid only if the link layout is disabled.

Generic features and parameters of the TML

Overview (TML)

TML supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

♦ *Allowed time (TML)*

♦ *Animation (TML)*

♦ *Layout of connected components (TML)*

♦ *Layout region (TML)*

♦ *Link clipping (TML)*

♦ *Link connection box (TML)*

♦ *Memory savings (TML)*

♦ *Preserve fixed links (TML)*

♦ *Preserve fixed nodes (TML)*

♦ *Save parameters to named properties (TML)*

♦ *Stop immediately (TML)*

Note that all the methods allowing the modification of these parameters are overridden in this subclass. This class keeps track of the changes for parameters that may affect the result of Topological Optimization separately from the parameters that may affect only the nodes placement step. In this way, the Topological Optimization step is not repeated. The previous results are used if no parameters were modified since the last time the layout was successfully performed on the same graph using the same layout instance.

Allowed time (TML)

The Topological Optimization step of TML stops if the allowed time setting has elapsed. In the same manner, the Nodes Placement step of TML stops if the allowed time is exceeded. (See *Allowed time*.)

You can specify separate time settings for each step. Each step is stopped if its specified time limit is exceeded. To learn how to do this, see *Optimization iterations and allowed time (TML)* and *Node placement iterations and allowed time (TML)*.

Animation (TML)

In TML, the animation parameter (see *Animation*) is used only in the Nodes Placement step. If specified, a redraw will be automatically performed after each iteration of the step. This is useful to better understand what is happening during the step and especially if you want to be able to choose the fixed nodes in a manual refinement procedure. (See *Using fixed nodes (TML)*.)

Layout of connected components (TML)

The layout algorithm can use the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of connected components*.)

If the generic connected component layout mechanism is disabled, the algorithm lays out only the connected component that contains the starting node.

Layout region (TML)

The Nodes Placement step of TML first draws the outer cycle computed in the Topological Optimization step as a regular polygon. It uses the layout region setting (either your own or the default setting) to choose the size and the position of the polygon. The remaining nodes are moved inside this polygon. All three ways to specify the layout region are available for this subclass. (See *Layout region*.)

Remember that if you are using the default settings, the visible area of the manager view (an instance of `IlvManagerView`) attached to the grapher is used as a layout region. If several manager views are attached, the first attached view is used. If no manager view is attached, the layout region is automatically estimated on the basis of the number and size of the nodes.

If TML produces a layout with overlapping nodes, one possible way to correct the problem is to increase the size of the layout region. (For details, see *Using the layout region parameter (TML)*.)

Link clipping (TML)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the TML* for details of the link clipping mechanism in TML.

Link connection box (TML)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link clipping interface with the TML*.

Memory savings (TML)

As with all classes supporting this parameter, a certain amount of memory savings can be obtained by selecting this option. Note that using this option does not change the resulting layout. It just slows down the computation. (See *Memory savings*.)

Preserve fixed links (TML)

TML does not reshape the links that are specified as fixed. (See *Preserve fixed links*. See also *Link style (TML)*.)

Preserve fixed nodes (TML)

TML does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve fixed nodes*.)

If TML produces a layout with overlapping nodes, you can use the fixed nodes mechanism to correct the problem. (For details, see *Using fixed nodes (TML)*.)

Save parameters to named properties (TML)

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ilv` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*).

Stop immediately (TML)

The layout algorithm stops after cleanup if the method `IlvTopologicalMeshLayout` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

See also

Support by algorithms of generic features and parameters

Specific parameters of the TML

The following parameters are specific to the `IlvTopologicalMeshLayout` class.

Link style (TML)

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To specify that the layout algorithm automatically remove all the intermediate points of the links (if any):

In CSS
Add to the `GraphLayout` section:

```
linkStyle: "STRAIGHT_LINE_STYLE";
```

In Java™
Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- ♦ `IlvTopologicalMeshLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- ♦ `IlvTopologicalMeshLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (except for links specified as fixed) are removed. This is the default value.

Note: The layout algorithm may raise an `IlvInappropriateLinkException` if layout is performed on an `IlvGrapher`, but inappropriate link classes or link connector classes are used. See *Layout exceptions* for details and solutions to this problem.

Optimization iterations and allowed time (TML)

The iterative computation performed in the Topological Optimization step is stopped if the number of iterations exceeds the allowed number of iterations for optimization or the time exceeds the allowed time for optimization (or, of course, if the general layout time has elapsed; see *Allowed time (TML)*).

To specify the parameters:

In CSS
Add to the `GraphLayout` section:

```
allowedOptimizationTime: "30000";
```

```
allowedNumberOfOptimizationIterations: "5";
```

In Java
Use the methods:

```
void setAllowedOptimizationTime(long time)
void setAllowedNumberOfOptimizationIterations(int iter)
```

The `time` is in milliseconds. The default value is 28000 (28 seconds).

Node placement iterations and allowed time (TML)

The iterative computation performed in the Nodes Placement step is stopped if the number of iterations exceeds the allowed number of iterations or the time exceeds the allowed time for node placement (or, of course, if the general layout time has elapsed; see *Allowed time (TML)*).

To specify these parameters:

In CSS
Add to the `GraphLayout` section:

```
allowedNodesPlacementTime: "30000";
allowedNumberOfNodesPlacementIterations: "5";
```

In Java
Use the methods:

```
void setAllowedNodesPlacementTime(long time)
```

```
void setAllowedNumberOfNodesPlacementIterations(int iter)
```

The `time` is in milliseconds. The default value is 28000 (28 seconds).

Node placement algorithm (TML)

Two barycentric algorithms are implemented for the Nodes Placement step of TML.

To specify the algorithm:

In CSS
Add to the `GraphLayout` section:

```
nodesPlacementAlgorithm: "QUICK_BAD";
```

In Java
Use the method:

```
void setNodesPlacementAlgorithm(int option)
```

The valid values for `option` are:

- `IlvTopologicalMeshLayout.SLOW_GOOD`

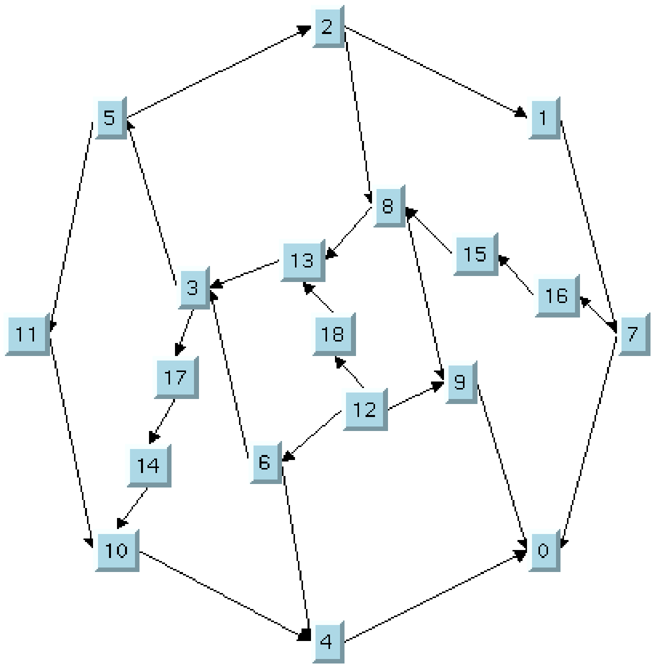
  This option provides more uniformity of the nodes distribution inside the outer cycle, but is slightly slower.

- `IlvTopologicalMeshLayout.QUICK_BAD`

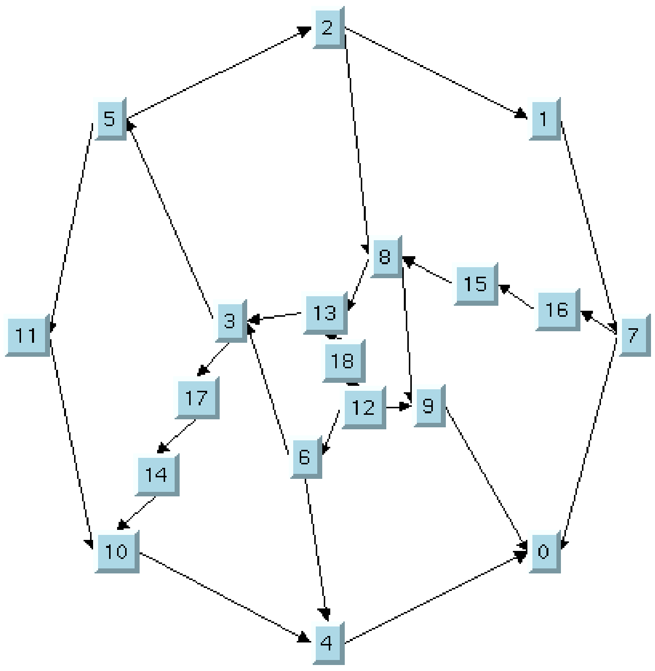
  This option provides less uniformity of the nodes distribution, but is slightly quicker.

In most cases, both algorithms are fairly quick. We recommend that you use the `SLOW_GOOD` version, which is the default value. Compare the layouts of the same graph in *Node placement algorithm:* *`SLOW_GOOD`* and *Node placement algorithm:* *`QUICK_BAD`* to get an idea of the difference between these algorithms.

In CSS, you omit the prefix `IlvTopologicalMeshLayout` when specifying the value of the nodes placement.



Node placement algorithm: *`SLOW_GOOD`*



Node placement algorithm: `QUICK_BAD`

Outer cycle (TML)

The Topological Optimization step of TML computes a set of cycles that can be used as the outer cycle in the Nodes Placement step . By default, the longest cycle is actually used (that is, the cycle containing the largest number of nodes). However, you may find it useful to try a different outer cycle. To do so in Java, use the method:

```
void setExteriorCycleId(int cycleId)
```

The valid values for `cycleId` range from zero to the number of cycles computed by the Topological Optimization step minus one. This number is returned by the method:

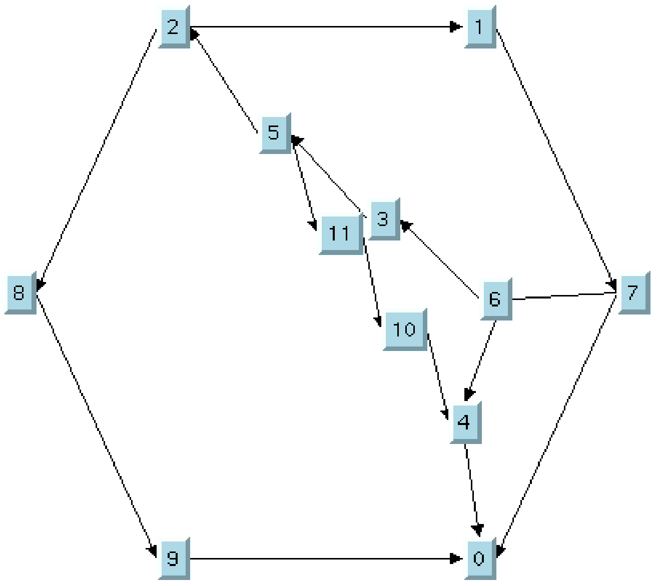
```
int getNumberOfPossibleExteriorCycles()
```

If the number is not in this range, the value zero is used.

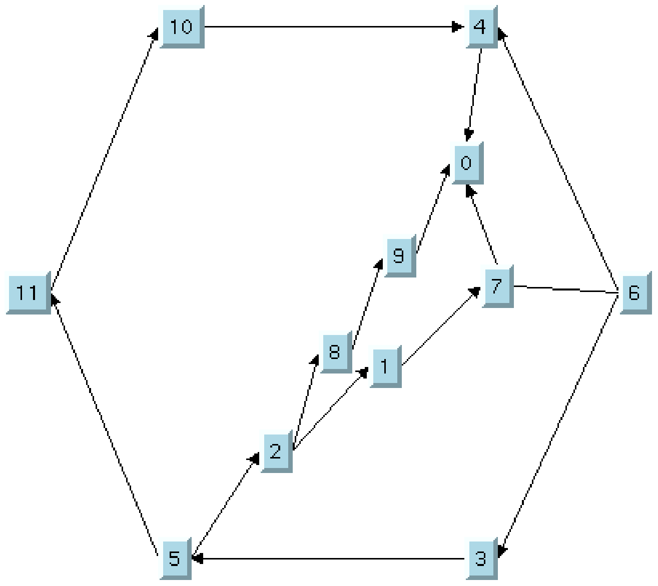
You can use these methods only after having performed the layout successfully. Otherwise, no outer cycle is defined.

When the layout is performed again with a new outer cycle, only the Nodes Placement step of TML is performed, and not the time-consuming the Topological Optimization step. This is true if the topology of the graph has not been changed (that is, no nodes or links were added or removed), and no parameters that affect the Topological Optimization step have been changed.

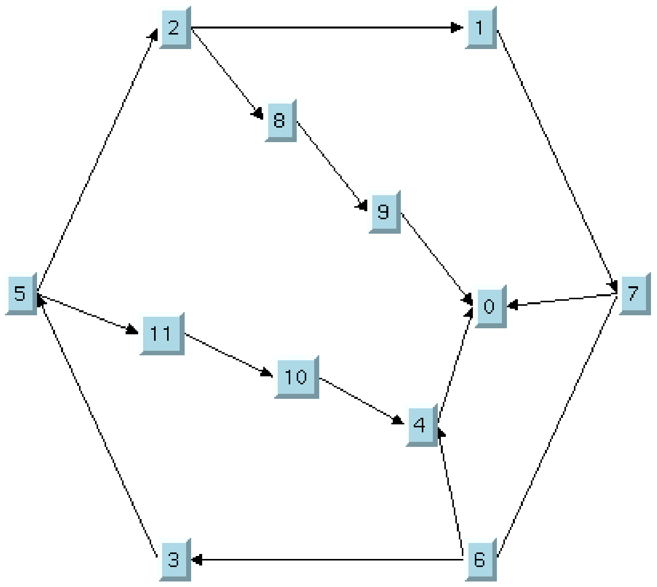
Layout using 1st outer cycle, *Layout using 2nd outer cycle*, *Layout using 3rd outer cycle*, and *Layout using 4th outer cycle* show various layouts produced **for the same graph** when the `cycleId` parameter is changed:



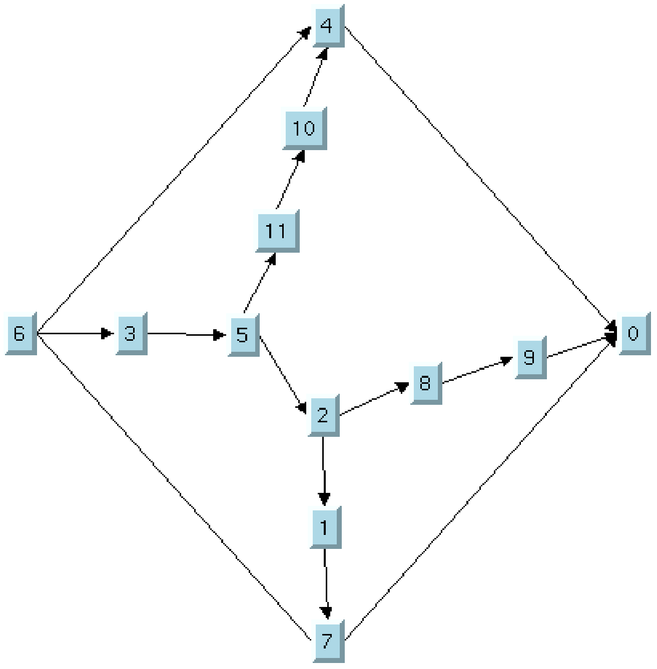
Layout using 1st outer cycle



Layout using 2nd outer cycle



Layout using 3rd outer cycle



Layout using 4th outer cycle

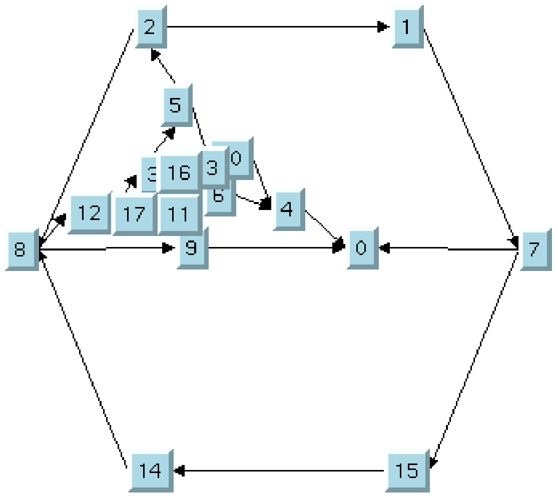
Refining a graph layout (TML)

After performing the layout on a graph, you may want to improve the quality of the layout by making some manual refinements. The subsequent sections describe several ways to refine your layouts. When the layout is performed again after the refinements have been applied, only the Nodes placement step of TML is redone. The results of the Topological Optimization are reused. This is an important benefit of TML because the algorithm can recompute a layout using new parameters very quickly, without performing the time-consuming Topological Optimization step again.

Using fixed nodes (TML)

One reason for applying manual refinements is to avoid overlapping nodes. To do this, you can use the fixed nodes mechanism. (See *Preserve fixed nodes*.)

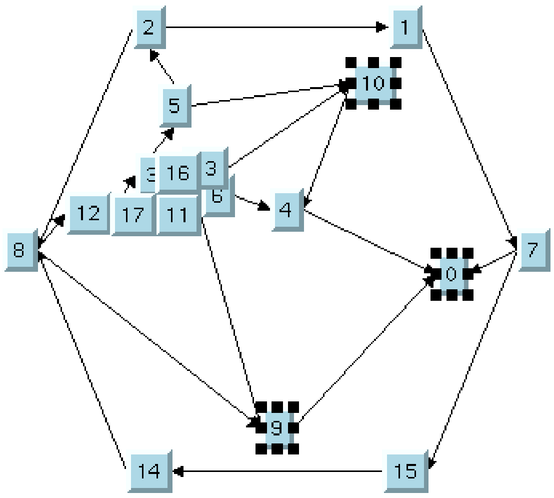
Take a look at the original layout shown in *The original TML layout* . Several overlapping nodes exist in the original layout because the nodes are concentrated in a small region and do not use the available space inside the outer cycle.



The original TML layout

To correct the problem, you can perform the following steps:

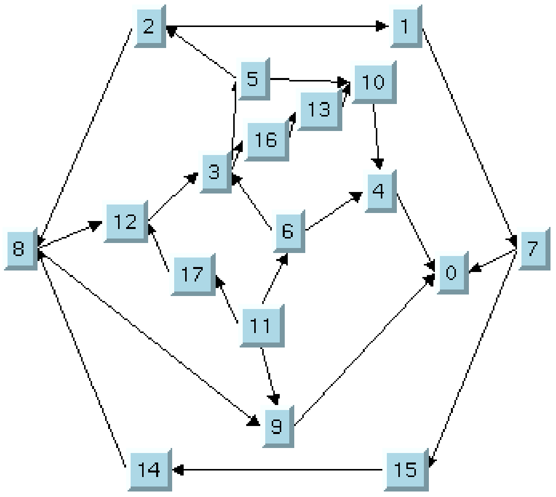
1. Move nodes 0, 9, and 10 to a place in the free space inside the outer cycle by hand as shown in *The TML layout with some nodes moved* .



The TML layout with some nodes moved

2. Specify nodes 0, 9, and 10 as fixed using the `setFixed(java.lang.Object, boolean)` method.
3. Use the `setPreserveFixedNodes(boolean)` method to specify that the fixed nodes will not be moved when the layout is performed.
4. Perform the layout again. Only Step 2 will be performed.

   The fixed nodes "attract" the other nodes, which are distributed in the larger area inside the outer cycle as shown in *The final TML layout with some fixed nodes*.



The final TML layout with some fixed nodes

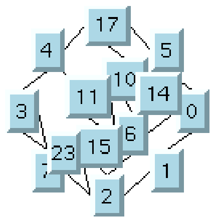
Using the outer cycle parameter (TML)

By default, the Nodes Placement step of TML produces a layout using the longest outer cycle computed in the Topological Optimization step. (The length of a cycle is the number of nodes that compose the cycle.) Sometimes, a better layout can be obtained using a different choice of the outer cycle. This process of changing the outer cycle parameter and performing the layout again (see *Outer cycle (TML)*a) is a manual refinement procedure that can also be used to avoid overlapping nodes.

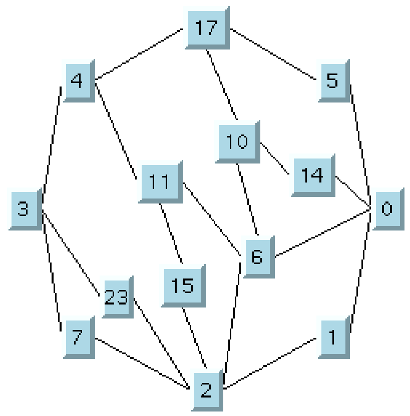
Note that performing the layout with a new outer cycle requires very little CPU time.

Using the layout region parameter (TML)

Often, overlapping nodes can be avoided by simply increasing the size of the layout region (see *Layout region (TML)*). *Layout with small layout region and overlapping nodes* shows a graph drawing where several nodes overlap because the layout region is too small for the graph. *Layout with larger layout region and no overlapping nodes* shows the same graph after increasing the size of the layout region. As you can see, now there are no overlapping nodes.



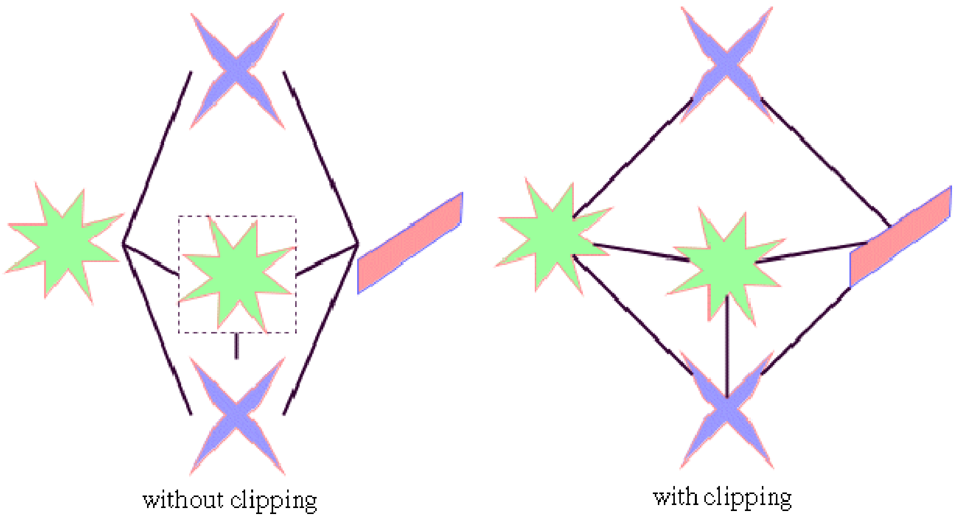
Layout with small layout region and overlapping nodes



Layout with larger layout region and no overlapping nodes

Using a link clipping interface with the TML

By default, TML does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of Link Clipping Interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*. To set a link clip interface.

To set a link clip interface:

In CSS
It is not possible to set the link clip interface. See *Link clipping*.

In Java™
Use the method:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: The link clip interface requires link connectors at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you use `IlvFreeLinkConnector` or `IlvClippingLinkConnector` for link connectors to be used in combination with `IlvGrapher` objects. The clip link connector updates the clipped connection points automatically during interactive node movements.

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface:

To set a link connection box interface:

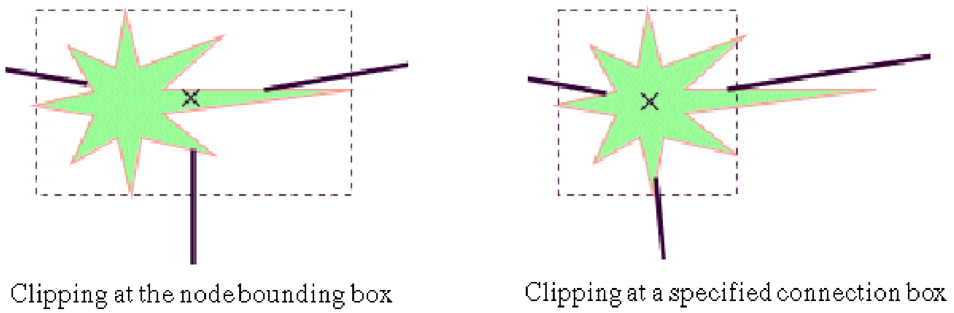
In CSS
It is not possible to set the link connection box interface. *Link connection box*

In Java
Use the method:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

The following figure shows an example of the combined effect.


Clipping at the node bounding box
Clipping at a specified connection box

Combined effect of link clipping interface and link connection box

If the links are clipped at the green irregular star node (see previous figure, left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (see previous figure, right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Force-directed or Uniform Length Edges Layout (ULEL)

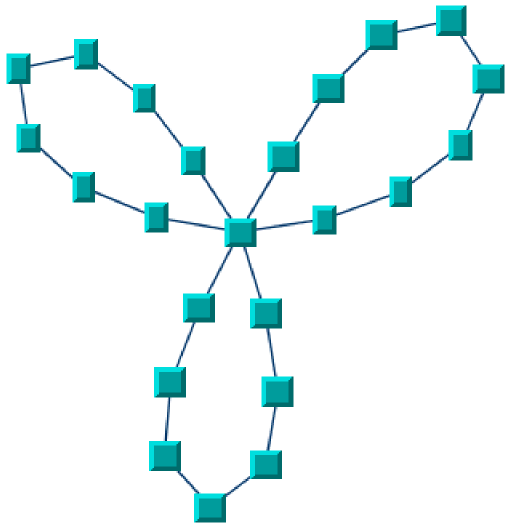
Describes the *Force-directed layout* or *Uniform Length Edges Layout* algorithm (class `IlvUniformLengthEdgesLayout` from the package `ilog.views.graphlayout.uniformlengthedges`).

In this section

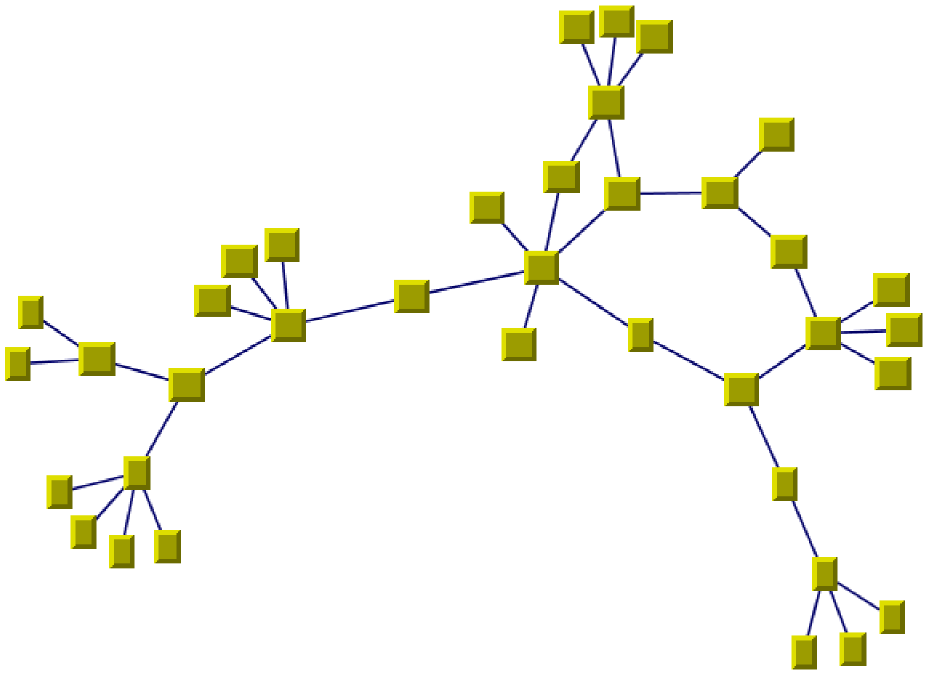
General information on the ULEL

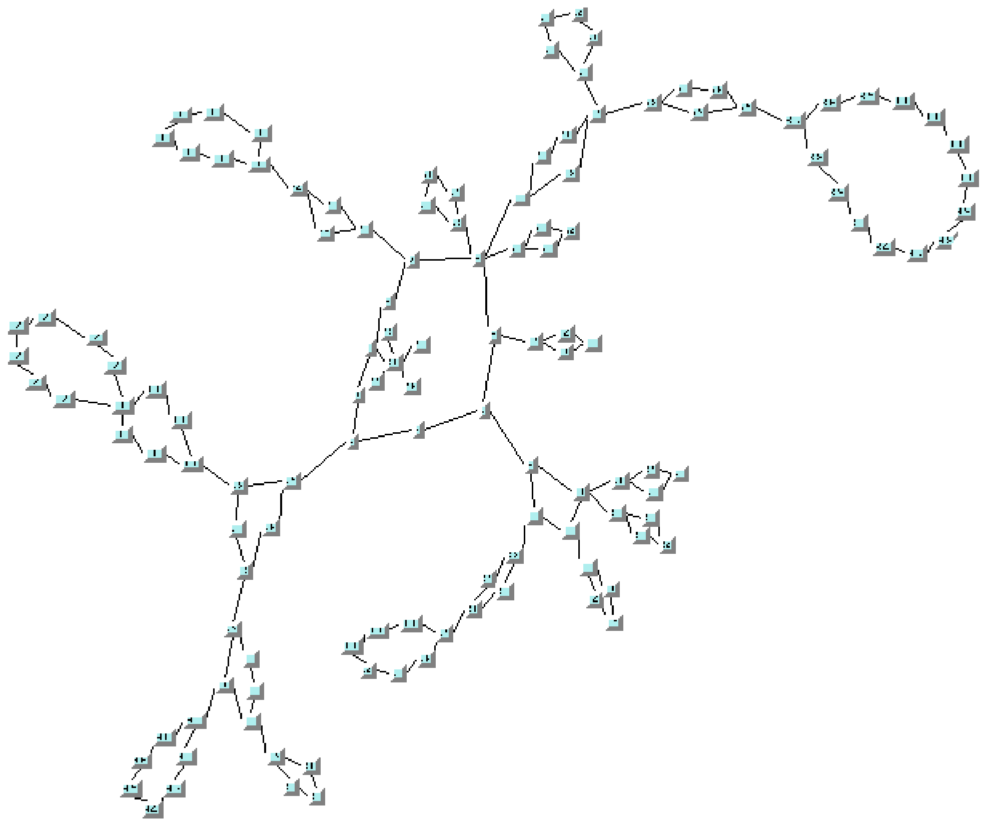
ULEL samples

The following sample drawings are produced with the Uniform Length Edges Layout (ULEL).

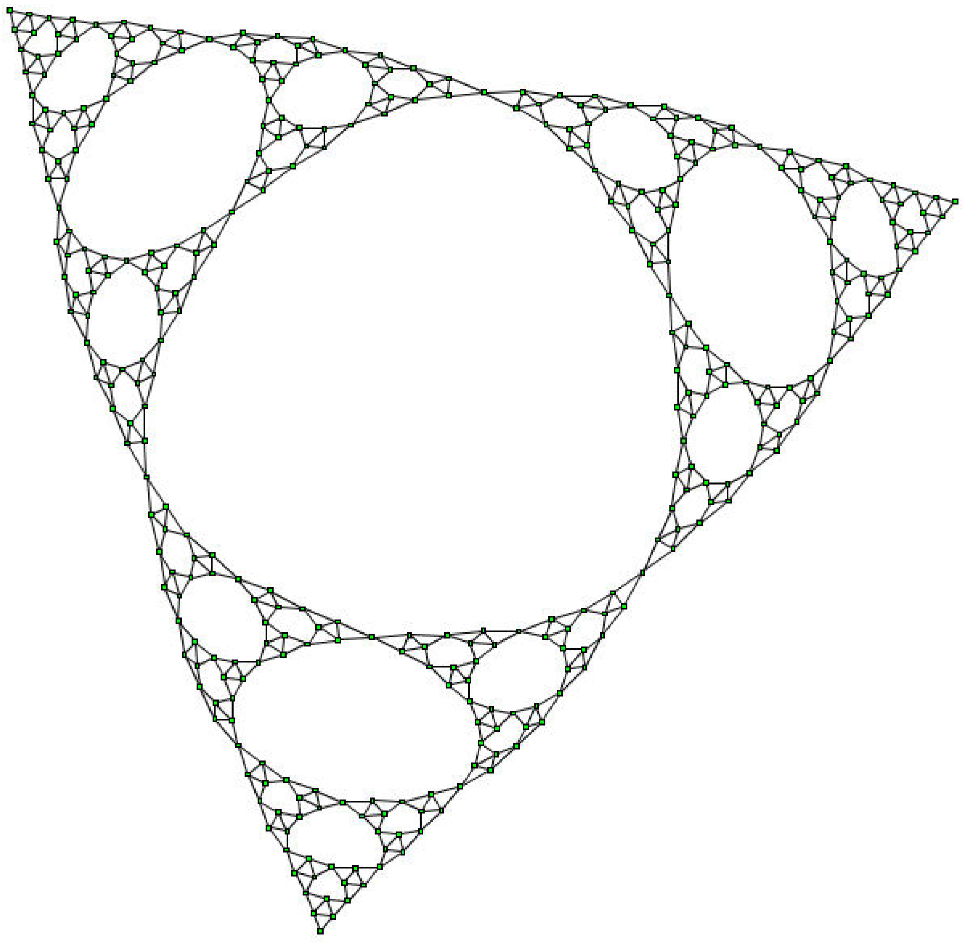


Small cyclic graph drawing produced with the Uniform Length Edges Layout


Medium graph drawing (combination of cycles and trees) produced with the Uniform Length Edges Layout



Large graph drawing (combination of cycles and trees) produced with the Uniform Length Edges Layout



Large graph drawing (Sierpinski Triangle) produced with the Uniform Length Edges Layout using the fast multilevel layout mode

What types of graphs suit the ULEL?

Any type of graph:

- ♦ connected graphs and disconnected graphs
- ♦ planar graphs and nonplanar graphs

Application domains for the ULEL

Application domains for the Uniform Length Edges Layout include:

- ♦ Telecoms and networking (WAN diagrams)

- Software management/software (re-)engineering (call graphs)
- CASE tools (dependency diagrams)
- Database and knowledge engineering (semantic networks, database query graphs, qualitative reasoning and other artificial intelligence diagrams, and so on)
- World Wide Web (Web hyperlink neighborhood)

Features and limitations of the ULEL

Features

Often provides a drawing without any or with only a few link crossings and with approximately equal length links for small- and medium-size graphs having a small number of cycles. The maximum number of nodes for which you can use the algorithm depends on the connectivity of the graph and is difficult to predict.

On demand, the algorithm can take into account the size (width and height) of the nodes. Otherwise, they are more efficiently considered as points.

It is possible to specify the length for each link individually.

The algorithm provides three optional layout modes: incremental, non-incremental and fast multilevel. The non-incremental and fast multilevel modes are in general faster and are recommended for large graphs. For details, see *Layout mode* .

Limitations

♦ The algorithm is not appropriate for all graphs. In particular, it will produce bad results on some highly connected cyclic graphs for which a planar drawing with equal-length links may simply not exist.

♦ The computation time required to obtain an appropriate drawing grows relatively quickly with the size of the graph (that is, the number of nodes and links) and the layout process may become time-consuming for large graphs.

♦ Overlapping nodes cannot always be avoided. Nevertheless, the layout algorithm often produces a drawing with no overlapping nodes.

The ULEL algorithm

This layout algorithm iteratively searches for a configuration of the graph where the length of the links is close to a user-defined or a default value.

Example of ULEL algorithm
In CSS
The following example of a specification in CSS uses the Uniform Length Edges Layout algorithm. Since the Uniform Length Edges Layout places nodes and reshapes the links, it is usually not necessary to specify an additional link layout in CSS. The specification in CSS can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see Graph Layout in IBM® ILOG® JViews Diagrammer).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

GraphLayout {
  graphLayout                : "UniformLengthEdges";
  linkStyle                  : "STRAIGHT_LINE_STYLE";
  preferredLinksLength       : "30";
  respectNodeSizes           : "true";
  layoutMode : "FAST_MULTILEVEL_MODE";
}
```

It is possible to enable the link layout additionally, and in this case, the link layout determines the shape of the links.

In Java™
The following code sample uses the `IlvUniformLengthEdgesLayout` class. This code sample shows how to perform a Uniform Length Edges Layout on a grapher directly without using a diagram component or any style sheet:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.uniformlengthedges.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*   Fill in the grapher with nodes and links here */

IlvUniformLengthEdgesLayout layout = new
                                        IlvUniformLengthEdgesLayout();
layout.attach(grapher);
try {
        IlvUniformLengthEdgesLayoutReport layoutReport =
                layout.performLayout();

        int code = layoutReport.getCode();
        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Important: All explanations in the subsequent sections regarding the shape of the links in Uniform Length Edges Layout are valid only if the link layout is disabled.

Generic features and parameters of the ULEL

Overview (ULEL)

The `IlvUniformLengthEdgesLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

♦ *Allowed time (ULEL)*

♦ *Animation (ULEL)*

♦ *Layout of connected components (ULEL)*

♦ *Layout region (ULEL)*

♦ *Link clipping (ULEL)*

♦ *Link connection box (ULEL)*

♦ *Preserve fixed links (ULEL)*

♦ *Preserve fixed nodes (ULEL)*

♦ *Save parameters to named properties (ULEL)*

♦ *Stop immediately (ULEL)*

The following subsections describe the particular way in which these parameters are used by this subclass.

Allowed time (ULEL)

The layout algorithm stops if the allowed time setting has elapsed. (See *Allowed time*.)

Animation (ULEL)

If animation is specified, a redraw is automatically performed after each step of the layout algorithm. (See *Animation*.)

Layout of connected components (ULEL)

The layout algorithm can utilize the generic mechanism to lay out connected components. (For more information about this mechanism, see *Layout of connected components*.)

Layout region (ULEL)

The layout algorithm can use the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout region*.)

Note that by default the Uniform Length Edges Layout algorithm does not use the layout region. (For details see also *Force fit to layout region (ULEL)*.)

Remember that if you are using the default settings, the visible area of the manager view (an instance of `IlvManagerView`) attached to the grapher is used as a layout region. If several manager views are attached, the first attached view is used. If no manager view is attached, the layout region is automatically estimated on the basis of the number and size of the nodes.

Link clipping (ULEL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the ULEL* for details of the link clipping mechanism.

Link connection box (ULEL)

The layout algorithm can use a link connection box interface (see *Link connection box*.) In combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link clipping interface with the ULEL*.

Preserve fixed links (ULEL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links* and *Link style (ULEL)*.)

Preserve fixed nodes (ULEL)

The layout algorithm does not move the nodes that are specified as fixed. Moreover, the algorithm takes into account the fixed nodes when computing the position of the nonfixed nodes. (See *Preserve fixed nodes*.)

Save parameters to named properties (ULEL)

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*).

Stop immediately (ULEL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the ULEL

The following parameters are specific to the `IlvUniformLengthEdgesLayout` class.

Link style (ULEL)

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

To specify that the ULEL algorithm automatically removes all the intermediate points of the links (if any):

In CSS
Add to the `GraphLayout` section:

```
linkStyle: "STRAIGHT_LINE_STYLE";
```

In Java™
Use this method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

♦ `IlvUniformLengthEdgesLayout.NO_RESHAPE_STYLE`

None of the links is reshaped in any manner.

♦ `IlvUniformLengthEdgesLayout.STRAIGHT_LINE_STYLE`

All the intermediate points of the links (if any) are removed. This is the default value.

Note: If you use CSS in a diagram component (instance of IlvDiagrammer), you must specify the correct link class in the style sheet. We recommend that you use `IlvGeneralLink` or `IlvSimpleLink` as classes for links in this case. If you call layout on an IlvGrapher directly in Java, you can use the method `EnsureAppropriateLinkTypes` or `EnsureAppropriateLinks` defined in the class `IlvGraphLayoutUtil` to replace inappropriate links automatically, either before layout or when the `IlvInappropriateLinkException` is caught. For details on these methods, see the *Java API Reference Manual*. For details on the graph model, see *Using the Graph Model*.

Note: If you call layout on an `IlvGrapher` directly in Java, you can use the method `EnsureAppropriateLinkTypes` or `EnsureAppropriateLinks` defined in the class `IlvGraphLayoutUtil` to replace inappropriate links automatically, either before layout or when the `IlvInappropriateLinkException` is caught. For details on these methods, see the *Java API Reference Manual*. For details on the graph model, see *Using the Graph Model*.

Number of iterations (ULEL)

The iterative computation of the layout algorithm is stopped if the time exceeds the allowed time (see *Allowed time*) or if the number of iterations exceeds the allowed number of iterations.

To specify the number of iterations:

In CSS
Add to the `GraphLayout` section:

```
allowedNumberOfIterations: "5";
```

In Java
Use the method:

```
void setAllowedNumberOfIterations(int iterations)
```

Preferred length (ULEL)

The main objective of this layout algorithm is to obtain a layout where all the links have a given length. This is called the “preferred length.”

To specify the preferred length:

Globally

♦ **In CSS**
Add to the `GraphLayout` section:

```
preferredLinksLength: "70.0";
```

Individually
It is also possible to specify a length for individual links. To do so:

♦ **In CSS**
Specify a rule that selects the link:

```
#link27 {
  PreferredLength: "80.0";
}
```

If a specific length is not specified for a link, the global settings are used.

Globally

♦ **In Java**
  Use the method:

```
void setPreferredLinksLength(float length)
```

  The default value is 60.0.

Individually
It is also possible to specify a length for individual links. To do so:

♦ **In Java**
  Use the method:

```
void setPreferredLength(Object link, float length)
```

  To obtain the current value, use the method:

```
float getPreferredLength(Object link)
```

If a specific length is not specified for a link, the global settings are used.

Respect node sizes (ULEL)

By default, the layout algorithm ignores the size (width and height) of the nodes. For efficiency reasons, the nodes are approximated with points placed in the center of the bounding box of the nodes. When dealing with large nodes, the preferred length parameter can be increased in such a way that the nodes do not overlap.

However, to improve the support for graphs with heterogeneous node sizes, the algorithm provides a special mode in which the particular size of each node is taken into consideration.

To set this mode:

In CSS
Add to the `GraphLayout` section:

```
respectNodeSizes: "true";
```

In Java
Use the method:

```
void setRespectNodeSizes(boolean respect)
```

The default value is `false`.

Force fit to layout region (ULEL)

For this layout algorithm, it is more difficult than for others to choose an appropriate size for the layout region. If the specified layout region is too small for a given graph, the resulting layout will not be the best. For this reason, by default, the Uniform Length Edges Layout algorithm does not use the layout region parameter. It can use as much space as it needs to lay out the grapher.

To specify whether the layout algorithm must use the layout region:

In CSS
Add to the `GraphLayout` section:

```
forceFitToLayoutRegion: "true";
```

In Java
Use the method:

```
void setForceFitToLayoutRegion(boolean option)
```

The default value of the parameter is `false`.

Layout mode

To fit a variety of needs, the algorithm provides three optional modes:

♦ Incremental mode

  The algorithm starts from the current position and iteratively tries to converge towards the optimal layout. Thus, in some cases, this mode avoids a major reorganization of the graph, which helps for preserving the "mental map" of the user as much as possible. However, this is not guaranteed, and depends on how far is the initial position of the nodes from the position that satisfies the criteria of the algorithm.

♦ Non-incremental mode

  The algorithm is free to reorganize the graph without trying to stay close to the initial positions. Often, the non-incremental mode is faster than the incremental mode, sometimes at the price of a lower quality.

♦ Fast multilevel mode

  The algorithm uses a multilevel graph decomposition strategy that leads to significant speed gain. This mode is usually the fastest for medium and large graphs.

To set this mode:

In CSS
Add to the `GraphLayout` section:

```
layoutMode: "FAST_MULTILEVEL_MODE";
```

In Java
Use the method:

```
void setLayoutMode(int mode)
```

The default value is `IlvUniformLengthEdgesLayout.INCREMENTAL_MODE`.

For experts: additional features of the ULEL

Expert users can also try and use the following parameters.

Maximum allowed move per iteration (ULEL)

At each iteration, the layout algorithm moves the nodes a relatively small amount. This amount should not be too large; otherwise the algorithm may not converge. But it should not be too small either, otherwise the number of necessary iterations increases and the running time does also.

The maximum amount of movement at each iteration is controlled by a parameter.

To set this parameter:

In CSS
Add to the `GraphLayout` section:

```
maxAllowedMovePerIteration: "10.0";
```

In Java™
Use the method:

```
void setMaxAllowedMovePerIteration(float maxMove)
```

Typical values for this setting are 1 to 30, but it depends on the value of the `PreferredLinksLength` parameter. For example, if the setting for the `PreferredLinksLength` parameter is 1000, then a value of 100 for the `MaxAllowedMovePerIteration` parameter is still meaningful.

Link length weight (ULEL)

The layout algorithm is based on the computation of attraction and repulsion forces for each of the nodes and the iterative search of an equilibrium configuration. One of these forces is related to the objective of obtaining a link length close to the specified preferred length. The weight of this force, representing the total amount of forces, is controlled by a parameter.

To set this parameter:

In CSS
Add to the `GraphLayout` section:

```
linkLengthWeight: "2.25" ;
```

In Java
Use the method:

```
void setLinkLengthWeight(float weight)
```

The default value is `1`. Increasing this parameter can help obtain link lengths closer to the specified length, but increasing too much can increase the number of link crossings.

Additional node repulsion weight (ULEL)

An additional repulsion force can be computed between nodes that are not connected by a link. The weight of this force, representing the total amount of forces, is controlled by a parameter.

To set this parameter:

In CSS
Add to the `GraphLayout` section:

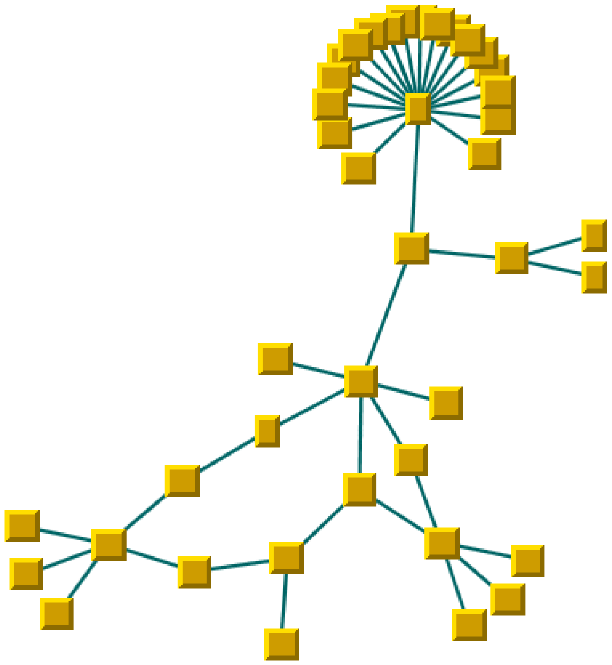
```
additionalNodeRepulsionWeight: "3.5";
```

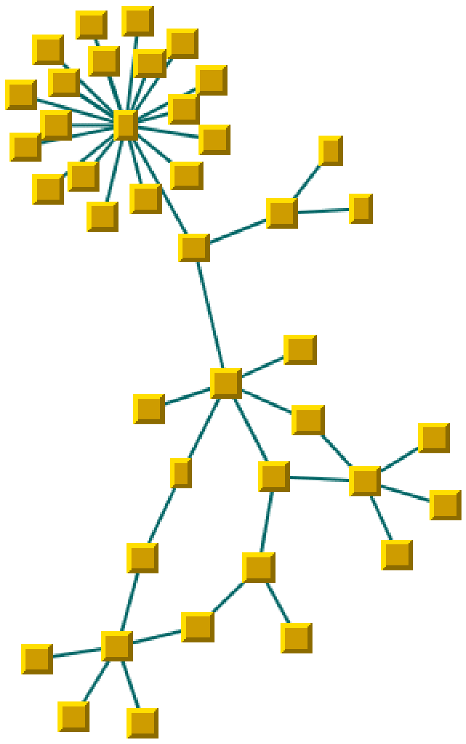
In Java
Use the methods:

```
void setAdditionalNodeRepulsionWeight(float weight)
```

The default value of this parameter is `0.2f`. Increasing (or decreasing) the weight increases (or decreases) the priority that is given to maintain the nodes at a distance larger than the node distance threshold (see `setNodeDistanceThreshold(float)`). On the other side, increasing the weight decreases the ability for the algorithm to reach convergence quickly.

The following two figures enable you to compare the same graph laid out with additional repulsion disabled (*Additional repulsion disabled, produced with the Uniform Length Edges Layout*) and then enabled (*Additional repulsion enabled, produced with the Uniform Length Edges Layout*). You can see that the “star” configuration, where many nodes are connected to the same central node, is better displayed when additional repulsion is enabled.


Additional repulsion disabled, produced with the Uniform Length Edges Layout


Additional repulsion enabled, produced with the Uniform Length Edges Layout

Node distance threshold (ULEL)

The additional repulsion force between two nodes not connected by a link is computed only when their distance is smaller than a predefined distance.

To set this distance:

In CSS
Add to the `GraphLayout` section:

```
nodeDistanceThreshold: "4.0";
```

In Java
Use the method:

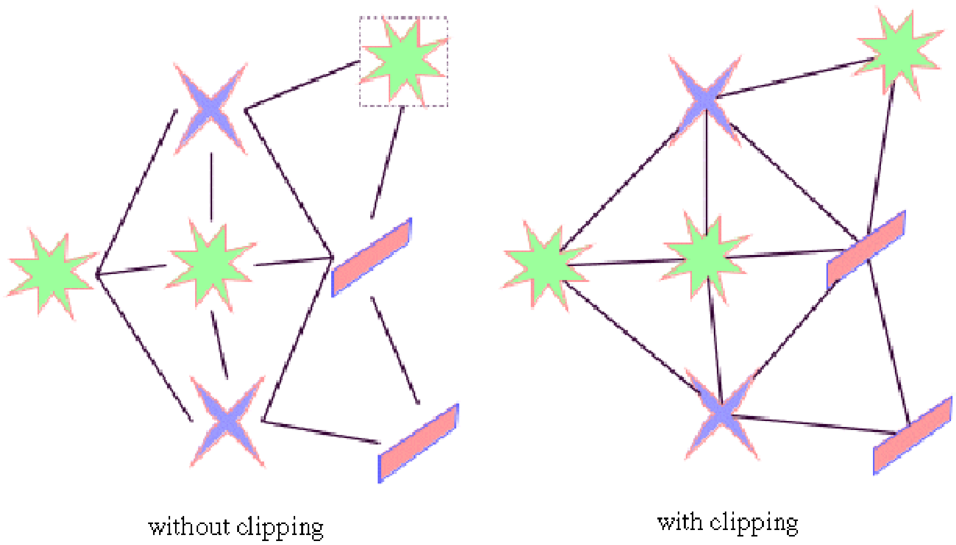
```
void setNodeDistanceThreshold(float threshold)
```

Note that this additional force is computed only if the “additional node repulsion weight” is set to a value larger than the default value `0`.

It is recommended that this threshold be set to a value smaller than the preferred length of the links.

Using a link clipping interface with the ULEL

By default, the Uniform Length Edges Layout does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of link clipping interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*.

To set a link clip interface:

In CSS
It is not possible to set the link clip interface. See *Link clipping*.

In Java™
Use the method

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: The link clip interface requires link connectors at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you use `IlvFreeLinkConnector` or `IlvClippingLinkConnector` for link connectors to be used in combination with `IlvGrapher` objects. The clip link connector updates the clipped connection points automatically during interactive node movements.

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*.

To set a link connection box interface:

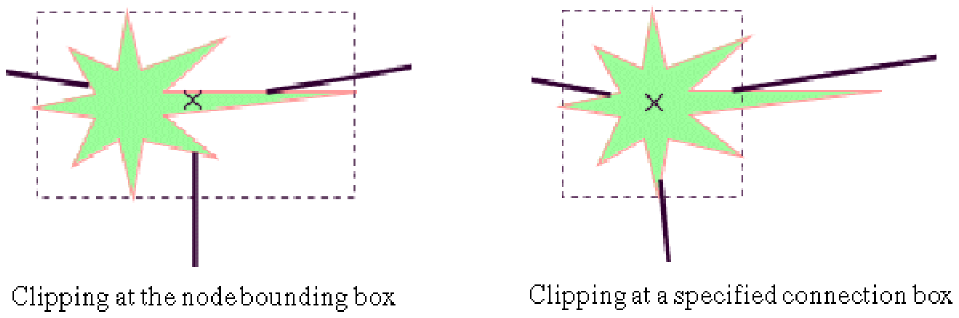
In CSS
It is not possible to set the link connection box interface. See *Link connection box*.

In Java
Use the method:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

The following figure shows an example of the combined effect.


Clipping at the node bounding box
Clipping at a specified connection box

Combined Effect of Link Clipping Interface and Link Connection Box

If the links are clipped at the green irregular star node (previous figure, left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (previous figure, right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Tree Layout (TL)

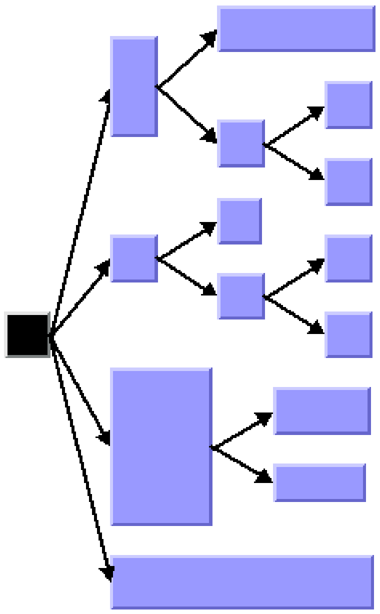
Describes the *Tree Layout* algorithm (class `IlvTreeLayout` from the package `ilog.views.graphlayout.tree`).

In this section

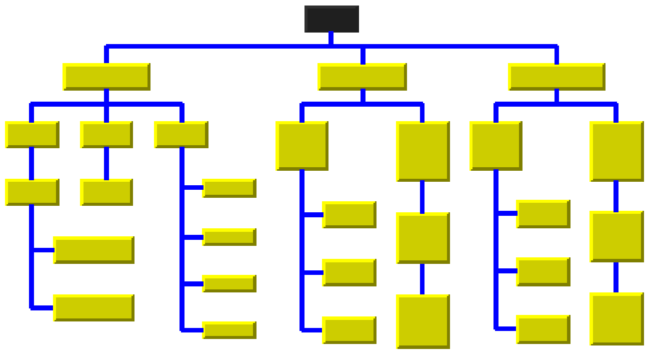
General information on the TL

TL samples

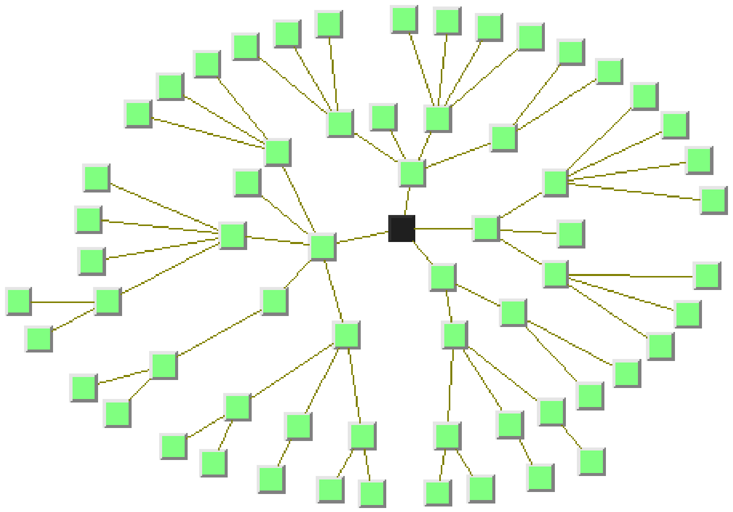
The following sample drawings are produced with the Tree Layout.



Tree layout in free layout mode with center alignment and flow direction to the right



Tree layout with flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes



Tree layout in radial layout mode with aspect ratio 1.5

What types of graphs suit the TL?

- ♦ Primarily designed for pure trees. It can also be used for non-trees, that is, for cyclic graphs. In this case, the algorithm computes and uses a spanning tree of the graph, ignoring all links that do not belong to the spanning tree.
- ♦ Directed and undirected trees. If the links are directed, the algorithm automatically chooses the canonical root node. If the links are undirected, you can choose a root node.
- ♦ connected graphs and disconnected graphs. If the graph is not connected, the layout algorithm treats each connected component separately. Each component has exactly one root node. In this case, a forest of trees is laid out.

Application domains for the TL

Application domains for the Tree Layout include:

- ♦ Business processing (organizational charts)
- ♦ Software management/software (re-)engineering (UML diagrams, call graphs)
- ♦ Database and knowledge engineering (decision trees)
- ♦ The World Wide Web (Web site maps)

Features and limitations of the TL

Features

- ♦ Takes into account the size of the nodes so that no overlapping occurs.
- ♦ Optionally reshapes the links to give them an orthogonal form (alternating horizontal and vertical line segments).
- ♦ Various layout modes: *free*, *levels*, *radial*, or *automatic tip-over*.
  - ● In the free layout mode, arranges the children of each node, starting recursively from the root, so that the links flow uniformly in the same direction.
  - ● In the level layout mode, partitions the nodes into levels, and arranges the levels horizontally or vertically.
  - ● In radial layout mode, partitions the nodes into levels, and arranges the levels in circles or ellipses around the root.
  - ● In the tip-over mode, arranges the nodes in a similar way to the free layout mode, but tries to tip children over automatically to fit the layout better to the given aspect ratio.
- ♦ Provides several alignment and offset options.
- ♦ Allows you to specify nodes that must be direct neighbors.
- ♦ Provides incremental and nonincremental modes. Incremental mode takes the previous position of nodes into account and positions the nodes without changing the relative order of the nodes in the tree so that the layout is stable on incremental changes of the graph.
- ♦ Very efficient, scalable algorithm. Produces a nice layout quickly even if the number of nodes is huge.

Limitations

- ♦ If the orthogonal setting is not specified as the link style (see *Link style*), some links may in rare cases overlap nodes depending on the size of the nodes, the alignment parameters, and the offset parameters.
- ♦ The layout algorithm first determines a spanning tree of the graph. If the graph is not a pure tree, some links will not be included as part of the spanning tree. These links are ignored. For this reason, they may cross other links or overlap nodes in the final layout.
- ♦ For stability in incremental mode, the algorithm tries to preserve the relative order of the children of each node. It uses a heuristic to calculate the relative order from the previous positions of the nodes. The heuristic may fail if children overlap their old positions or are not aligned horizontally or vertically.
- ♦ Despite preserving the relative order of the children, in rare cases the layout is not perfectly stable in incremental radial layouts. Subsequent layouts may rotate the nodes around the root, although the relative circular order of the nodes within their circular levels is still preserved.

- The tip-over layout modes will perform several trial layouts with different tip-over alignment options according to various heuristics. From these trial layouts, the algorithm picks the layout that best fits the given aspect ratio. This may not be the optimal layout for the aspect ratio, but it is the best layout among the trials. Calculating the absolute best-fitting layout is not computationally feasible (it is generally an NP-complete problem).

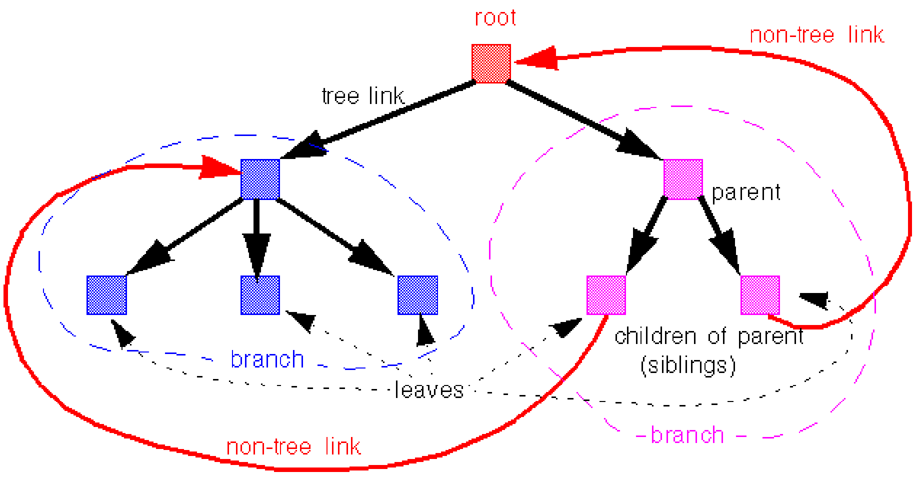
The TL algorithm

The core algorithm for the free, level, and radial layout modes works in just two steps and is very fast. The variations of the tip-over layout mode perform the second step several times and pick the layout result that best fits the given aspect ratio (the ratio between width and height of the drawing area). For this reason, the tip-over layout modes are slower.

Step 1: Calculating the spanning tree

If the graph is disconnected, the layout algorithm chooses a *root node* for each connected component. Starting from the root node, it traverses the graph to choose the links of the *spanning tree*. If the graph is a pure tree, all links are chosen. If the graph has cycles, some links will not be included as part of the spanning tree. These links are called *non-tree links*, while the links of the spanning tree are called *tree links*. The non-tree links are ignored in step 2 of the algorithm.

In *Tree layout in free layout mode with center alignment and flow direction to the right*, *Tree layout with flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes*, and *Tree layout in radial layout mode with aspect ratio 1.5*,the *root* is the node that has no parent node. In the spanning tree, each node except the root has a parent node. All nodes that have the same parent are called *children* with respect to the parent and *siblings* with respect to themselves. Nodes without children are called *leaves*. Each child at a node starts a *subtree* (also called a branch of the tree). *A Spanning tree* show an example of a spanning tree.



A Spanning tree

Step 2: Calculating node positions and link shapes

The layout algorithm arranges the nodes according to the layout mode and the offset and alignment options. In the free mode and level mode, the nodes are arranged horizontally or vertically so that all tree links flow roughly in the same direction. In the radial layout modes, the nodes are arranged in circles or ellipses around the root so that all tree links flow radially away from the root. Finally, the link shapes are calculated according to the link style and alignment options.

Example of TL

In CSS

The following example is a specification in CSS using the Tree Layout algorithm. Since the Tree Layout places nodes and links, it is usually not necessary to specify an additional link layout in CSS. The specification in CSS can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see Graph Layout in IBM® ILOG® JViews Diagrammer).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

GraphLayout {
    graphLayout       : "Tree";
    layoutMode        : "FREE";
    flowDirection     : "Bottom";
    globalLinkStyle   : "ORTHOGONAL_STYLE";
    globalAlignment   : "CENTER";
    connectorStyle    : "EVENLY_SPACED_PINS";
    siblingOffset     : "15";
    branchOffset      : "30";
    parentChildOffset : "20";
    position          : "200,20";
}
```

However, it is possible to enable the link layout additionally and in this case, the link layout determines the shapes of the links.

In Java™

The following code sample uses the `IlvTreeLayout` class in Java. This code sample shows how to perform a Tree Layout on a grapher directly without using a diagram component or any style sheet:

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.tree.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /* Fill in the grapher with nodes and links here */
 ... /* Suppose we have added rootNode as a node in the grapher */

IlvTreeLayout layout = new IlvTreeLayout();
layout.attach(grapher);

/* Specify the root node, orientation and alignment */
layout.setRoot(rootNode);
layout.setFlowDirection(IlvDirection.Right);
layout.setGlobalAlignment(IlvTreeLayout.CENTER);
```

```
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Important: All explanations in the subsequent sections regarding the shape of the links in Tree Layout are valid only if the link layout is disabled.

Generic features and parameters of the TL algorithm

Overview (TL)

The `IlvTreeLayout` class supports the following generic features defined in the `IlvGraphLayout` class (see also *Base class parameters and features*):

- *Allowed time (TL)*
- *Layout of connected components (TL)*
- *Link clipping (TL)*
- *Link connection box (TL)*
- *Spline routing (TL)*
- *Percentage of completion calculation (TL)*
- *Preserve fixed links (TL)*
- *Preserve fixed nodes (TL)*
- *Save parameters to named properties (TL)*
- *Stop immediately (TL)*

The following subsections describe the particular way in which these features are used by the subclass `IlvTreeLayout`.

Allowed time (TL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components (TL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*). It has, however, a specialized internal mechanism to layout connected components and, therefore, the generic mechanism is switched off by default.

The generic connected component layout mechanism has the disadvantage that it moves connected components completely. Fixed nodes within a component do not preserve their old position, and the resulting layout may be unstable on incremental changes, depending on which layout instance is used for the component layout.

If the generic connected component layout mechanism is disabled, the algorithm uses its own specialized internal mechanism instead of the generic mechanism to lay out each component as a separate tree. This is usually faster and more stable on incremental changes than the generic mechanism. Furthermore, it enables the user to set the position of the layout.

Link clipping (TL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface* for details of the link clipping mechanism.

Link connection box (TL)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link connection box interface*.

Spline routing (TL)

The layout algorithm supports the generic spline routing mechanism (see *Spline routing*). If the style of a link is orthogonal and the link is a spline, it is routed by the generic spline routing mechanism when it is enabled.

Percentage of completion calculation (TL)

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of layout. (For a detailed description of this feature, see *Percentage of completion calculation* and *Graph layout event listeners*.)

Preserve fixed links (TL)

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve fixed links* and *Link style (TML)*.)

Preserve fixed nodes (TL)

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve fixed nodes*.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in unwanted overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Save parameters to named properties (TL)

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*.)

Stop immediately (TL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call, and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters (for all tree layout modes)

The following parameters are specific to the `IlvTreeLayout` class. They apply to all layout modes.

Root node (TL)

The final layout is influenced mainly by the choice of the root node.

The root node is placed in a prominent position. For instance, in a top-down drawing with free layout mode, it is placed at the top of the tree. With the radial layout mode, it is placed at the center of the tree.

The spanning tree is calculated starting from the root node. If the graph is disconnected, the layout algorithm needs one root node for each connected component.

The layout algorithm automatically selects a root node when needed. It uses a heuristic that calculates preferences for all nodes to become a root. It chooses the node with the highest preference. The heuristic gives nodes without incoming links the highest preference and leaf nodes without outgoing links the lowest preference. Hence, in a directed tree, the canonical root is always chosen automatically.

It is possible to influence the choice of the root node.

To set a node explicitly as the root:

In CSS
Specify a rule that selects the node, for instance:

```
#node1 {
  Root: "true";
}
```

In Java™
Use the method:

```
void setRoot(Object node);
```

In this case, the node argument must be a graphic node (subclass of `IlvGraphic`).

This gives the node the maximal preference to become the root during layout. If only one node is specified this way, the algorithm selects this node. If several nodes of the same connected component are specified this way, the layout algorithm chooses one of them as the root.

For experts: additional options for root nodes (TL)

The layout algorithm manages a list of the root nodes that have been specified by the `setRoot` method. To obtain the nodes in this list in Java, use the method:

```
Enumeration getSpecRoots();
```

After layout, you can also retrieve the list of root nodes that were actually used by the algorithm. This list is not necessarily the same as the list of specified roots. For instance, it contains the chosen root nodes if none were specified or if too many were specified. To obtain the root nodes that were used by the algorithm in Java, use the method:

```
Enumeration getCalcRoots();
```

This example shows how to iterate over the calculated root nodes and print the root node preferences:

```
Enumeration e = layout.getCalcRoots();
while (e.hasMoreElements()) {
  node = e.nextElement();
  System.out.println("Preference:" + layout.getRootPreference(node));
}
```

To directly manipulate the root node preference value of an individual node:

In CSS
Write a rule to select the node:

```
#node1 {
  RootPreference: "1000";
}
```

In Java
Use the method:

```
setRootPreference(Object node, int preference);
```

In this case, the layout uses the specified value instead of the heuristically calculated preference for the node. The normal preference value should be between `0` and `10000`. Specifying a root node explicitly corresponds to setting the preference value to `10000`. If you want to prohibit a node from becoming the root, specify a preference value of zero (`0`).

A negative preference value indicates that the layout algorithm should recalculate the root node preference using the heuristic. If a root was specified by the `setRoot` method but this node should no longer be the root in subsequent layouts, use the following call to clear the root node setting:

```
layout.setRootPreference(node, -1);
```

This call also removes the node from the list of specified roots.

Position parameters (TL)

To set the position of the *top left corner* of the layout to (10, 10):

In CSS
Specify:

```
GraphLayout {
    position          : "10,10";
    rootPosition      : "false";
}
```

In Java
In Java, use the method:

```
layout.setPosition(new IlvPoint(10, 10), false);
```

If the graph consists of only a single tree, it is often more convenient to set the position of the root node instead. To do this:

In CSS
Specify in the `GraphLayout` section:

```
GraphLayout {
    position          : "10,10";
    rootPosition      : "true";
}
```

In Java
Use the same method and pass `true` instead of `false`:

```
layout.setPosition(point, true);
```

If no position is specified, the layout keeps the root node at its previous position.

Using compass directions for positional layout parameters (TL)

The compass directions *north, south, east*, and *west* are used to simplify the explanations of the layout parameters. The center of the root node of a tree is considered the north pole.

In the nonradial layout modes, the link flow direction always corresponds to south. If the root node is placed at the top of the drawing, north is at the top, south at the bottom, east to the right, and west to the left. If the root node is placed at the left border of the drawing, north is to the left, south to the right, east at the top, and west at the bottom.

In the radial layout modes, the root node is placed in the center of the drawing. The meaning of north and south depends on the position relative to the root: the north side of the node is the side closer to the root and the south side is the side further away from the root. The east direction is counterclockwise around the root and the west direction is clockwise around the root. This is similar to a cartographic map of a real globe that shows the area of the north pole as if you were looking down at the top of the globe.

Compass directions are used to provide uniform naming conventions for certain layout options. They occur in the alignment options, the level alignment option, and the east-west neighboring feature, which are explained later. In *Flow directions* and *Radial layout mode*, the compass icons show the compass directions in these drawings.

Layout modes (TL)

The tree layout algorithm has several layout modes. The following example shows how to specify the layout mode.

In CSS

Add to the `GraphLayout` section:

```
layoutMode: "FREE";
```

In Java

Use the method:

```
void setLayoutMode(int mode);
```

The available layout modes are the following:

- `IlvTreeLayout.FREE` (the default)
- `IlvTreeLayout.LEVEL`
- `IlvTreeLayout.RADIAL`
- `IlvTreeLayout.ALTERNATING_RADIAL`
- `IlvTreeLayout.TIP_OVER`
- `IlvTreeLayout.TIP_ROOTS_OVER`
- `IlvTreeLayout.TIP_LEAVES_OVER`
- `IlvTreeLayout.TIP_ROOTS_AND_LEAVES_OVER`

In CSS, you omit the prefix `IlvTreeLayout` when specifying the value of the layout mode.

Layout modes of the TL algorithm

Describes the characteristics and the layout parameters of each layout mode in the TL algorithm.

In this section

Free layout mode

Describes how the free layout mode organizes nodes and describes the parameters of this mode.

In this section

Overview

The free layout mode arranges the children of each node starting recursively from the root so that the links flow roughly in the same direction. For instance, if the link flow direction is top-down, the root node is placed at the top of the drawing. Siblings (nodes that have the same parent) are justified at their top borders, but nodes of different tree branches (nodes with different parents) are not justified.

To set the free layout mode:

In CSS
Add to the `GraphLayout` section:

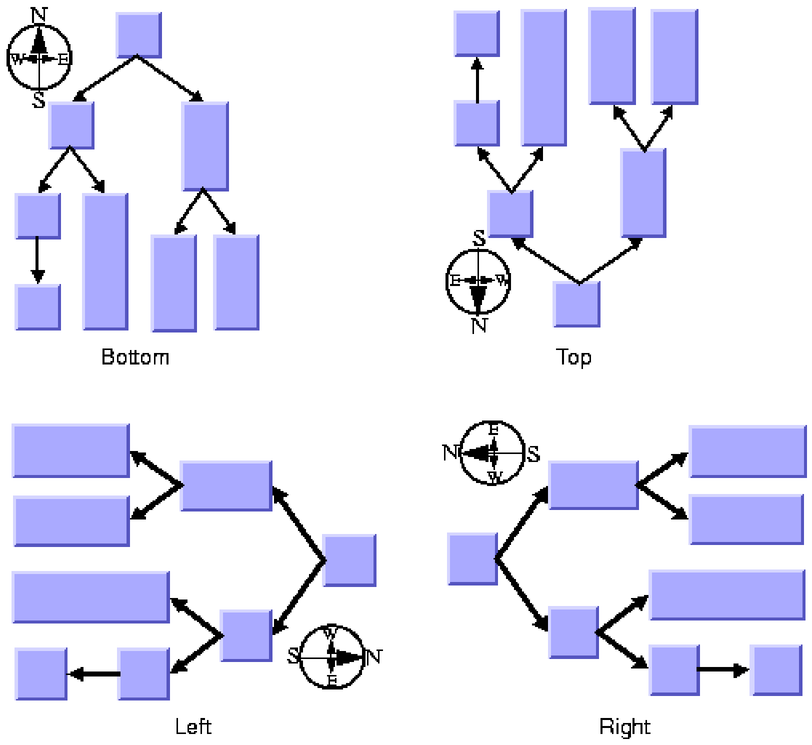
```
layoutMode: "FREE";
```

In Java™
Call:

```
layout.setLayoutMode(IlvTreeLayout.FREE);
```

Flow direction

The flow direction parameter specifies the direction of the tree links. The compass icons show the compass directions in these layouts.



Flow directions

If the flow direction is to the bottom, the root node is placed topmost. Each parent node is placed above its children, which are normally arranged horizontally. (This tip-over alignment is an exception.)

If the flow direction is to the right, the root node is placed leftmost. Each parent node is placed to the left of its children, which are normally arranged vertically.

To specify the flow direction:

In CSS
Add to the `GraphLayout` section, for instance

```
flowDirection: "Left";
```

In Java
In Java™ , use the method:

```
void setFlowDirection(int direction);
```

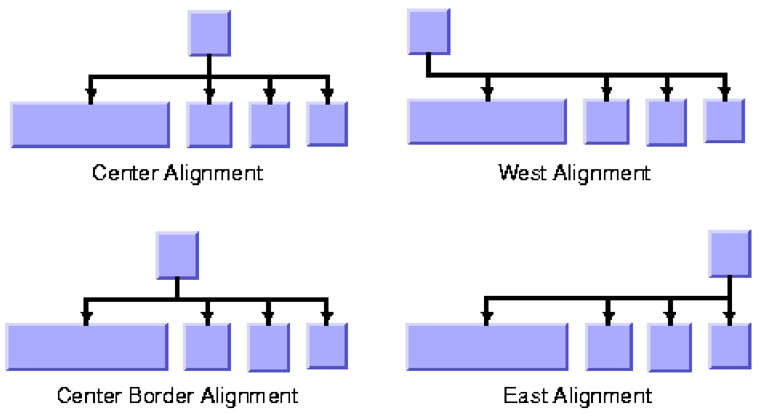
The valid values for the flow direction are:

- `IlvDirection.Right` (the default)
- `IlvDirection.Left`
- `IlvDirection.Bottom`
- `IlvDirection.Top`

In CSS, you omit the prefix `IlvDirection` when specifying the value of the flow direction.

Alignment parameter

The alignment option controls how a parent is placed relative to its children. The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.



Alignment Options

Global alignment

To set the global alignment:

In CSS
Add to the GraphLayout section, for instance:

```
globalAlignment: "CENTER";
```

In Java
In Java™ , use the method:

```
void setGlobalAlignment(int alignment);
```

The valid values for the global alignment are:

- `IlvTreeLayout.CENTER` (the default)

  The parent is centered over its children, taking the center of the children into account.

- `IlvTreeLayout.BORDER_CENTER`

  The parent is centered over its children, taking the border of the children into account. If the size of the first and the last child varies, the border center alignment places the parent closer to the larger child than to the default center alignment.

- `IlvTreeLayout.EAST`

The parent is aligned with the border of its easternmost child. For instance, if the flow direction is to the bottom, east is the direction to the right. If the flow direction is to the top, east is the direction to the left. See *Using compass directions for positional layout parameters (TL)* for details.

- ♦ `IlvTreeLayout.WEST`

  The parent is aligned with the border of its westernmost child. For instance, if the flow direction is to the bottom, west is the direction to the left. If the flow direction is to the right, west is the direction to the bottom. See *Using compass directions for positional layout parameters (TL)* for details.

- ♦ `IlvTreeLayout.TIP_OVER`

  The children are arranged sequentially instead of in parallel, and the parent node is placed with an offset to the children. For details see *Tip-over alignment*.

- ♦ `IlvTreeLayout.TIP_OVER_BOTH_SIDES`

  The children are arranged sequentially instead of in parallel. Whereas the alignment `TIP_OVER` arranges all children at the same side of the parent, this alignment arranges the children at both sides of the parent. For details see *Tip-over alignment*.

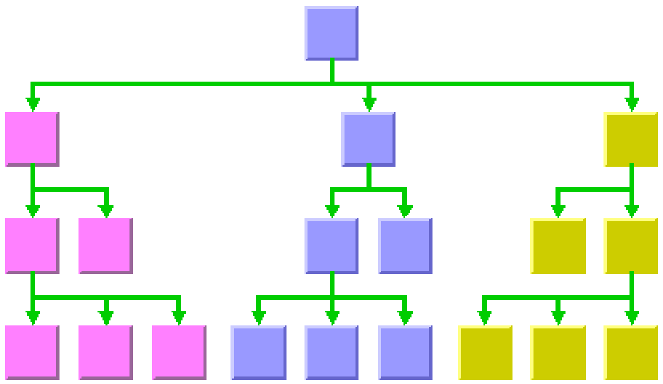
- ♦ `IlvTreeLayout.MIXED`

  Each parent node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

In CSS, you omit the prefix `IlvTreeLayout` when specifying the value of the alignment.

Alignment of individual nodes

All nodes have the same alignment unless the global alignment is set to `MIXED`. Only when the global alignment is set to `MIXED` can each node have an individual alignment style.



Different Alignments Mixed in the Same Drawing

To specify the alignment of an individual node:

In CSS
First set the global alignment to MIXED, then specify a rule that selects the node, for instance:

```
GraphLayout {
   globalAlignment: "MIXED";
}

#node1 {
  Alignment: "EAST";
}
```

In Java

Use the methods:

```
void setAlignment(Object node, int alignment);
```

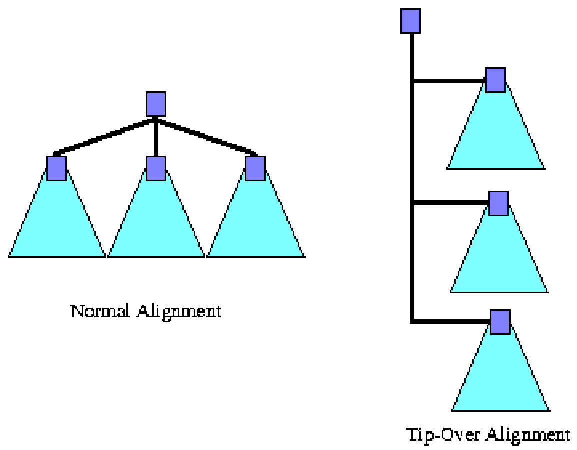
```
int getAlignment(Object node);
```

The valid values for `alignment` are:

- ♦ `IlvTreeLayout.CENTER` (the default)
- ♦ `IlvTreeLayout.BORDER_CENTER`
- ♦ `IlvTreeLayout.EAST`
- ♦ `IlvTreeLayout.WEST`
- ♦ `IlvTreeLayout.TIP_OVER`
- ♦ `IlvTreeLayout.TIP_OVER_BOTH_SIDES`

Tip-over alignment

Normally, the children of a node are placed in a *parallel* arrangement with siblings as direct neighbors of each other. Tip-over alignment means a *sequential* arrangement of the children instead.



Normal alignment and tip-over alignment

Tip-over alignment is useful when the tree has many leaves. With normal alignment, a tree with many leaves would result in the layout being very wide. If the global alignment style is set to tip-over, the drawing is very tall rather than wide. To balance the width and height of the drawing, you can set the global alignment to mixed, for example, in Java:

```
layout.setGlobalAlignment(IlvTreeLayout.MIXED);
```

Also, you can set the individual alignment to tip-over for some parents with a high number of children as follows:

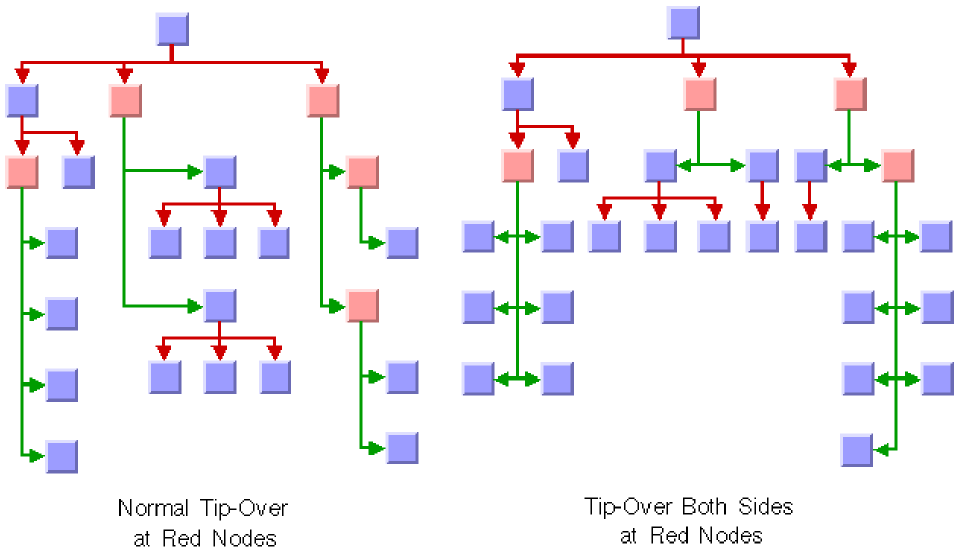
```
layout.setAlignment(parent, IlvTreeLayout.TIP_OVER);
```

Tip-over alignment can be specified explicitly for some or all of the nodes. Furthermore, the Tree Layout offers layout modes that automatically determine when to tip over, yielding a drawing that fits into a given aspect ratio. These layout modes are described in *Tip-over layout modes*.

Besides the normal tip-over alignment, there is also a variant that distributes the subtrees on both sides of the center line that starts at the parent. You can specify this variant at a parent node with a high number of children by the following code:

```
layout.setAlignment(parent, IlvTreeLayout.TIP_OVER_BOTH_SIDES);
```

The following figure illustrates the difference between normal tip-over alignment and tip-over at both sides. Tip-over alignment works very well with the orthogonal link style (see *Link style*).



Tip-over alignment

Link style

When using the Tree layout, it is preferable to use link connectors of type `IlvFreeLinkConnector` and links of type `IlvPolylineLinkImage` or `IlvSplineLinkImage`. The links can be straight or have a specific shape with intermediate points. You can specify that the links be reshaped into an “orthogonal” form. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Link style and link shapes

Link styles work only when you use links that can be reshaped. Subclasses of `IlvPolylineLinkImage` or of `IlvSplineLinkImage`, (such as `IlvGeneralLink`) can be reshaped. Furthermore, link styles work only if free link connectors are installed. Free link connectors are subclasses of `IlvFreeLinkConnector`. If you use a diagram component, the free link connectors are automatically installed when necessary unless specified differently. If you call layout in Java™ on an `IlvGrapher` instance directly, the layout algorithm may raise an `IlvInappropriateLinkException` if links are neither a subclass of `IlvPolylineLinkImage` nor of `IlvSplineLinkImage`, or if connectors are not a subclass of `IlvFreeLinkConnector`. In this case, you can use the methods `EnsureAppropriateLinkTypes`, `EnsureAppropriateLinkConnectors`, or `EnsureAppropriateLinks` defined in `IlvGraphLayoutUtil` to replace inappropriate links or link connectors automatically, either before layout or when the `IlvInappropriateLinkException` is caught. For details on these methods, see the *Java API Reference Manual*. For details on the graph model, see *Using the Graph Model*.

Global link style

To specify the global link style:

In CSS
Add this statement to the `GraphLayout` section:

```
globalLinkStyle: "STRAIGHT_LINE_STYLE";
```

In Java
Use the method:

```
void setGlobalLinkStyle(int style);
```

The valid values for `style` are:

♦ `IlvTreeLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

♦ `IlvTreeLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value. See *Tree layout in free layout mode with center alignment and flow direction to the right* and *Tree layout in radial layout mode with aspect ratio 1.5* as examples.

- `IlvTreeLayout.ORTHOGONAL_STYLE`

  The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See *Tree layout with flow direction to the bottom, orthogonal link style, and tip-over alignment at some leaf nodes* and*Tip-over alignment* as examples.

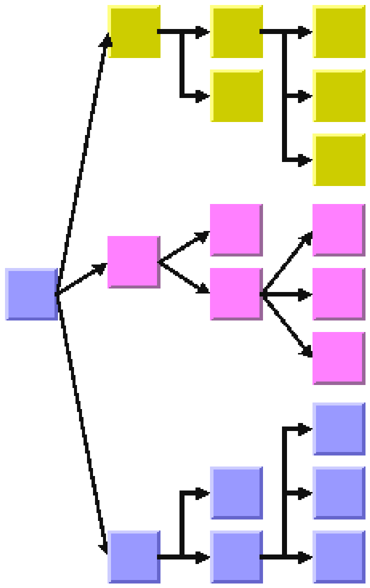
- `IlvTreeLayout.MIXED_STYLE`

  Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

In CSS, you omit the prefix `IlvTreeLayout` when specifying the value of the link style.

Individual link style

All links have the same style of shape unless the global link style is `MIXED_STYLE`. Only when the global link style is set to `MIXED_STYLE` can each link have an individual link style.



Different Link Styles Mixed in the Same Drawing

To specify the style of an individual link:

In CSS
First set the global link style to MIXED_STYLE, then specify a rule that selects the link, for instance:

```
GraphLayout {
   globalLinkStyle: "MIXED_STYLE";
}
#link1{
  LinkStyle: "ORTHOGONAL_STYLE";
}
```

In Java

Use the methods:

```
setLinkStyle(java.lang.Object, int)

getLinkStyle
```

The valid values for `style` are:

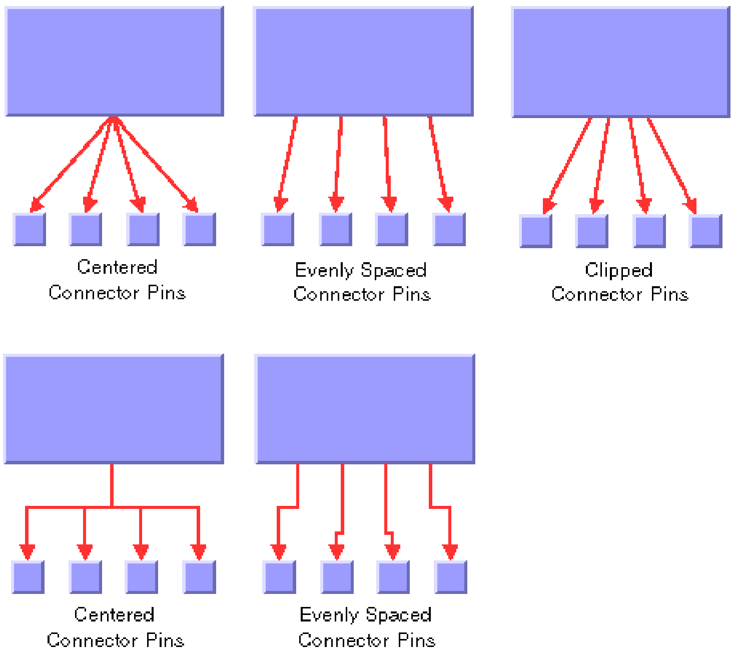
- ♦ `IlvTreeLayout.STRAIGHT_LINE_STYLE` (the default)
- ♦ `IlvTreeLayout.NO_RESHAPE_STYLE`
- ♦ `IlvTreeLayout.ORTHOGONAL_STYLE`

Note: The link style of a Tree Layout graph requires links in an `IlvGrapher` that can be reshaped. Links of type `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, and `IlvDoubleSplineLinkImage` cannot be reshaped. You should use the class `IlvPolylineLinkImage` or `IlvSplineLinkImage` instead.

Connector style

The layout algorithm automatically positions the end points of links (the connector pins) at the nodes. The connector style parameter specifies how these end points are calculated for the outgoing links at the parent node.

By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side.



Connector styles

To specify the connector style:

In CSS
Add to the `GraphLayout` section:

```
connectorStyle: "CLIPPED_PINS";
```

In Java™
Use the method:

```
void setConnectorStyle(int style);
```

The valid values for `style` are:

- `IlvTreeLayout.CENTERED_PINS`

  The end points of the links are placed in the center of the border where the links are attached.

- `IlvTreeLayout.CLIPPED_PINS`

  Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This style affects straight links. It behaves like centered connector pins for orthogonal links.

- `IlvTreeLayout.EVENLY_SPACED_PINS`

  The connector pins are evenly distributed along the node border. This style works for straight and orthogonal links.

- `IlvTreeLayout.AUTOMATIC_PINS`

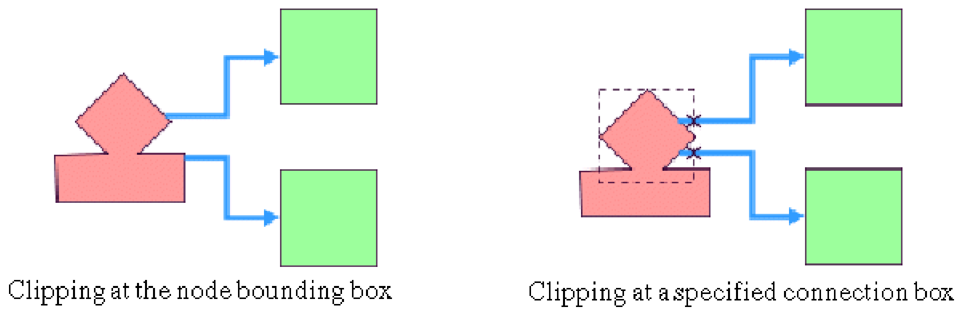
  The connector style is selected automatically depending on the link style and the layout mode. In the nonradial modes, the algorithm always chooses centered pins. In the radial layout modes, it chooses clipped pins.

In CSS, you omit the prefix `IlvTreeLayout` when specifying the value of the connector style.

Note: The connector style parameter requires link connectors at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you use `IlvFreeLinkConnector` for link connectors to be used in combination with `IlvGrapher` objects. If you use a diagram component, the free link connectors are automatically installed when needed, unless specified differently.

The connector style, the link connection box interface, and the link clip interface work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box interface (or on the bounding box of the node, if no link connection box interface was specified). Then, the proposed connection point is passed to the link clip interface and the returned connection points are used to connect the link to the node.

The following figure shows an example of the combined effect.


Clipping at the node bounding box
Clipping at a specified connection box

Combined effect of link clipping interface and link connection box

If the links are clipped at the red node in the previous figure (left), they appear unsymmetrical with respect to the node shape, because the relevant part of the node (here: the upper rhombus) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box interface to explicitly specify a smaller connection box for the relevant part of the node (previous figure, right) such that the proposed connection points are placed symmetrically at the upper rhombus of the node.

Using a link connection box interface

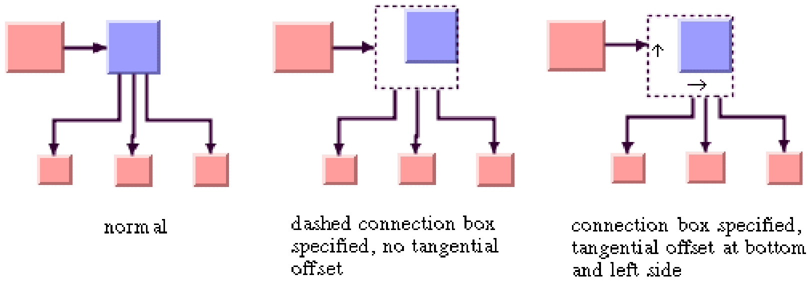
Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, possibly in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface in Java™ , call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface provides each node with a link connection box and a tangential shift offset that defines how much the connection points are “shifted” tangentially depending on which side the links connect.

The following figure illustrates the effects of customizing the connection box when the connector style is evenly spaced.


normal
dashed connection box specified, no tangential offset
connection box specified, tangential offset at bottom and left side

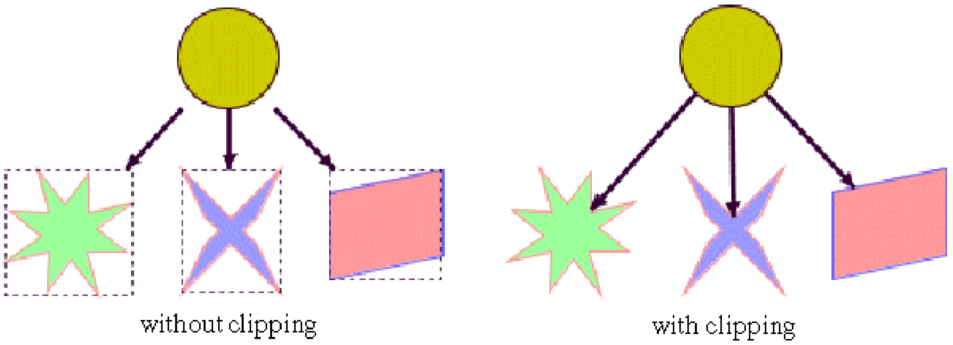
Effect of connection box interface

On the left is the result without any connection box interface. The middle picture shows the effect if the connection box interface returns the dashed rectangle for the blue node but the tangential offset at all sides of the node is 0. Notice that the outgoing links are spaced according to the dashed rectangle, which appears too wide for the blue node in this situation. The picture on the right shows the effect of the connection box interface if, in addition, a positive tangential offset was specified for the bottom side and a negative offset was specified for the left side of the blue node.

Using a link clipping interface

By default, the Tree Layout places the connection points of links at the border of the bounding box of the nodes.

If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of link clipping interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*. To set a link clip interface in Java™ , call:

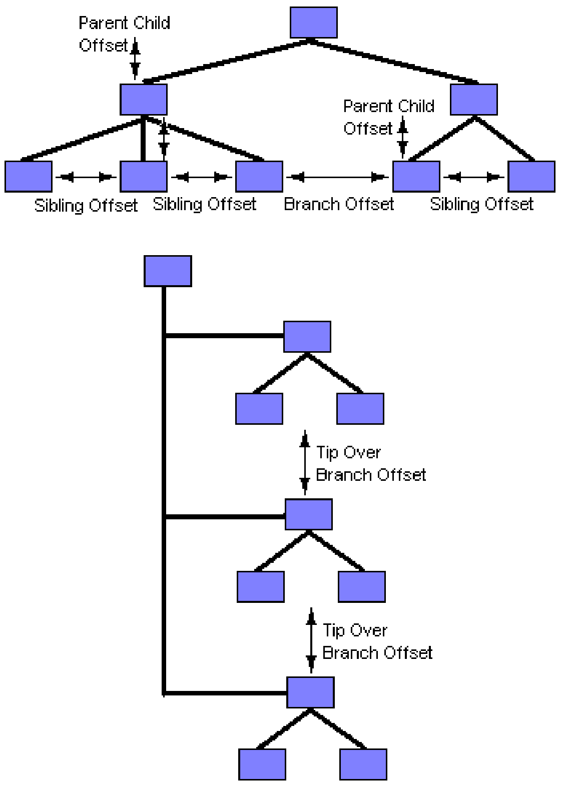
```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: In addition to the link clip interface, you can use the `IlvClippingLinkConnector`. This special link connector updates the clipped connection points automatically during interactive node movements.

Spacing parameters

The spacing of the layout is controlled mainly by three spacing parameters: the distance between a parent and its children, the minimum distance between siblings, and the minimum distance between nodes of different branches. For instance, if the flow direction is to the top or bottom, the offset between parent and children is vertical, while the sibling offset and the branch offset are horizontal.

For tip-over alignment, an additional spacing parameter is provided: the minimum distance between branches starting at a node with tip-over alignment. This offset is always orthogonal to the normal branch offset. If the flow direction is to the top or bottom, the tip-over branch offset is vertical.



Spacing parameters

To specify the spacing parameters:

In CSS
Specify in the `GraphLayout` section:

```
parentChildOffset: "30.0";
```

```
siblingOffset: "15.0";
branchOffset: "20.0";
tipOverBranchOffset: "30.0";
```

In Java
In Java™ , use the methods:

```
void setParentChildOffset(float offset);
```

```
void setSiblingOffset(float offset);
```

```
void setBranchOffset(float offset);
```

```
void setTipOverBranchOffset(float offset);
```

For experts: additional spacing parameters

The spacing parameters normally specify the minimal offsets between the node borders. Hence, the layout algorithm places the nodes such that they do not overlap. You can also specify that the layout should ignore the node sizes.

In CSS
Add to the `GraphLayout` section:

```
respectNodeSizes: "false";
```

In Java
In Java, call:

```
layout.setRespectNodeSizes (false);
```

In this case, the spacing parameters are interpreted as the minimum distances between the node centers, and the node sides are not taken into account during the layout. However, if the specified offset parameters are now smaller than the node size, the nodes and links will overlap. This often happens with orthogonal links in particular. It makes sense to use this option only if all nodes have approximately the same size, all links are straight, and the spacing parameters are larger than the largest node.

If the link style is orthogonal, the shape of the links from the parent to its children looks like a fork (see *Different Alignments Mixed in the Same Drawing*). The position of the bend points in this shape can be influenced by the *orthogonal fork percentage*, a value between `0` and `100`. This is a percentage of the parent child offset. If the orthogonal fork percentage is `0`, the link shape forks directly at the parent node. If the percentage is `100`, the link shape forks at the child node. A good choice is between `25` and `75`. This percentage can be set.

In CSS
Add a statement such as:

```
orthForkPercentage: "66.6";
```

In Java
Use the method:

```
void setOrthForkPercentage(float percentage);
```

If the link style is not orthogonal, links may overlap neighboring nodes. This happens only in a very few cases, for instance, when a link starts at a very small node that is neighbored by a very large node. This deficiency can be fixed by increasing the branch offset. However, this influences the layout globally, affecting nodes without that deficiency. To avoid a global change, you can change the *overlap percentage* instead, which is a value between 0 and 100. This value is used by an internal heuristic of the layout algorithm that considers a node to be smaller by this percentage. The default percentage is `30`. This usually results in better usage of the space. However, if very small nodes are neighbored to very large nodes, it is recommended to decrease the overlap percentage or to set it to `0` to switch this heuristic off to avoid links overlapping nodes.

To set the overlap percentage: :

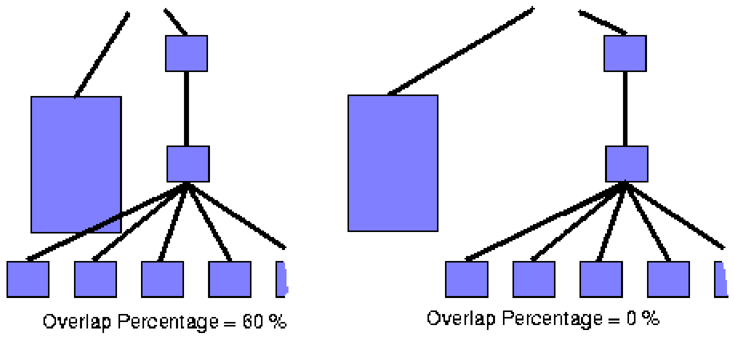
In CSS
Add a statement such as:

```
overlapPercentage: "33.3";
```

In Java
Use the method:

```
void setOverlapPercentage(float percentage);
```

Note: It is recommended that you always set the orthogonal fork percentage to a value larger than the value of the overlap percentage.



Effect of using the overlap percentage

Level layout mode

Describes how the level layout mode organizes nodes and describes the parameters of this mode.

In this section

Overview

The level layout mode partitions the node into levels and arranges the levels horizontally or vertically. The root is placed at level 0, its children at level 1, the children of those children at level 2, and so on. In contrast to the free layout mode, in level layout mode the nodes of the same level are justified with each other even if they are not siblings (that is, they do not have the same parent).

To set the level layout mode:

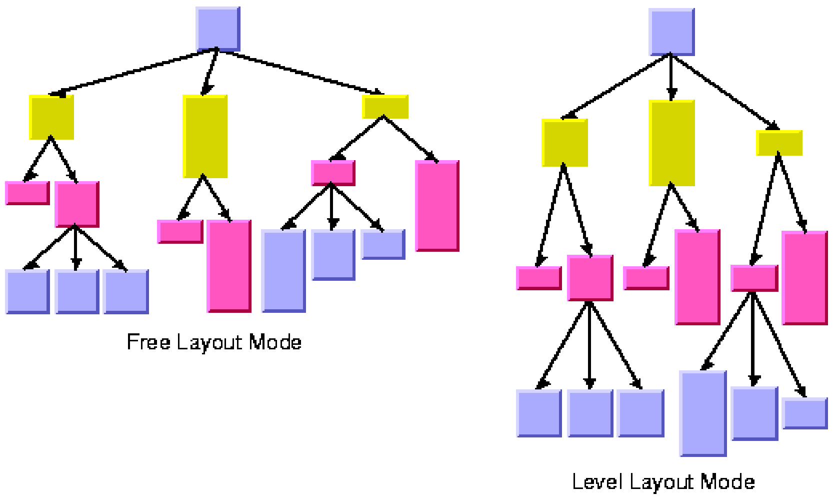
In CSS
Add to the `GraphLayout` section:

```
layoutMode: "LEVEL";
```

In Java
In Java™ , call:

```
layout.setLayoutMode(IlvTreeLayout.LEVEL);
```

The following figure shows the same graph in free layout mode and in level layout mode.



Free layout mode and level layout mode

General parameters

Most layout parameters that work for the free layout mode work as well for the level layout mode. You can set the flow direction, the spacing offsets, the global or individual link style, and the global or individual alignment. See *Free layout mode* for details.

The differences from the free layout mode are:

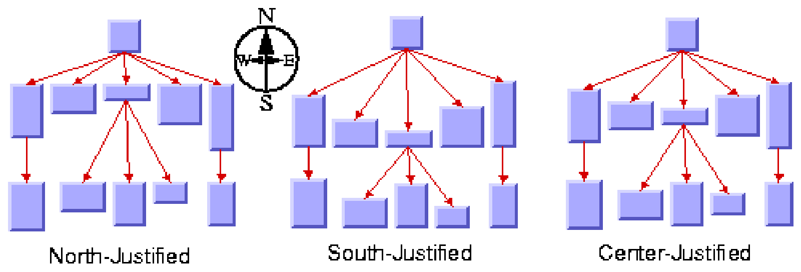
- ♦ The tip-over alignment does not work in level layout mode.
- ♦ The parent-child offset parameter controls the spacing between the levels. In level layout mode, it is the *minimum* distance between parent and its children, while in free layout mode, it is the *exact* distance between parent and its children.
- ♦ The overlap percentage has no effect in level layout mode.

Level alignment

In level layout mode with flow direction to the top or bottom, the nodes are organized in horizontal levels such that the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

In flow direction to the left or right, the nodes are organized in vertical levels approximately at the same x-coordinate. The nodes of the same level can be justified at the left border, at the right border, or at the center.

To distinguish the level alignment independently from the flow direction, the directions north and south are used (see *Using compass directions for positional layout parameters (TL)*). The north border of a node is the border that is closer to the level where its parent is placed, and the south border of a node is the border that is closer to the level where its children are placed. If the flow direction is to the bottom, the level alignment north means that the nodes are justified at the top border, and south means that the nodes are justified at the bottom border. If the flow direction is to the top, north and south are inverted: north means the bottom border and south means the top border. If the flow direction is to the right, then north means the left border and south means the right border.



Level Alignment

To specify the level alignment:

In CSS
Add to the `GraphLayout` section:

```
levelAlignment: "NORTH";
```

In Java
In Java™ , use the method:

```
void setLevelAlignment(int alignment);
```

The valid values for `alignment` are:

- `IlvTreeLayout.CENTER` (the default)
- `IlvTreeLayout.NORTH`
- `IlvTreeLayout.SOUTH`

In CSS, you omit the prefix `IlvTreeLayout` when specifying the value of the level alignment.

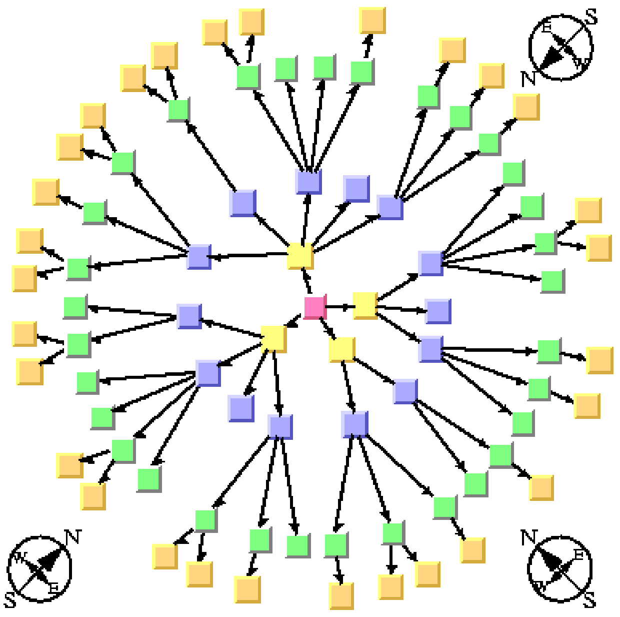
Radial layout mode

Describes how the radial layout mode organizes nodes and describes the parameters of this mode.

In this section

Overview

The radial layout mode partitions the node into levels and arranges the levels in circles around the root node. *Radial layout mode* shows an example of the radial layout mode. The compass icons show the compass directions in this drawing.



Radial layout mode

To set the radial layout mode:

In CSS
Specify in the `GraphLayout` section:

```
layoutMode: "RADIAL";
```

In Java
In Java™ , call:

```
layout.setLayoutMode(IlvTreeLayout.RADIAL);
```

General parameters

Most layout parameters that work for the free and level layout mode work as well for the radial layout mode. You can set the spacing offsets, the level alignment, the global or individual link style, and the global or individual alignment. See *Free layout mode* and *Level layout mode* for details.

The radial layout mode differs from the other layout modes as follows:

- ♦ The tip-over alignment does not work in radial layout mode.
- ♦ The orthogonal link style does not work in radial layout mode.
- ♦ The clipped connector style is always used.
- ♦ The parent-child offset parameter controls the minimal distance between the circular levels. However, it is sometimes necessary to increase the offset between circular levels to obtain enough space on the circle to place all nodes of a level.
- ♦ The level alignment *north* indicates alignment at the inner border of the circular level (that is, towards the root), and the level alignment *south* indicates alignment at the outer border of the circular level (that is, away from the root).
- ♦ The level alignments *north* and *south* sometimes result in overlapping nodes.
- ♦ The overlap percentage has no effect in radial layout mode.

Alternating radial mode

Describes how the alternating radial mode organizes nodes and describes the parameters of this mode.

In this section

Overview

If levels of the graph contain many nodes, it is sometimes necessary to increase the radius of the circular level to provide enough space on the circumference of the circle for all the nodes. This may result in a considerable distance from the previous level. To avoid this, there is an *alternating* radial mode. The alternating radial mode places the nodes of a level alternating between two circles instead of one circle, resulting in better use of the space of the layout.

The alternating radial mode uses two circles only when necessary. For many small and light trees, there will be no difference from the normal radial mode. Only for large graphs with a large number of children will the alternating radial mode have an effect.

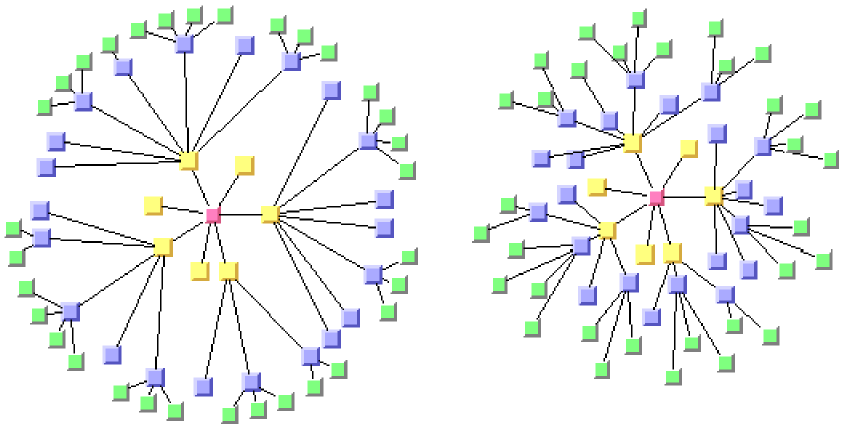
To set the alternating radial layout mode:

In CSS
Specify in the `GraphLayout` section:

```
layoutMode: "ALTERNATING_RADIAL";
```

In Java
In Java™ , call:

```
layout.setLayoutMode(IlvTreeLayout.ALTERNATING_RADIAL);
```


Radial layout mode (right) and alternating radial layout mode (left)

Aspect ratio

If the drawing area is not a square, arranging the levels as circles is not always the best choice. You can specify the aspect ratio of the drawing area to better fit the layout to the drawing area. In this case, the algorithm uses ellipses instead of circles. See *Tree layout in radial layout mode with aspect ratio 1.5* for an example.

To specify the aspect ratio:

In CSS
Add this, for instance, to the `GraphLayout` section:

```
aspectRatio: "0.7";
```

In Java
In Java™, there are several possibilities.

If the drawing area is a view (a subclass of `IlvManagerView`), you can use the method:

```
void setAspectRatio(IlvManagerView view);
```

If the drawing area is given only as a rectangle, use the following:

```
void setAspectRatio(IlvRect rect);
```

If neither a view nor a rectangle is specified, you can calculate the aspect ratio from the width and height of the drawing area as `aspectRatio = width/height` and use the method:

```
void setAspectRatio(float aspectRatio);
```

Spacing parameters

The spacing parameters of the radial layout modes are controlled by the same CSS statements and methods as used for the free and level layout modes:

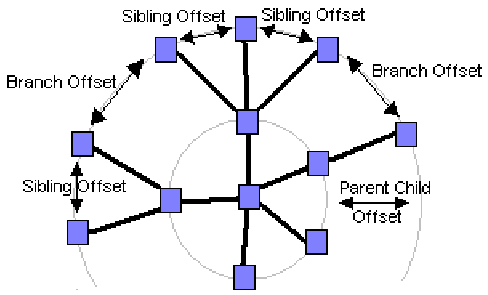
```
void setParentChildOffset(float offset);
```

```
void setSiblingOffset(float offset);
```

```
void setBranchOffset(float offset);
```

Note that the sibling and branch offsets are minimum distances tangential to the circles or ellipses, while the parent-child offset is a minimum distance radial to the circles or ellipses.

The following figure shows the spacing parameters in radial layout mode.



Spacing parameters in radial layout mode

Tips and tricks

Adding an invisible root to the layout

If the graph contains several trees that are disconnected from each other, the layout places them individually next to each other. Each connected component has its own radial structure with circular layers. However, sometimes it is appropriate to fit all connected components into a single circular layer structure. Conceptually, this is done by adding an invisible root at the center and connecting all disconnected trees to this root. *Layout of connected components without and with an invisible root* shows the effect of using an invisible root. This works only if the generic mechanism to lay out connected components is switched off.

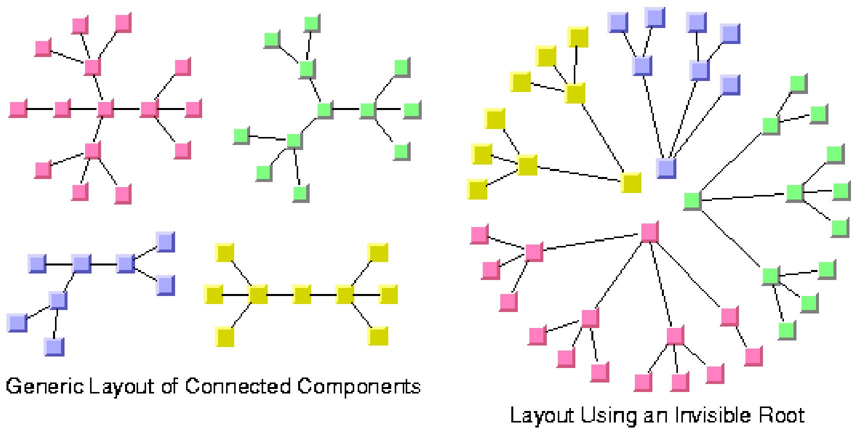
To add an invisible root to the layout:

In CSS
Specify:

```
layoutOfConnectedComponentsEnabled: "false";
invisibleRootUsed: "true";
```

In
Call:

```
layout.setLayoutOfConnectedComponentsEnabled(false);
layout.setInvisibleRootUsed(true);
```


Generic Layout of Connected Components
Layout Using an Invisible Root

Layout of connected components without and with an invisible root

Even spacing for the first circle

The radial mode is designed to optimize the space such that the circles have a small radius and the overall space for the entire layout is small. To achieve this, the layout algorithm may create larger gaps on the inner circles for better space usage of the outer circles. This may produce unevenly spaced circles, most notably for the first circle where all nodes have the same parent node.

To avoid this effect, you can force the nodes to be evenly spaced on the entire first circle. Depending on the structure of the graph, this may cause the overall layout to waste more space on the other circles but may produce a more pleasing graph.

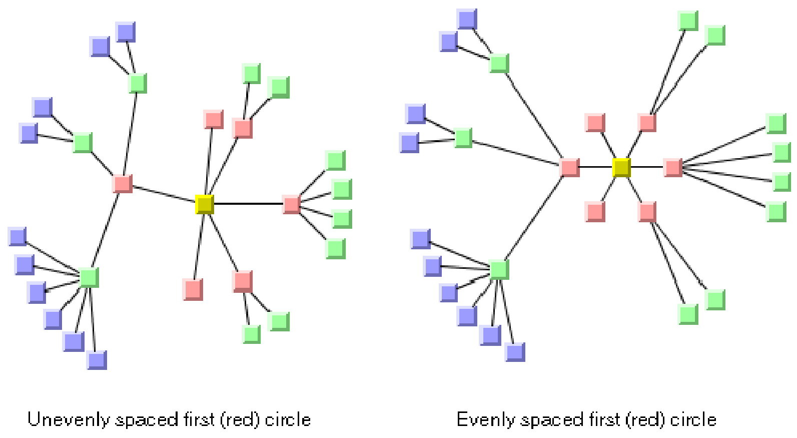
To enable even spacing:

In CSS
Specify:

```
firstCircleEvenlySpacing: "true";
```

In Java
In Java™ , call:

```
layout.setFirstCircleEvenlySpacing (true);
```


Unevenly spaced first (red) circle
Evenly spaced first (red) circle

Evenly and Unevenly Spaced First Circle

For experts: forcing all levels to alternating

When the layout mode `ALTERNATING_RADIAL` is used, the layout checks whether the alternating node arrangement of a level saves space. If that does not save space, it uses the normal radial arrangement. Hence, for many sparse graphs, radial and alternating radial mode yield the same result because the alternating arrangement does not save space for any level. It is possible to disable the space check, that is, to perform an alternating arrangement for all levels even if this results in waste of space.

In CSS
Add to the `GraphLayout` section:

```
allLevelsAlternating: "true";
```

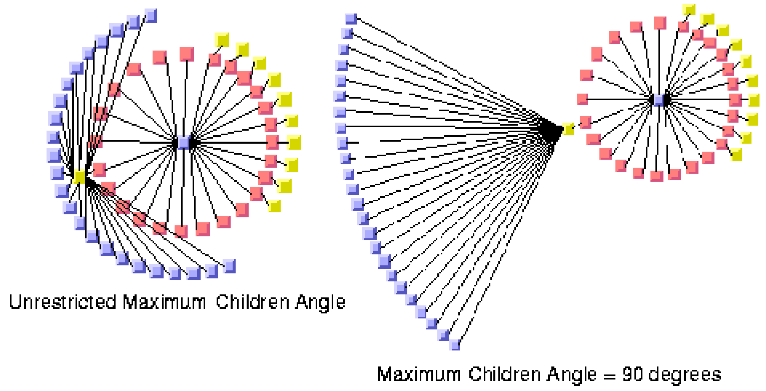
In Java
Call:

```
layout.setAllLevelsAlternating(true);
```

For experts: setting a maximum children angle

If a node has a lot of children, they may extend over a major portion of the circle and, therefore, are placed nearly 360 degrees around the node. This can result in links overlapping some nodes. The deficiency can be fixed by increasing the offset between parent and children. However, this affects the layout globally which means that nodes without the deficiency are also affected.To avoid a global change such as this, you can limit the maximum angle between the two rays from the parent (if it is not the root) to its two outermost children. This increases the offset between parent and children only where necessary.

In *Maximum Children Angle*, you can see in the layout on the left that many of the links overlap other nodes. In the layout on the right, you can see how this problem was solved by setting a maximum children angle between two rays from a parent to the two outermost children.


Unrestricted Maximum Children Angle
Maximum Children Angle = 90 degrees

Maximum Children Angle

To set an angle in degrees:

In CSS
Specify:

```
maxChildrenAngle: "90";
```

In Java
Use the method:

```
void setMaxChildrenAngle(int angle);
```

Recommended values are between `30` and `180`. Setting the value to `0` means the angle is unrestricted. The calculation of the angle is not very precise above 180 degrees or if the aspect ratio is not 1.0.

Tip-over layout modes

Drawing in radial layout mode and free layout mode can be adjusted according to the aspect ratio of the drawing area. To balance the height and depth of the drawing, free layout mode can also use tip-over alignment.

Tip-over alignment can be specified explicitly for individual nodes; the Tree Layout algorithm also has layout modes that automatically use tip-over alignment when needed. These are the tip-over layout modes.

The tip-over layout modes work as follows: Several trial layouts are performed in free layout mode. For each trial, tip-over alignment is set for certain individual nodes, while the specified alignment of all other nodes is preserved. The algorithm picks the trial layout that best fits the specified aspect ratio of the drawing area.

The aspect ratio can be set in CSS and in Java, by one of the methods (see *Aspect ratio* in the Radial Layout Mode):

```
void setAspectRatio(IlvRect rect);
```

```
void setAspectRatio(float aspectRatio);
```

The tip-over modes are slightly more time-consuming than the other layout modes. For very large trees, it is recommended that you set the allowed layout time to a high value (for instance, 60 seconds) when using the tip-over modes.

To set this mode:

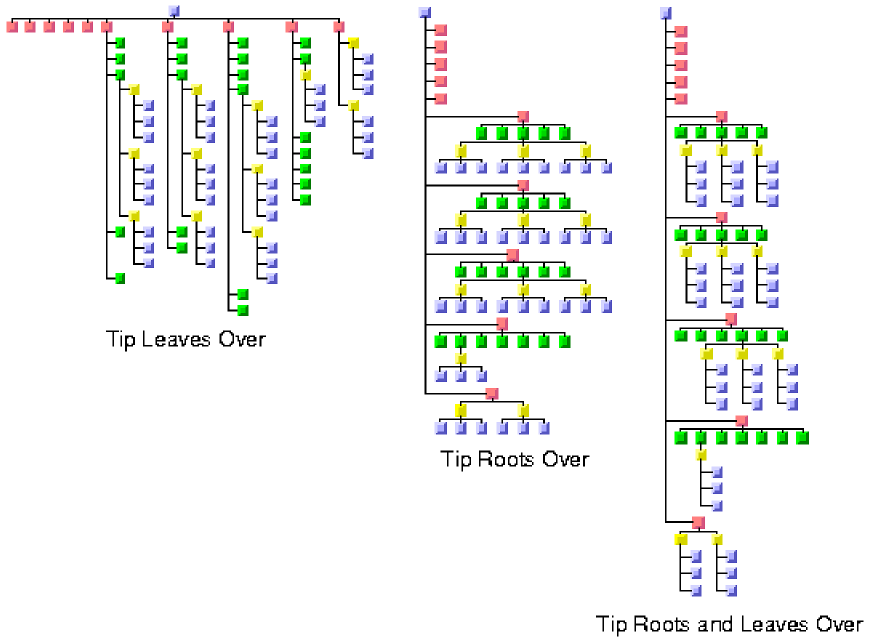
In CSS
Add to the `GraphLayout` section:

```
allowedTime: "60000";
```

In Java
Call:

```
layout.setAllowedTime(60000);
```

By using this call, you avoid running short of time for sufficient iterations of the layout algorithm. Because it would be too time-consuming to check all possibilities of tip-over alignment use, there are heuristics that check only certain trials according to the following different strategies, illustrated in the following figure.



Tip-over strategies

- *Tip leaves over*
- *Tip roots over*
- *Tip roots and leaves over*
- *Tip over fast*

Tip leaves over

To use this tip-over strategy, set the layout mode as follows:

In CSS
Add in the `GraphLayout` section:

```
layoutMode: "TIP_LEAVES_OVER";
```

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_LEAVES_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the leaves, then the leaves and their parents, then additionally the parents of these parents, and so on. As a result, the nodes closest to the root use normal alignment and the nodes closest to the leaves use tip-over alignment.

Tip roots over

To use this tip-over strategy, set the layout mode as follows:

In CSS
Add in the `GraphLayout` section:

```
layoutMode: "TIP_ROOTS_OVER";
```

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_ROOTS_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node, then the root and its children, then additionally the children of these children, and so on. As a result, the nodes closer to the leaves use normal alignment and the nodes closer to the root use tip-over alignment.

Tip roots and leaves over

To use this tip-over strategy, set the layout mode as follows:

In CSS
Add in the `GraphLayout` section:

```
layoutMode: "TIP_ROOTS_AND_LEAVES_OVER";
```

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_ROOTS_AND_LEAVES_OVER);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node and the leaves simultaneously; then the root and its children, and the leaves and its parent; then additionally the children of these children and the parents of these parents, and so on. As result, the nodes in the middle of the tree use normal alignment and the nodes closest to the root or leaves use the tip-over alignment.

This is the slowest strategy because it includes all trials of the strategy *"tip leaves over"* as well as all tries of the strategy *"tip roots over."*

Tip over fast

The fast tip-over provides a compromise among all other strategies. The heuristic tries a small selection of the other strategies, not all possibilities. Therefore, it is the fastest strategy for large graphs.

To use this strategy, set the layout mode as follows:

In CSS
Add in the `GraphLayout` section:

```
layoutMode: "TIP_OVER";
```

In Java

```
layout.setLayoutMode(IlvTreeLayout.TIP_OVER);
```

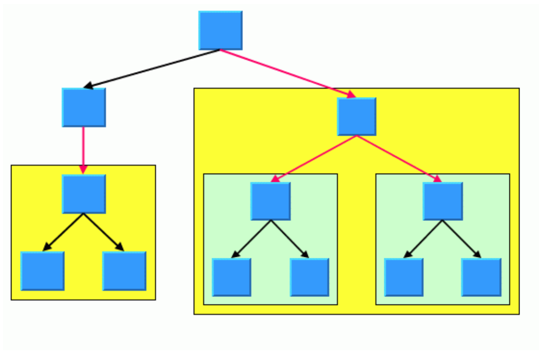
It is possible that all four strategies yield the same result because the strategies are not disjoint; that is, certain trials are performed in all four strategies. In addition, the tip-over modes do not necessarily produce the optimal layout that gives the best possible fit to the aspect ratio. The reason is that some unusual configurations of tip-over alignment are never tried because doing so would cause the running time to be too high.

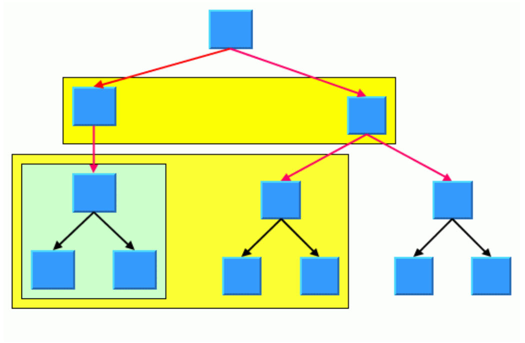
Recursive mode

JViews Diagrammer supports nested graphs, that is, it can render graphs containing nodes that are graphs. A graph that is a node in another graph is called a subgraph. Links that connect nodes of different subgraphs are called intergraph links. In *Leaf recursive tree*, all red links are intergraph links and all black links are normal links. This is explained in detail in *Nested layouts*.

The tree layout can treat a nested graph in specific situation at once and route the intergraph links as well as the normal links that belong to the tree. It can handle a leaf-recursive tree at once. A leaf recursive tree has the following properties:

- it is a tree
- only leaf nodes of the tree can contain nested graphs
- the root node of the tree nested in a leaf node is connected by a link to the parent node of the leaf


Leaf recursive tree


Non leaf recursive tree

This graph is not a leaf recursive tree: the subgraphs are not nested in the leafs of the tree. The graph cannot be handled by the tree layout in recursive mode, but it can be handled by the hierarchical layout in recursive mode. If the graph is a leaf recursive tree and the layout mode is not the radial layout mode, the tree layout can handle the nested graph at once.

To enable the recursive mode:

In CSS
Specify in the `GraphLayout` section:

```
recursiveLeafLayoutMode: "true";
```

In Java
In Java™ , use the method:

```
void setRecursiveLeafLayoutMode(boolean enable);
```

and call `performLayout` with the third parameter set to `true` as follows:

```
layout.performLayout(force, redraw, true);
```

The recursive leaf layout mode requires that all subtrees are laid out in the same style (for instance, they must all use the same flow direction). This is automatically the case when calling `layout.performLayout(force, redraw, true)`, and it is also the case when using CSS without specifying individual graph layouts per subgraph. If different layout styles are needed per subgraph, you must specify an individual layout per subgraph as described in *Individual layout styles per subgraph* and in *Advanced recursion: mixing different layouts in a nested graph* and in this case the recursive leaf layout mode cannot be used.

Setting layout parameters in recursive leaf layout mode by Java code

When you use CSS, the layout parameters are specified in CSS as usual, and the SDM engine handles the internal details automatically. However, if you want to specify layout parameters by code, note that in recursive leaf layout mode the tree layout is attached to the top level graph. Global layout parameters must be set on this layout instance. Layout parameters per node or per link must be set in the following way:

```
// link is directly contained in subgraph
   IlvTreeLayout sublayout =
      (IlvTreeLayout)topLevelLayout.getRecursiveLayout().getLayout(subgraph);

   sublayout.setLinkStyle(link, IlvTreeLayout.ORTHOGONAL_STYLE);
```

This means that layout parameters per nodes and per links cannot be set on the top level layout, but on a sublayout retrieved via the `IlvRecursiveLayout` from the top level layout.

For experts: additional tips for the TL

The tips and trick are relevant when accessing the layout instance directly in Java.

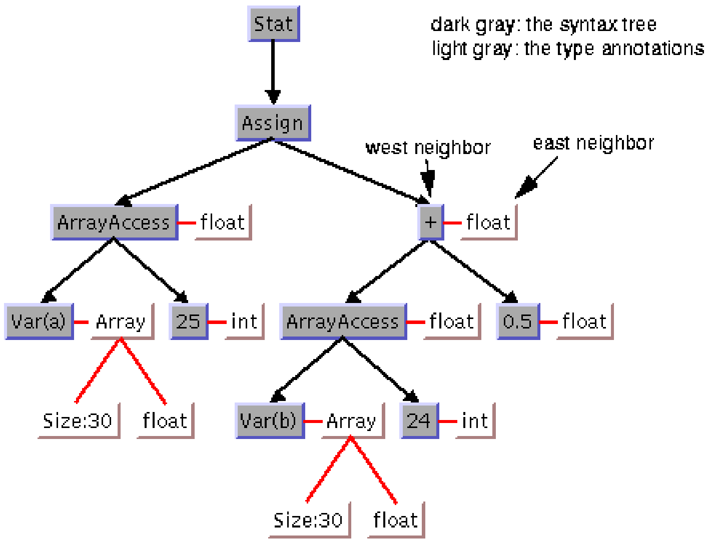
Most of the tips cannot be used when specifying CSS alone.

Specifying east-west neighbors

You can specify that two unrelated nodes must be direct neighbors in a direction perpendicular to the flow direction. In the level and radial layout modes, the nodes are placed in the same level next to each other. In the free layout and tip-over modes, the nodes are placed aligned at the north border. Such nodes are called *east-west neighbors* because one node is placed as the direct neighbor on the east side of the other node. The other node becomes the direct neighbor on the west side of the first node. (See also *Using compass directions for positional layout parameters (TL)*).

Technically, the nodes are treated as parent and child, even if there may be no link between them. Therefore, one of the two nodes can have a real parent, but the other node should not because its virtual parent is its *east-west neighbor*.

The east-west neighbor feature can be used, for example, for annotating nodes in a typed syntax tree occurring in compiler construction. *Annotated Syntax Tree of Statement a[25] = b[24] + 0.5;* shows an example of such a tree.



Annotated Syntax Tree of Statement a[25] = b[24] + 0.5;

To specify that two nodes are east-west neighbors, use the method:

```
void setEastWestNeighboring(Object eastNode, Object westNode);
```

You can also use the following method, which is identical except for the reversed parameter order:

```
void setWestEastNeighboring(Object westNode, Object eastNode);
```

If the flow direction is to the bottom, the latter method may be easier to remember because, in this case, west is to the left of east in the layout, which is similar to the text flow of the parameters.

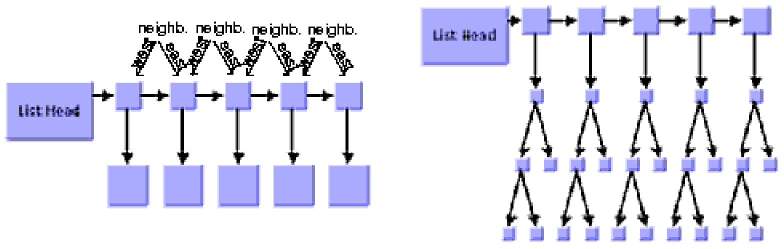
To obtain the node that is the east or west neighbor of a node, use the calls:

```
Object getEastNeighbor(Object node);
```

```
Object getWestNeighbor(Object node);
```

Note that each node can have at most one east neighbor and one west neighbor because they are *direct* neighbors. If more than one direct neighbor is specified, it is partially ignored. Cyclic specifications can cause conflict as well. For instance, if node B is the east neighbor of node A and node C is the east neighbor of B, then node A cannot be the east neighbor of C. (Strictly speaking, such cycles could be technically possible in some situations in the radial layout mode, but nonetheless they are not allowed in any layout mode.)

If B is the east neighbor of A, then A is automatically the west neighbor of B. On the other hand, the east neighbor of A can itself have another east neighbor. This allows the creation of chains of east-west neighbors, which is a common way to visualize lists of trees. Two examples are shown in *Chains of east-west neighbors to visualize lists of trees*.



Chains of east-west neighbors to visualize lists of trees

Retrieving link categories

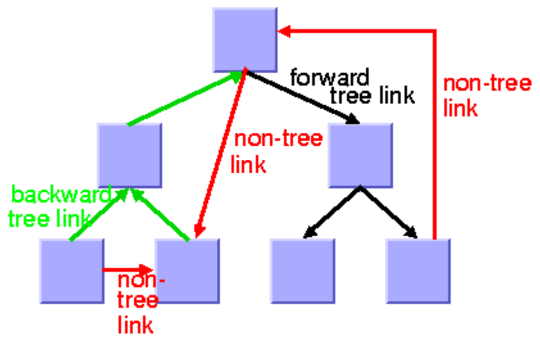
The Tree Layout algorithm works on a spanning tree, as mentioned in a *The TL algorithm*. If the graph to be laid out is not a pure tree, the algorithm ignores some links. To treat such links in a special way, you can obtain a list of nontree links.

Because there are parents and children in the spanning tree, the following link categories must be distinguished:

♦ A forward tree link is a link from a parent to its child.

- A backward tree link is a link from a child to its parent. If the link is drawn as a directed arrow, the arrow will point in the opposite direction to the flow direction.

- A nontree link is a link between two unrelated nodes; neither one is a child of the other.



Link categories

The layout algorithm uses these link categories internally but does not store them permanently to save time and ensure memory efficiency. If you want to treat some link categories in a special way (for example, to call the Link Layout on the nontree links), you must specify *before the layout* that you want to access the link categories *after the layout*. To do this, use the method `setCategorizingLinks(boolean)` in the following way:

```
layout.setCategorizingLinks(true);
// now perform a layout
layout.performLayout();
// now you can access the link categories
```

After the layout, the link categories can be obtained by the methods:

`getCalcForwardTreeLinks()`

`getCalcBackwardTreeLinks()`

`getCalcNonTreeLinks()`

The link category data gets filled each time the layout is called, unless you set the method `setCategorizingLinks(boolean)` back to `false`.

Sequences of layouts with incremental changes

You can work with trees that have become out-of-date, for example, those that need to be extended with more children. If you perform a layout after an extension, you probably want to identify the parts that had already been laid out in the original graph. The Tree Layout algorithm supports these incremental changes in incremental mode because it takes the previous positions of the nodes into account. It preserves the relative order of the children in the subsequent layout.

In nonincremental mode, the Tree Layout algorithm calculates the order of the children from the node order given by the attached graph model (or grapher). In this case, the layout is independent from the positions of the nodes before layout. It does not preserve the relative order of the children in subsequent layouts.

The incremental mode is enabled by default.

To disable the incremental mode:

In CSS
Add to the `GraphLayout` section:

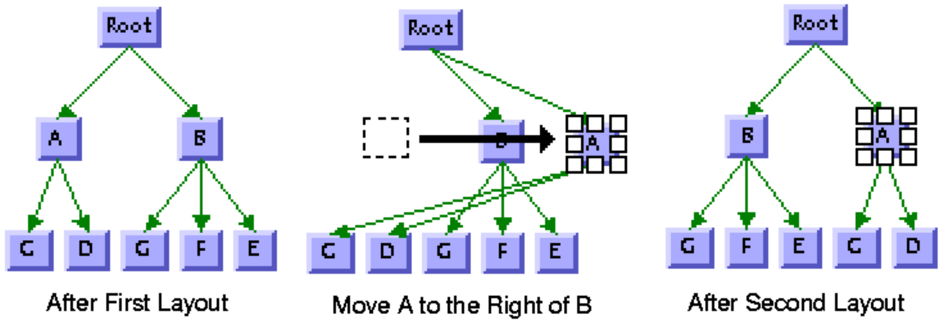
```
incrementalMode: "false";
```

In Java
Call:

```
layout.setIncrementalMode (false);
```

Interactive editing

The fact that the relative order of the layout is preserved is particularly useful during interactive editing. It allows you to correct the layout easily. For instance, if the first layout places a node A left to its sibling node B but you need to reverse the order, you can simply move node A to the right of node B and start a new layout to clean up the drawing. In the second layout, A remains to the right of B, and the subtree of A will “follow” node A.



Interactive Editing to Achieve a Specific Order of Children

Specifying the order of children

Some applications require a specific relative order of the children in the tree. This means that, for instance, when the flow direction is to the bottom, which child must be placed to the left of another child. Even if the graph has never been laid out, you can use the coordinates to specify a certain order of the children at a node. You can use the following:

♦ First, make sure that the incremental mode is enabled.

♦ In free and level layout modes with flow direction to the bottom or top, determine the maximal width `w` of all nodes. Simply move the child that should be in the leftmost position to the coordinate `(0, 0)`, and the child that should get the *i*th relative position (in order from left to right) to the coordinate `((W+1)*i, 0)`.

- If the flow direction is to the left or to the right, determine the maximal height `H` of all nodes. Move the child that should be in the topmost position to the coordinate `(0, 0)` and the child that should get the *i*th relative position (in the order from top to bottom) to coordinate `(0, (H+1)*i)`.

- In the radial layout modes, determine the maximal diagonal `D = W2 + H` of all nodes. If the position of the parent is `(x, y)` before the layout, move the child that should be the first in the circular order to the coordinate `(x, y+D)` and the child that should get the *i*th relative position in the circular order to coordinate `(x+D*i, y+D)`.

  If you want to specify a relative order for all nodes in radial layout mode, you must do this for the parents before you do it for the children. In this case, moving the children can be performed easily during a depth-first traversal from the root to the leaves.

The layout that is performed after moving the children arranges the children with the relative order.

Hierarchical Layout (HL)

Describes the *Hierarchical Layout* algorithm (class `IlvHierarchicalLayout` from the package `ilog.views.graphlayout.hierarchical`).

In this section

Specifying constraints in CSS for HL
Describes how to specify constraints in CSS in a style sheet.

Adding and removing constraints in Java for HL
Describes how to specify constraints in Java™ .

Level range constraints (HL)
Explains how modes are partitioned into levels and how to set constraints at a specific level.

Level index parameter (HL)
Describes how to force a node to a particular level with the level index parameter constraint.

Same level constraints (HL)
Describes how to force several nodes to be at the same level.

Group spread constraints (HL)
Describes how to force a group of nodes to the same level.

Relative level constraints (HL)
Describes how to force a node into a higher level than another node.

Position index parameter (HL)
Describes how to use the position index parameter.

Relative position constraints (HL)
Describes how to use relative position constraints.

Side-by-side constraints (HL)
Describes how to use side-by-side constraints.

Extremity constraints (HL)
Describes how to use extremity constraints.

Swim lane constraints (HL)
Describes how to use swim lane constraints.

Summary of constraints file as opposed to constraints in Java (HL)
Gives a summary of how to specify constraints.

Constraint priorities (HL)
Discusses constraint priorities.

For experts: constraint validation (HL)
Discusses how validation is done during layout and how to force it if necessary.

For experts: specifying constraints in CSS directly (HL)
Describes when a CSS file can be used.

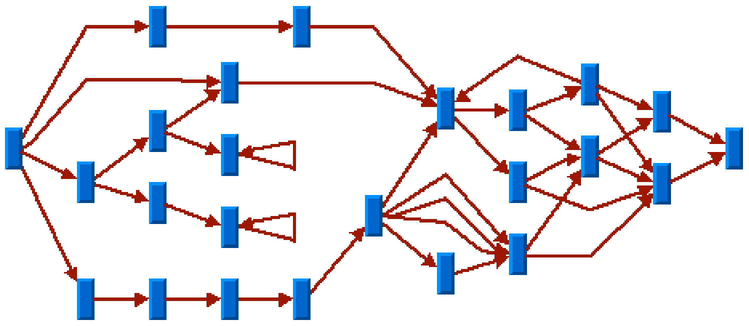
For experts: more indices (HL)
Describes how to specify level and position indices or retrieve calculated indices.

Recursive layout
Explains the recursive mode supported by the hierarchical layout.

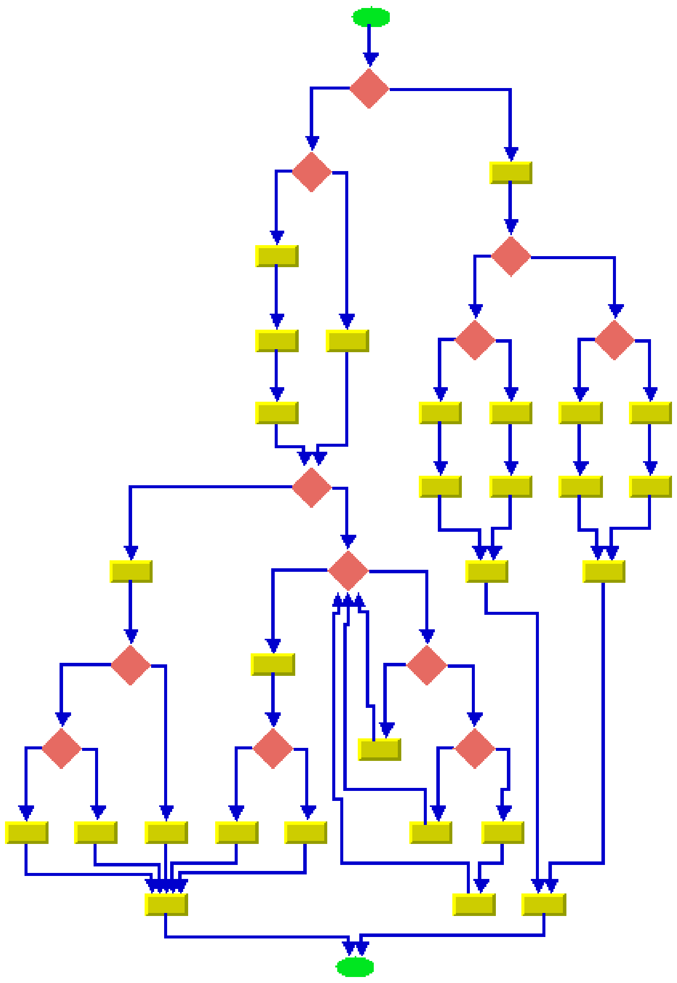
General information on the HL

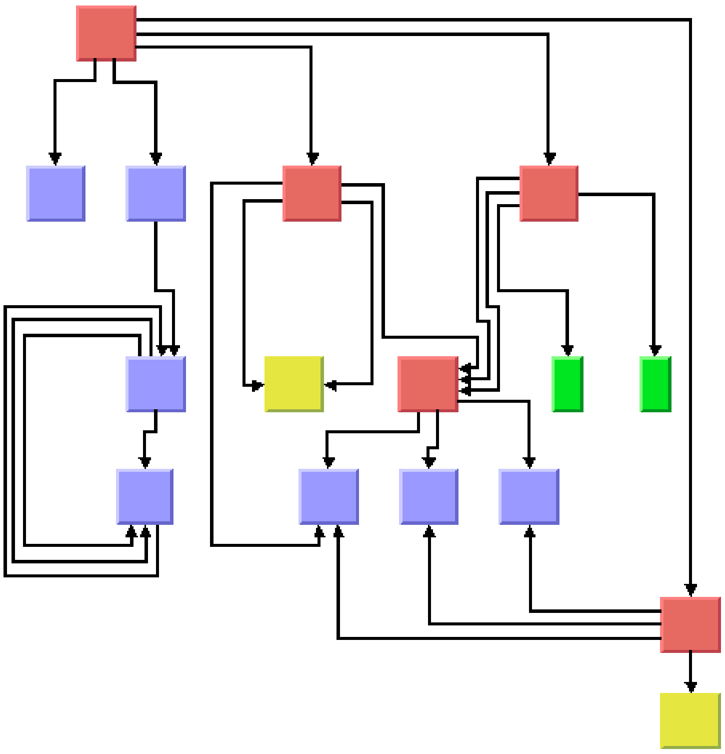
HL samples

Here are some sample drawings produced with the Hierarchical Layout:

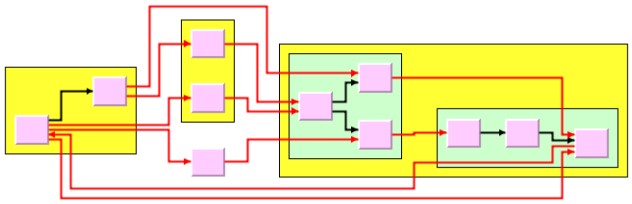


Sample layout with self-loops, multiple links, and cycles


Flowchart with orthogonal link style



Sample layout with ports and orthogonal link style



Sample layout of nested graph in recursive layout mode

What types of graphs suit the HL?

Any type of graph:

♦ Preferably graphs with directed links. A directed link has a direction from source node to target node and is usually drawn with an arrow. The algorithm takes the link directions into account..

♦ connected graphs and disconnected graphs

- planar graphs and nonplanar graphs
- nested graphs with intergraph links

Application domains for the HL

Application domains for the Hierarchical Layout include:

- Electrical engineering (logic diagrams, circuit block diagrams)
- Industrial engineering (industrial process diagrams, schematic design diagrams)
- Business processing (workflow diagrams, process flow diagrams, PERT charts)
- Software management/software (re-)engineering (UML diagrams, flowcharts, data inspector diagrams, call graphs)
- Database and knowledge engineering (database query graphs)
- CASE tools (designs diagrams)

Features and limitations of the HL

Features

- ♦ Organizes nodes without overlaps in horizontal or vertical levels.
- ♦ Arranges the graph such that the majority of links are short and flow uniformly in the same direction (from left to right, from top to bottom, and so on).
- ♦ Reduces the number of link crossings. Most of the time, produces drawings with no crossings or only a small number of crossings.
- ♦ Often produces balanced drawings that emphasize the symmetries in the graph.
- ♦ Supports self-links (that is, links with the same origin and destination node), multiple links between the same pair of nodes, and cycles.
- ♦ Efficient, scalable algorithm. Produces a nice layout for most sparse and medium-dense graphs relatively quickly, even if the number of nodes is very large.
- ♦ Provides several alignment and offset options.
- ♦ Supports port specifications where links attach the nodes. Allows you to specify which side of a node (top, bottom, left, right) a link can be connected to or to specify which relative port position should be used for the connection.
- ♦ Supports layout constraints. Allows you to specify relative positional constraints, for instance, that a node is above another node or left of another node.
- ♦ Incremental and nonincremental mode. In incremental mode, the previous position of nodes are taken into account. Positions the nodes without changing the relative order of the nodes so that the layout is stable on incremental changes of the graph.
- ♦ Can handle flat and nested graphs. In recursive layout mode, it routes the intergraph links of nested graphs and places the labels of nodes and links in subgraphs.
- ♦ The computation time depends on the number of nodes, the number of levels, and the number of links that cross several levels. Most of the time, the links are placed between adjacent levels, which keeps the computation time small.

Limitations

- ♦ The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to solve this problem quickly for any graph size. Therefore, the algorithm uses a very fast heuristic that obtains a good layout, but not always with the theoretical minimum number of link crossings.
- ♦ The algorithm tries to place the nodes such that all links point uniformly in the same direction. It is impossible to place cycles of links in this way. For this reason, it sometimes produces a graph where a small number of links are reversed to point into the opposite direction. The algorithm tries to minimize the number of reversed links (which, again, is an NP-complete problem). Therefore, the algorithm uses a very fast heuristic resulting in a good layout, but not always with the theoretical minimum number of reversed links.

- The computation time required to obtain an appropriate drawing depends most significantly on the number of bends in the links. Since the algorithm places one bend whenever a link crosses a level, the number of bends can grow relatively quickly if the layout requires many long links that span several levels. Therefore, the layout process may become very time-consuming for dense graphs (the number of links is relatively high compared to the number of nodes) or for graphs that require a large number of node levels.

The HL algorithm

A brief description of the HL algorithm

This algorithm works in four steps:

Step 1: Leveling

The nodes are partitioned into groups. Each group of nodes forms a level. The objective is to group the nodes in such a way that the links always point from a level with smaller index to a level with larger index.

Step 2: Crossing reduction

The nodes are sorted within each level. The algorithm tries to keep the number of link crossings small when, for each level, the nodes are placed in this order on a line (see *Level and position indices*). This ordering results in the relative position index of each node within its level.

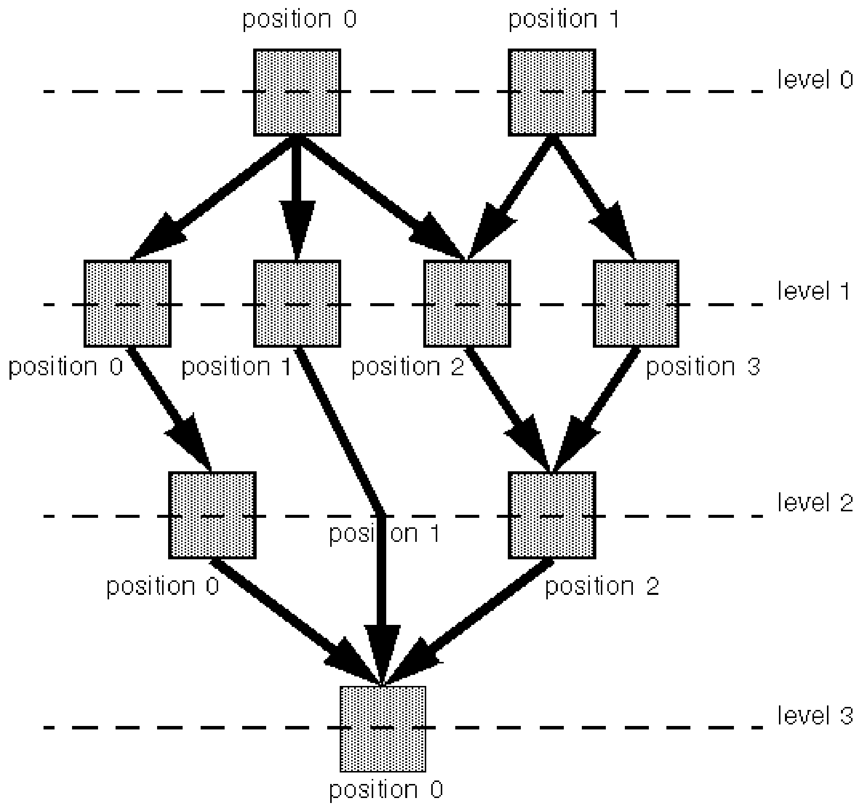
Step 3: Node positioning

From the level indices and position indices, balanced coordinates for the nodes are calculated. For instance, for a layout where the link flow is from top to bottom, the nodes are placed along horizontal lines such that all nodes belonging to the same level have (approximately) the same y-coordinate. The nodes of a level with a smaller index have a smaller y-coordinate than the nodes of a level with a higher index. Within a level, the nodes with a smaller position index have a smaller x-coordinate than the nodes with a higher position index.

Step 4: Link routing

The shapes of the links are calculated such that the links bypass the nodes at the level lines. In many cases, this requires that a bend point be created whenever a link needs to cross a level line. In a top-to-bottom layout, these bend points have the same y-coordinate as the level line they cross. (Note that these bend points also obtain a position index).

Level and position indices shows how the Hierarchical Layout algorithm uses the level and position indices to draw the graph.



Level and position indices

You can set parameters for the steps of the layout algorithm in several ways. For instance, you can specify the level index that the algorithm should choose for a node in Step 1 or the relative node position within the level in Step 2. You can also specify the justification of the nodes within a level and the style of the link shapes.

Example of HL

In CSS

The following example is a specification that uses the Hierarchical Layout algorithm. Since the Hierarchical Layout places nodes and links, it is usually not necessary to specify an additional link layout in CSS.

The specification can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see Graph Layout in IBM® ILOG® JViews Diagrammer).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}
```

```
GraphLayout {
    enabled             : "true";
    graphLayout         : "Hierarchical";
    flowDirection       : "Bottom";
    globalLinkStyle     : "POLYLINE_STYLE";
    connectorStyle      : "CLIPPED_PINS";
    horizontalNodeOffset: "20";
    verticalNodeOffset  : "20";
}
```

In some situations, a separate link layout renderer may be required:

♦ if the links must be rerouted during interactions, for example, when nodes are moved manually;

♦ if the graph contains nested subgraphs, because the Hierarchical Layout does not handle intergraph links.

In these cases, it is recommended to use the “hierarchical link layout”. The hierarchical link layout is not a separate layout algorithm. It is merely the feature of the link layout renderer to reuse the Hierarchical Layout as a link layout. The following CSS sample specifies the Hierarchical Layout algorithm as node and link layout renderer:

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "true";
}

GraphLayout {
    enabled             : "true";
    graphLayout         : "Hierarchical";
    flowDirection       : "Bottom";
    globalLinkStyle     : "ORTHOGONAL_STYLE";
    connectorStyle      : "EVENLY_SPACED_PINS";
}

LinkLayout {
    hierarchical: "true";
}
```

It is also possible to use the standard link layout (class `IlvLinkLayout`) in the link layout renderer (the `hierarchical` parameter is set to `"false"`). However in this case, the link layout determines the shapes of the links. The explanations pertaining to the shape of the links in Hierarchical Layout are valid only if the link layout is disabled.

In Java
In Java™ , below is a code sample that uses the `IlvHierarchicalLayout` class. This code sample shows how to perform a Hierarchical Layout on a grapher directly without using a diagram component or any style sheet:

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.hierarchical.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /* Fill in the grapher with nodes and links here */

IlvHierarchicalLayout layout = new IlvHierarchicalLayout();
layout.attach(grapher);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic features and parameters of the HL

Overview of generic features

The `IlvHierarchicalLayout` class supports the following generic features defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- *Allowed time (HL)*
- *Layout of connected components (HL)*
- *Link clipping (HL)*
- *Link connection box (HL)*
- *Spline routing (HL)*
- *Percentage of completion calculation (HL)*
- *Preserve fixed links (HL)*
- *Preserve fixed nodes (HL)*
- *Stop immediately (HL)*
- *Save parameters to named properties (HL)*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed time (HL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components (HL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*.) When using this mechanism, each component is laid out in its own individual level structure. Nodes of the first level of one component may be placed at a different position than nodes of the first level of another component.

The generic mechanism to layout connected components is, however, switched off by default. In this case, the layout algorithm can still handle disconnected graphs. It merges all components into a global level structure.

Link clipping (HL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface (HL)* for details of the link clipping mechanism.

Link connection box (HL)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link connection box interface (HL)*.

Spline routing (HL)

The layout algorithm supports the generic spline routing mechanism (see *Spline routing*). If the style of a link is polyline or orthogonal and the link is a spline, it is routed by the generic spline routing mechanism when it is enabled.

Percentage of completion calculation (HL)

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage of completion calculation* and *Graph layout event listeners*.)

Preserve fixed links (HL)

The layout algorithm does not reshape the links that are specified as fixed. In fact, fixed links are completely ignored. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve fixed links* and *Link style*.)

Preserve fixed nodes (HL)

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve fixed nodes*.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in unwanted overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Save parameters to named properties (HL)

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*).

Stop immediately (HL)

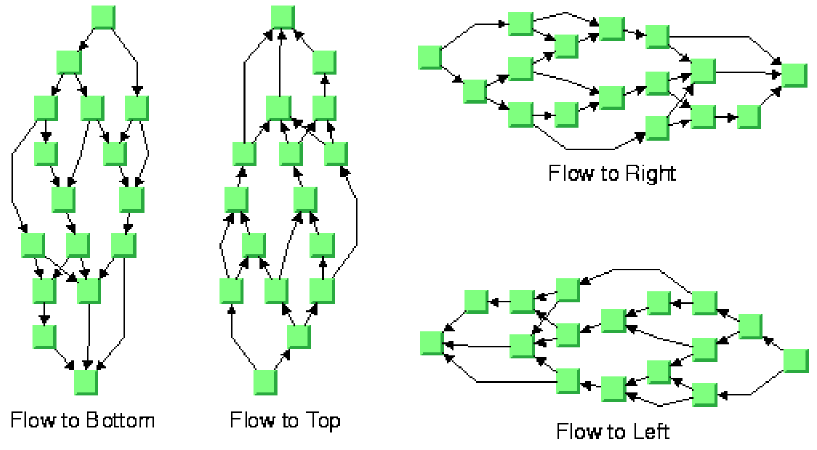
The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call and the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the HL

Flow direction (HL)

The flow direction parameter specifies the direction in which the majority of the links should point. If the flow direction is to the top or to the bottom, the node levels are oriented horizontally and the links mostly vertically. If the flow direction is to the left or to the right, the node levels are oriented vertically and the links mostly horizontally.

If the flow direction is to the bottom, the nodes of the level with index 0 are placed at the top border of the drawing. The nodes with level index 0 are usually the root nodes of the drawing (that is, the nodes without incoming links). If the flow direction is to the top, the nodes with level index 0 are placed at the bottom border of the drawing. If the flow direction is to the right, the nodes are placed at the left border of the drawing.



Flow directions

To specify the flow direction towards the bottom:

In CSS
Add to the `GraphLayout` section:

```
flowDirection: "Bottom";
```

In Java
In Java, use the method:

```
void setFlowDirection(int direction)
```

The valid values for the flow direction are:

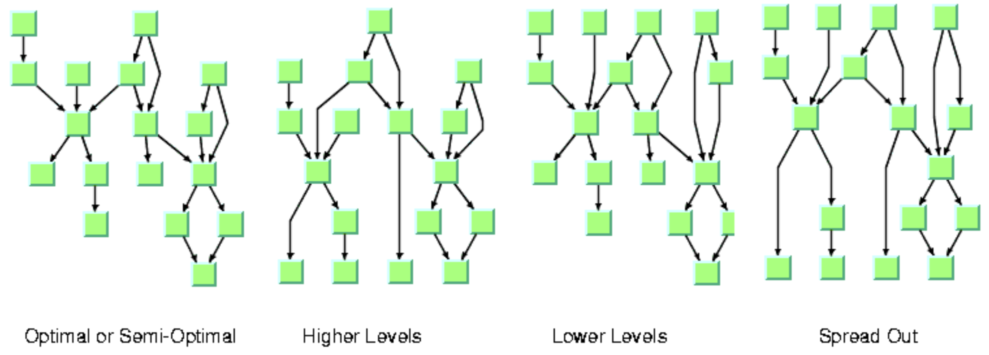
♦ `IlvDirection.Right` (the default)

- `IlvDirection.Left`
- `IlvDirection.Bottom`
- `IlvDirection.Top`

In CSS, you omit the prefix `IlvDirection` when specifying the value of the flow direction.

Leveling strategy (HL)

The layout algorithm partitions the nodes into levels (see *A brief description of the HL algorithm*). The leveling strategy specifies how the levels are calculated. Besides the leveling strategy, layout constraints (see *Layout constraints for HL*), level indices (see *For experts: more indices (HL)*) as well as the incremental mode (see *Incremental mode with HL*) also affect the way the levels are calculated. If the incremental mode is disabled, the leveling strategy determines the levels of all nodes that are not subject to layout constraints and level index specifications.



Leveling strategies

To specify the leveling strategy:

In CSS
Add to the `GraphLayout` section:

```
levelingStrategy: "OPTIMAL";
```

In Java
In Java™ , use the method:

```
void setLevelingStrategy(int strategy)
```

The valid values for the leveling strategy are:

- `IlvHierarchicalLayout.SEMI_OPTIMAL` (the default)

  This produces often the same result as the optimal strategy, but it is quicker. The layout algorithm uses a heuristic to minimize the sum of level distances for all edges. It pulls root nodes to the highest-numbered possible level and leaf nodes to the lowest-numbered possible level.

♦ `IlvHierarchicalLayout.OPTIMAL`

  This uses an algorithm that minimizes the sum of level distances for all edges. The optimal strategy is slower than the other strategies, but often produces the best result.

♦ `IlvHierarchicalLayout.HIGHER_LEVELS`

  Nodes have a tendency to use the possible level with the highest level number. All leaf nodes will be at the higest-numbered level. All root nodes are pulled to high-numbered levels as much as possible.

♦ `IlvHierarchicalLayout.LOWER_LEVELS`

  Nodes have a tendency to use the possible level with the lowest level number. All root nodes will be at level 0. All leaf nodes are pulled to low-numbered levels as much as possible.

♦ `IlvHierarchicalLayout.SPREAD_OUT`

  This is a combination of the lower-level and higher-level strategies. All root nodes will be at level 0. All leaf nodes will be at the higest-numbered level. All inner nodes are at balanced positions.

In CSS, you omit the prefix `IlvHierarchicalLayout` when specifying the value of the leveling strategy.

Level justification (HL)

If the layout uses horizontal levels, the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, or the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

If the layout uses vertical levels, the nodes of the same level are placed approximately at the same x-coordinate. In this case, the nodes can be justified to be aligned at the left border, at the right border, or at the center of the nodes that belong to the same level.

To specify the level justification towards the top:

In CSS
Add to the `GraphLayout` section:

```
levelJustification: "Top";
```

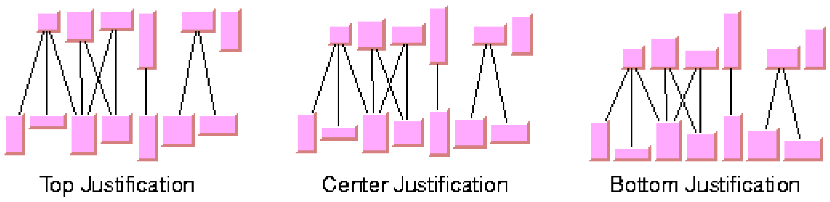
In Java
Use the method:

```
void setLevelJustification(int justification)
```

If the flow direction is to the top or to the bottom, the valid values for the level justification are:

♦ `IlvDirection.Top`

- `IlvDirection.Bottom`
- `IlvDirection.Center` (the default)

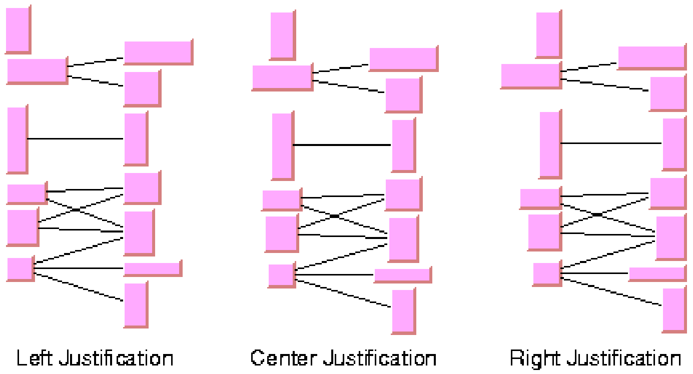


Level justification for horizontal levels

If the flow direction is to the left or to the right, the valid values for the level justification are:

- `IlvDirection.Left`
- `IlvDirection.Right`
- `IlvDirection.Center` (the default)

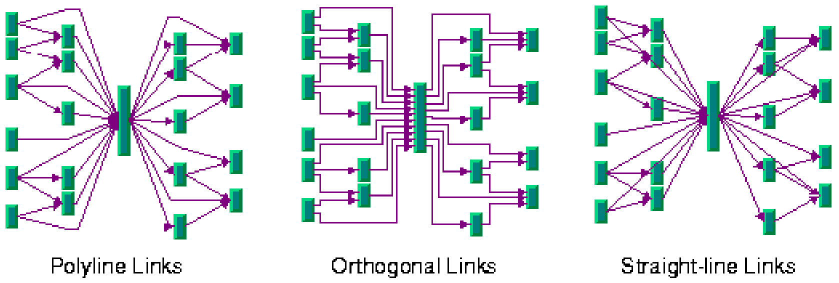
In CSS, you omit the prefix `IlvDirection` when specifying the value of the flow direction.



Level justification for vertical levels

Link style (HL)

The layout algorithm positions the nodes and routes the links. To avoid overlapping nodes and links, it creates bend points for the shapes of links. The link style parameter controls the position and number of bend points. The link style can be set globally, in which case all links have the same kind of shape, or locally on each link such that different link shapes occur in the same drawing.



Link styles

Link style and link shapes

Link styles work only when you use links that can be reshaped. Subclasses of `IlvPolylineLinkImage` or of `IlvSplineLinkImage`, (e.g., `IlvGeneralLink`) can be reshaped. Furthermore, link styles work only if free link connectors are installed. Free link connectors are subclasses of `IlvFreeLinkConnector`. If you use a diagram component, the free link connectors are automatically installed when needed unless specified differently. If you call layout on an `IlvGrapher` directly in Java, the layout algorithm may raise an `IlvInappropriateLinkException` if links are neither a subclass of `IlvPolylineLinkImage` nor of `IlvSplineLinkImage`, or if connectors are not a subclass of `IlvFreeLinkConnector`. In this case, you can use the methods `EnsureAppropriateLinkTypes`, `EnsureAppropriateLinkConnectors` or `EnsureAppropriateLinks` defined in the class `IlvGraphLayoutUtil` to replace inappropriate links or link connectors automatically, either before layout or when the `IlvInappropriateLinkException` is caught. For details on these methods, see the *Java API Reference Manual*.For details on the graph model, see *Using the Graph Model*.

Global link style

To set the global link style:

In CSS
Add to the `GraphLayout` section:

```
globalLinkStyle: "POLYLINE_STYLE";
```

In Java
Use the method:

```
void setGlobalLinkStyle(int style)
```

The valid values for the link style are:

- `IlvHierarchicalLayout.POLYLINE_STYLE`

  All links get a polyline shape. A polyline shape consists of a sequence of line segments that are connected at bend points. The line segments can be turned into any direction. This is the default value.

- `IlvHierarchicalLayout.ORTHOGONAL_STYLE`

All links get an orthogonal shape. An orthogonal shape consists of orthogonal line segments that are connected at bend points. An orthogonal shape is a polyline shape where the segments can be turned only in directions of 0, 90, 180 or 270 degrees.

- `IlvHierarchicalLayout.STRAIGHT_LINE_STYLE`

  All links get a straight-line shape. All intermediate bend points (if any) are removed. This often causes overlapping nodes and links.

- `IlvHierarchicalLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner. Note, however, that unlike fixed links, the links are not ignored completely. They are still used to calculate the leveling.

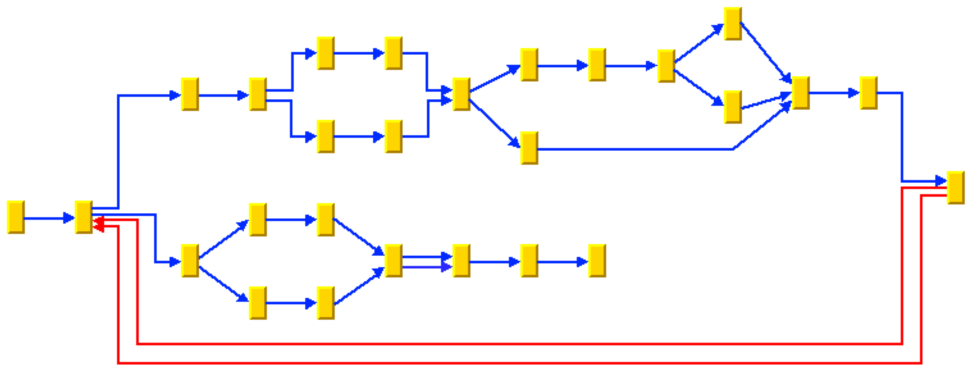
- `IlvHierarchicalLayout.MIXED_STYLE`

  Each link can have a different link style. The style of each individual link can be set such that different link shapes can occur in the same graph.

In CSS, you omit the prefix `IlvHierarchicalLayout` when specifying the value of the link style.

Individual link style

All links have the same style of shape unless the global link style is `MIXED_STYLE`. Only when the global link style is `MIXED_STYLE` can each link have an individual link style.



Different Link Styles Mixed in the Same Drawing

To specify the style of an individual link:

In CSS
First set the global link style to MIXED-STYLE, then specify a rule that selects the link, for instance:

```
GraphLayout {
   globalLinkStyle: "MIXED_STYLE";
}
#link1
{
  LinkStyle: "ORTHOGONAL_STYLE";
}
```

In Java

Use the methods:

```
void setLinkStyle(Object link, int style)
```

```
int getLinkStyle(Object link)
```

In this case, the link argument must be a graphic link (subclass of `IlvLinkImage`).

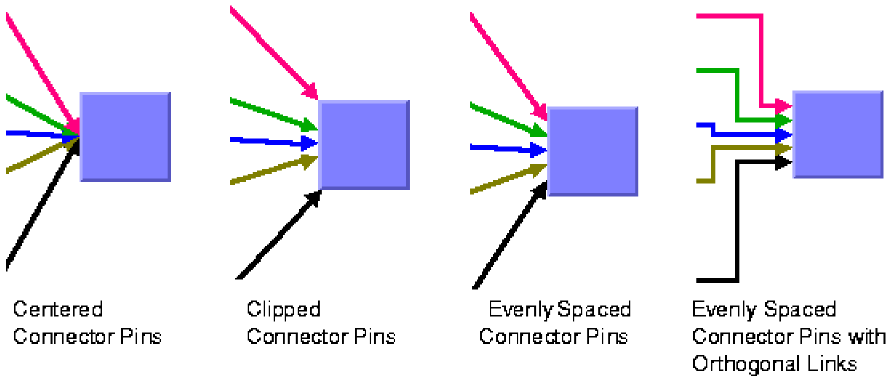
The valid values for the link style of local links are the same as for the global link style:

- `IlvHierarchicalLayout.POLYLINE_STYLE`
- `IlvHierarchicalLayout.ORTHOGONAL_STYLE`
- `IlvHierarchicalLayout.STRAIGHT_LINE_STYLE`
- `IlvHierarchicalLayout.NO_RESHAPE_STYLE`

Note: The link style of a Hierarchical Layout graph requires links in an `IlvGrapher` that can be reshaped. Links of type `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, and `IlvDoubleSplineLinkImage` cannot be reshaped. You should use the class `IlvPolylineLinkImage` or `IlvSplineLinkImage` instead.

Connector style (HL)

The layout algorithm positions the end points of links (the connector pins) at the nodes automatically. The connector style parameter specifies how these end points are calculated.



Connector styles

To specify the connector style:

In CSS
Add to the `GraphLayout` section:

```
connectorStyle: "CLIPPED_PINS";
```

In Java
Use the method:

```
void setConnectorStyle(int style)
```

The valid values for `style` are:

♦ `IlvHierarchicalLayout.CENTERED_PINS`

  The end points of the links are placed in the center of the border where the links are attached. This option is well-suited for polyline links and straight-line links. It is less well-suited for orthogonal links, because orthogonal links can look ambiguous in this style.

♦ `IlvHierarchicalLayout.CLIPPED_PINS`

  Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This option is particularly well-suited for polyline links without port specifications. It should not be used if a port side for any link is specified.

♦ `IlvHierarchicalLayout.EVENLY_SPACED_PINS`

  The connector pins are evenly distributed along the node border. This style guarantees that the end points of the links do not overlap. This is the best style for orthogonal links and works well for other link styles.

♦ `IlvHierarchicalLayout.AUTOMATIC_PINS`

  The connector style is selected automatically depending on the link style. If any of the links has an orthogonal style or if any of the links has a port side specification, the algorithm chooses evenly spaced connectors. If all the links are straight, it chooses centered connectors. Otherwise, it chooses clipped connectors.

In CSS, you omit the prefix `IlvHierarchicalLayout` when specifying the value of the connector style.

Note: The connector style parameter requires link connectors at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you use `IlvFreeLinkConnector` for link connectors to be used in combination with `IlvGrapher` objects. If you use a diagram component, the free link connectors are automatically installed when needed, unless specified differently.

End point mode (HL)

Normally, the layout algorithm is free to choose the termination points of each link. However, if fixed-link connectors are used (for instance, `IlvPinLinkConnector`), the user can specify that the current fixed termination pin of a link should be used.

The layout algorithm provides two end point modes. You can set the end point mode globally, in which case all end points have the same mode, or locally on each link, in which case different end point modes occur in the same drawing.

Global end point mode

To set the global end point mode:

In CSS
Add to the `GraphLayout` section:

```
globalOriginPointMode: "FIXED_MODE";
```

```
globalDestinationPointMode: "FIXED_MODE";
```

In Java
Use the methods:

```
void setGlobalOriginPointMode(int mode);
```

```
void setGlobalDestinationPointMode(int mode);
```

The valid values for `mode` are:

- `IlvLinkLayout.FREE_MODE` (the default)

  The layout is free to choose the appropriate position of the connection point on the origin/destination node.

- `IlvLinkLayout.FIXED_MODE`

The layout must keep the current position of the connection point on the origin/destination node.

- ♦ `IlvLinkLayout.MIXED_MODE`

  Each link can have a different end point mode.

In CSS, you omit the prefix `IlvHierarchicalLayout` when specifying the value of the end point mode.

The connection points are automatically considered as fixed if they are connected to grapher pins.

Individual end point mode

All links have the same end point mode unless the global end point mode is `IlvLinkLayout.MIXED_MODE`. Only when the global end point mode is set to `MIXED_MODE` can each link have an individual end point mode.

To set the end point mode of an individual link:

In CSS
First set the global point modes to MIXED_MODE, then specify a rule that selects the link, for instance:

```
LinkLayout {
   globalOriginPointMode      : "MIXED_MODE";
   globalDestinationPointMode : "MIXED_MODE";
}
#link1
{
  OriginPointMode            : "FIXED_MODE";
  DestinationPointMode       : "FIXED_MODE";
}
```

In Java
Use the methods:

```
void setOriginPointMode(Object link, int mode);
```

```
int getOriginPointMode(Object link);
```

```
void setDestinationPointMode(Object link, int mode);
```

```
int getDestinationPointMode(Object link);
```

The valid values for `mode` are:

- ♦ `IlvLinkLayout.FREE_MODE` (the default)
- ♦ `IlvLinkLayout.FIXED_MODE`

The connection points are automatically considered as fixed if they are connected to grapher pins.

Using a link connection box interface (HL)

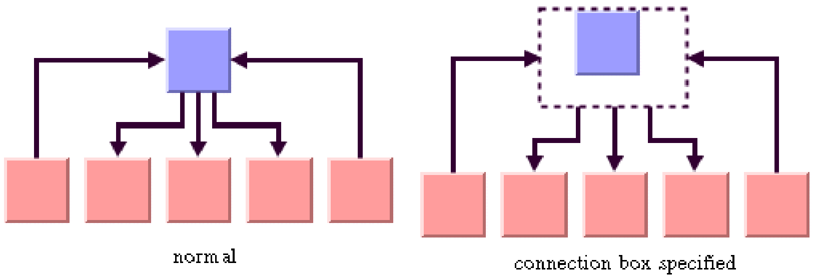
By default, the connector style determines how the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface in Java, use the method:

```
setLinkConnectionBoxInterface
```

The link connection box interface provides each node with a link connection box and tangential shift offsets. The Hierarchical Layout uses the link connection box but does not use the tangential offsets.

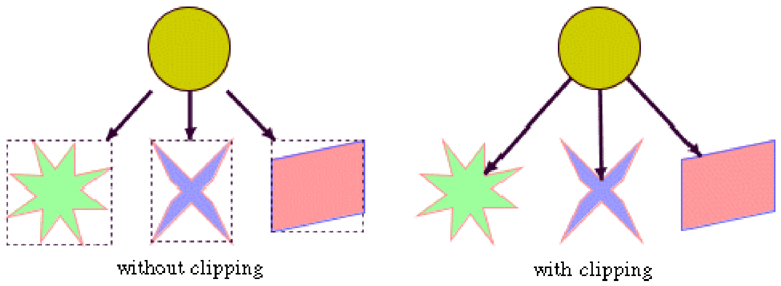
The following figure illustrates the effects of customizing the connection box. On the left is the result without any connection box interface. The picture on the right shows the effect if the connection box interface returns the dashed rectangle for the blue node.



Effect of connection box interface

Using a link clipping interface (HL)

By default, the Hierarchical Layout places the connection points of links at the border of the bounding box of the nodes. If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of link clipping interface

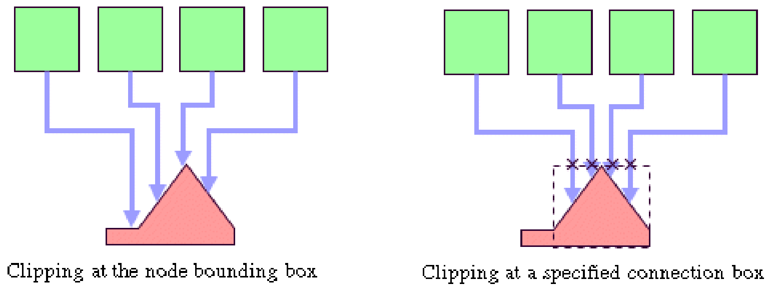
You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*. To set a link clip interface in Java, use the method:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: Additionally to the link clip interface, the `IlvClippingLinkConnector` can be used. This special link connector updates the clipped connection points automatically during interactive node movements.

The connector style, the link connection box interface, and the link clip interface work together in the following way: by respecting the connector style, the proposed connection points are calculated on the rectangle obtained from the link connection box interface (or on the bounding box of the node, if no link connection box interface was specified). Then, the proposed connection point is passed to the link clip interface and the returned connection points are used to connect the link to the node.

The following figure shows an example of the combined effect.


Clipping at the node bounding box
Clipping at a specified connection box

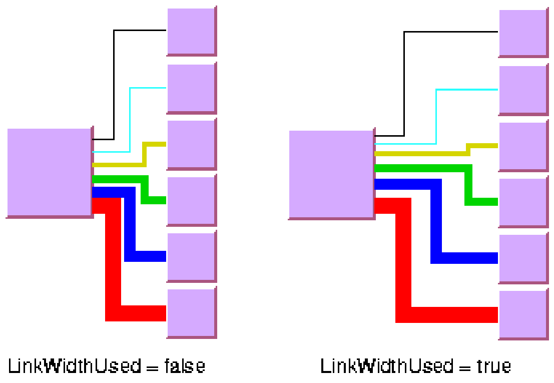
Combined effect of link clipping interface and link connection box

If the links are clipped at the red node in previous figure (left), they appear unsymmetrical with respect to the node shape, because the relevant part of the node (here: the triangle) is not in the center of the bounding box of the node, but the proposed connection points are calculated with respect to the bounding box. This can be corrected by using a link connection box interface to explicitly specify a smaller connection box for the relevant part of the node (previous figure, right) such that the proposed connection points are placed symmetrically at the triangle of the node.

For experts: thick links (HL)

If evenly spaced pins are used as connector style, the links can be evenly spaced with respect to the link center or with respect to the link border. The difference is only visible when links that connect to the same node have different widths. For instance, when the link width indicates the cost or capacity of a flow in the application, many different link width may occur.

Using the link width shows the effect of using different link widths. In the drawing on the left, the center of the links are evenly distributed at the left node. Each link has the same space available at the node side. Therefore, the thick links appear closer to each other than do the thinner links and the offsets between the link borders are different. In the drawing on the right, the thick links have more space available than do the thinner links. The offset between the link border (at the segments that connect to the left node) is constant because the link width is considered in the calculation of the connection points.



Using the link width

To enable the connector calculation to respect the link width:

In CSS
Add to the `GraphLayout` section for instance the statement:

```
linkWidthUsed: "true";
```

In Java
Call:

```
layout.setLinkWidthUsed(true);
```

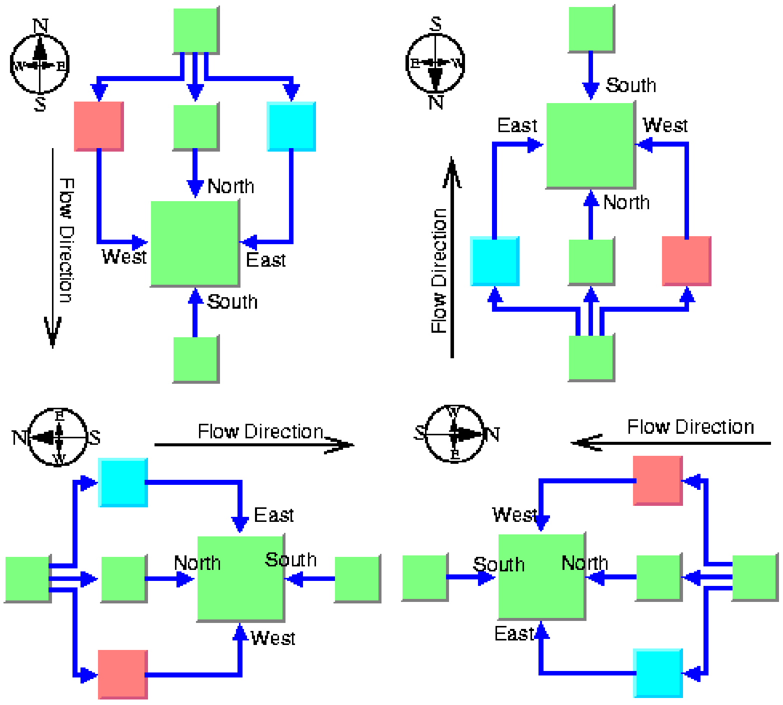
The link width setting is disabled by default. The link width has no effect if the connector styles `CENTERED_PINS` or `CLIPPED_PINS` are used.

Port sides parameter (HL)

The Hierarchical Layout algorithm produces a layout where the majority of the links flow are in the same direction. If the flow direction is towards the bottom, usually the incoming links are connected to the top side of the node and the outgoing links are connected to the bottom side of the node. It is also possible to specify on which side a link connects to the node.

To simplify the explanations of the port sides, we use the compass directions *north, south, east,* and *west*. The specified link flow direction is always towards south and the first level is towards north. If the flow direction is towards bottom, north is at the top, south at the bottom, east on the right, and west on the left side of the drawing. If the flow direction is towards right, north is on the left, south on the right, east at the top, and west at the bottom.

Link connections to port sides shows a drawing where the links connect to the larger middle node at the specified port sides. A compass icon shows the compass directions in these drawings.



Link connections to port sides

You can set at which side the link connects to its source node.

To set at which side the link connects to its source node:

In CSS
Specify a rule that selects the link, for instance:

```
#link1 {
  FromPortSide: "NORTH";
}
```

In Java
Use the method:

```
void setFromPortSide(Object link, int side);
```

In a similar way, you can set at which side the link connects to its destination node.

To set at which side the link connects to its destination node:

In CSS
Specify a rule that selects the link, for instance:

```
#link1 {
  ToPortSide: "SOUTH";
}
```

In Java
Use the method:

```
void setToPortSide(Object link, int side);
```

The valid values for `side` are:

- `IlvHierarchicalLayout.UNSPECIFIED` (the default)
- `IlvHierarchicalLayout.NORTH`
- `IlvHierarchicalLayout.SOUTH`
- `IlvHierarchicalLayout.EAST`
- `IlvHierarchicalLayout.WEST`

In CSS, you omit the prefix `IlvHierarchicalLayout` when specifying the value of the port side.

To retrieve the current choice for a link, use the methods:

```
int getFromPortSide(Object link);
```

```
int getToPortSide(Object link);
```

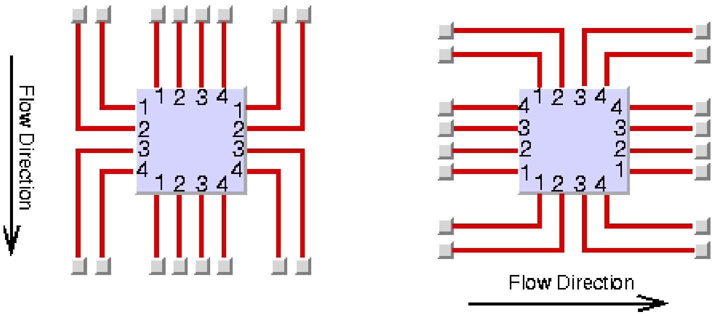
The port sides east and west work particularly well with the orthogonal link style. Polyline links with these port sides sometimes have unnecessary bends. Furthermore, if port sides are specified, the connector style `CLIPPED_PINS` should not be used.

Port index parameter (HL)

Instead of asking the layout algorithm to decide at which point a link connects to the node border, you can specify where the links connect to the node. You cannot specify the exact location, but you can specify the relative location compared to the connection points of the other links. This is done by using a port index. *Sample layout with ports and orthogonal link style* shows a sample layout with ports at many nodes.

Links that have the same port index connect at the same point of the node. The ports are evenly distributed at the node sides, in a similar way as with the connector style `EVENLY_SPACED_PINS`.The ports are ordered according to their indices. On the north and south side of a node, the port indices increase toward the east. On the east and west sides of a node, the port indices increase toward the south. By using port indices in this way, it is easier to rotate a graph by simply changing the flow direction without needing to update all the port specifications.

Port Index Numbering Conventions in Relation to Flow Direction show how the port indices depend on the flow direction.



Port Index Numbering Conventions in Relation to Flow Direction

Port numbers are normally used in combination with port sides. Therefore, you must specify how many ports are available on each side of a node.

To specify the number of ports:

In CSS
Write a rule that selects the node, for instance:

```
node.tag1 {
  EastNumberOfPorts: "4";
  WestNumberOfPorts: "4";
  NorthNumberOfPorts: "4";
  SouthNumberOfPorts: "4";
}
```

Alternatively, you can write:

```
node.tag1 {
  NumberOfPorts: "EAST,4";
  NumberOfPorts: "WEST,4";
  NumberOfPorts: "NORTH,4";
  NumberOfPorts: "SOUTH,4";
}
```

Both are equivalent.

In Java
Use the method:

```
void setNumberOfPorts(Object node, int side, int numberOfPorts);
```

For example, to use 4 ports on each side of a specific node, use the calls:

```
layout.setNumberOfPorts(node, IlvHierarchicalLayout.EAST, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.WEST, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.NORTH, 4);
layout.setNumberOfPorts(node, IlvHierarchicalLayout.SOUTH, 4);
```

The node side is specified again by `EAST`, `WEST`, `NORTH`, and `SOUTH`. To retrieve the retrieve the number of ports available at the node, use the method:

```
int getNumberOfPorts(Object node, int side);
```

After the number of ports per side is specified, you can choose which port each link connects to.

To choose the port side and the port index for a link:

In CSS
Specify a rule that selects the link, for instance:

```
link.tag1 {
  FromPortSide: "NORTH";
  FromPortIndex: "3";
  ToPortSide: "SOUTH";
  ToPortIndex: "3";
}
```

In Java
To specify the connection at the source node, use the methods:

```
void setFromPortSide(Object link, int portSide);
```

```
void setFromPortIndex(Object link, int portIndex);
```

To specify the connection at the destination node, use the methods:

```
void setToPortSide(Object link, int portSide);
```

```
void setToPortIndex(Object link, int portIndex);
```

To obtain the current port index of a link, use the methods:

```
int getFromPortIndex(Object link);
```

```
int getToPortIndex(Object link);
```

Using the port side and port index specifications are additional constraints for the layout algorithm. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, if too many links have a specified port index, this resulting layout may have more link crossings and be less balanced.

Fork link shapes (HL)

If several links start at the same position and are orthogonally routed, it is sometimes preferred that the links share the first two link segments. The shape of a link bundle of this kind looks like a fork. To enable the fork shape mode for outgoing links, call:

```
layout.setFromFork(true);
```

To enable the fork shape mode for incoming links:

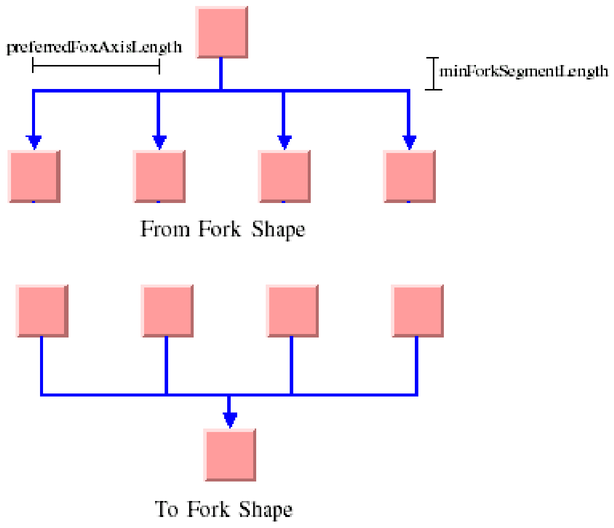
In CSS
Add to the `GraphLayout` section:

```
fromFork: "true";
toFort: "true";
```

In Java
Call:

```
layout.setToFork(true);
```

These statements have an effect only if the links are routed orthogonally. The fork appears only at those links that start or end exactly at the same point. Specifying `setFromFork(true)` by itself does not force the links to start at the same point. To force links to start or end at the same point, use the center connector style (see *Connector style (HL)*) or specify the same port for the links (see *Port index parameter (HL)*).



Fork Link Shapes

There are two spacing parameters for the fork shape:

In CSS
Add to the `GraphLayout` section:

```
minForkSegmentLength: "30.0";
preferredForkAxisLength: "10.0";
```

In Java

```
void setMinForkSegmentLength(float length)
```

It sets the minimal length of the segment that is directly adjacent to the node.

```
void setPreferredForkAxisLength(float length)
```

This method sets the preferred length of the fork axis per branch (the second segment adjacent to the node). If the fork has five branches, the entire axis has the preferred length five times the specified parameter. The preferred fork axis length is only a hint for the layout algorithm. If enough space is available, the algorithm will enlarge the fork axis to avoid unnecessary link bends. If there is not enough space, the algorithm may as well calculate a fork axis that is smaller than the preferred one.

Fork link shapes may sometimes look ambiguous, in particular when a link starts at the same point where another link ends, because in this case it is impossible to recognize whether the arrowhead belongs to one or the other link.

Link priority parameter (HL)

The layout algorithm tries to place the nodes such that all links are short, point in the flow direction, and do not cross each other. However, this is not always possible. Often, links cannot have the same length. If the graph has cycles, some links must be reversed against the flow direction. If the graph is a nonplanar graph, some links have to cross each other.

The link priority parameter controls which links should be selected if long, reversed, or crossing links are necessary. Links with a low priority are more likely to be selected than links with a high priority. This does not mean that low-priority links are always longer, reversed, or crossed, because the graph may have a structure such that no long, reversed or crossing links are necessary.

To set the link priority:

In CSS
Specify a rule that selects the link, for instance:

```
link.tag1 {
  LinkPriority: "2.0";
}
```

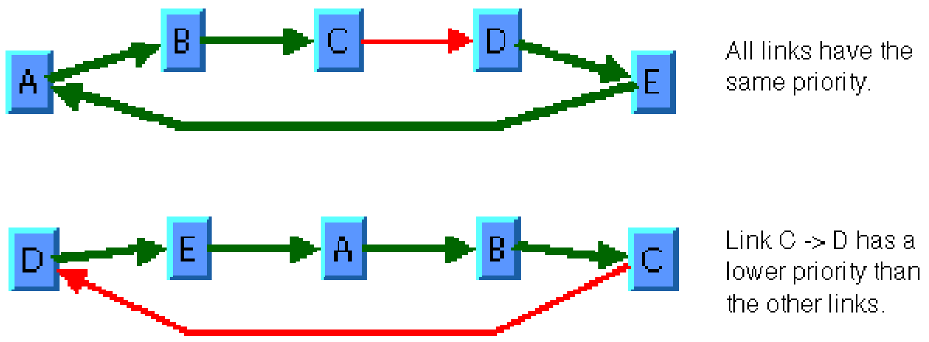
In Java
Use the methods.

```
void setLinkPriority(Object link, float priority)
```

```
float getLinkPriority(Object link)
```

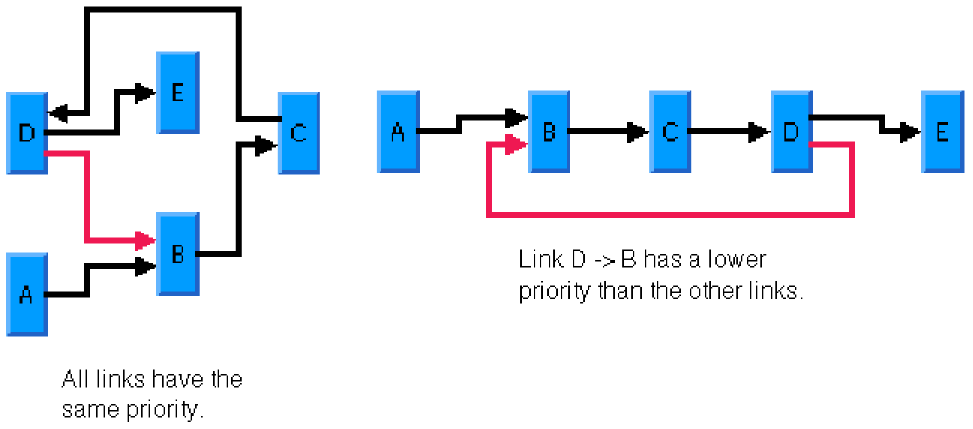
The default value of the link priority is 1.0. Negative link priorities are not allowed.

For an example of using the link priority, consider a cycle A->B->C->D->E->A. It is impossible to lay out this graph without reversing any link. Therefore, the layout algorithm selects one link to be reversed. To control which link is selected, you can give one link a lower priority than the others. This link will be reversed. In *Working with link priorities*, the bottom layout shows the use of the link priority. The link C->D was given the priority 0.5, while all the other links have the priority 1.0. Therefore C-D is reversed. The top layout in *Working with link priorities* shows what happens when all links have the same priority. Link E->A is reversed.


A
B
C
D
E
All links have the same priority.
D
E
A
B
C
Link C -> D has a lower priority than the other links.

Working with link priorities

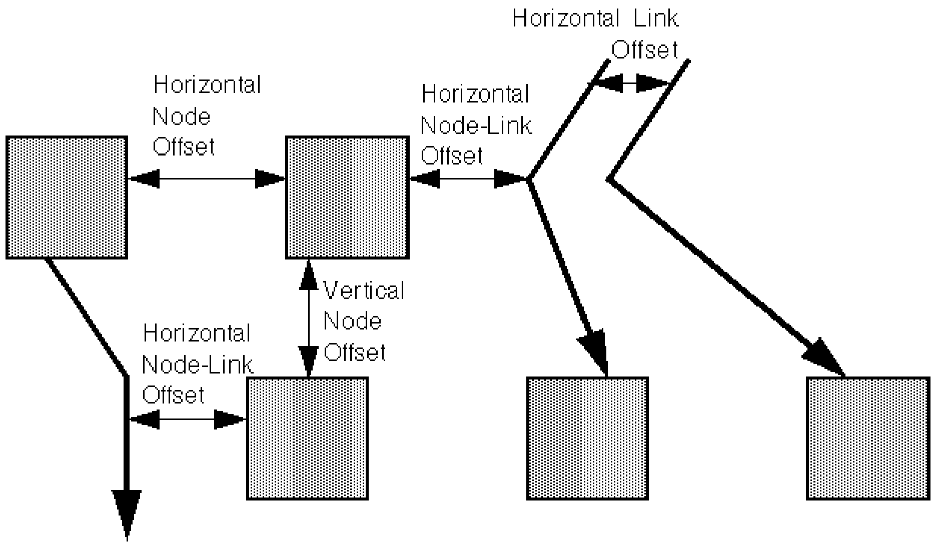
The use of link priorities is important in combination with ports. Links with “from” ports on the south side and “to” ports on the north side are preferably laid out opposite to the flow direction. Such a feedback link may cause parts of the drawing to tip over. *Using Link Priorities and Ports* shows an example. The red link is a feedback link with port specifications. To obtain the correct result as shown in the right side of the following figure, you would set the priority of the feedback link to a very low value.


D
E
C
B
A
All links have the same priority.
A
B
C
D
E
Link D -> B has a lower priority than the other links.

Using Link Priorities and Ports

Spacing parameters (HL)

The spacing of the layout is controlled by three kinds of spacing parameters: the minimal offset between nodes, the minimal offset between parallel segments of links and the minimal offset between a node border and a bend point of a link or a link segment that is parallel to this border. The offset between parallel segments of links is at the same time the offset between bend points of links. All three kind of parameters occur in both directions: horizontally and vertically.



Spacing parameters

To set the spacing parameters:

In CSS
Add to the `GraphLayout` section:

```
horizontalNodeOffset: "30.0";
horizontalLinkOffset: "15.0";
horizontalNodeLinkOffset: "20.0";
verticalNodeOffset: "30.0";
verticalLinkOffset: "15.0";
verticalNodeLinkOffset: "20.0";
```

In Java

♦ For the horizontal direction, use the methods:

```
void setHorizontalNodeOffset(float offset)
```

```
void setHorizontalLinkOffset(float offset)
```

```
void setHorizontalNodeLinkOffset(float offset)
```

♦ For the vertical direction, use the methods:

```
void setVerticalNodeOffset(float offset)
```

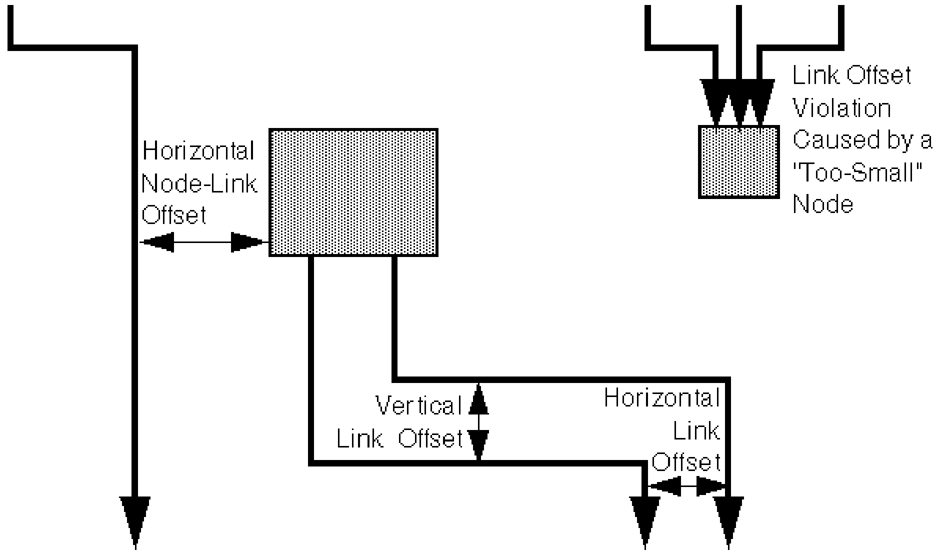
```
void setVerticalLinkOffset(float offset)
```

```
void setVerticalNodeLinkOffset(float offset)
```

For a layout with horizontal levels (the flow direction is to the top or to the bottom), the horizontal node offset is the minimal distance between nodes of the same level. The vertical node offset is the minimal distance between nodes of different levels, that is, the minimal distance between the levels. For non-orthogonal link styles, the horizontal link offset is basically the minimal distance between bend points of links. The horizontal node-link offset is the minimal distance between the node border and the bend point of a link. For horizontal levels, the vertical link offset and the vertical node-link offset play a role only if the link shapes are orthogonal.

Similarly, for a layout with vertical levels (the flow direction is to the left or to the right), the vertical node offset controls node distances within the levels. The horizontal node offset is the minimal distance between the levels. In this case, the vertical link offset and the vertical node-link offset always play a role, while the horizontal link offset and the horizontal node-link offset affect the layout only with orthogonal links.

For orthogonal links, the horizontal link offset is the minimal distance between parallel, vertical link segments. The vertical link offset is the minimal distance between parallel, horizontal link segments. However, the layout algorithm cannot always satisfy these offset requirements. If a node is very small but has many incident links, it may be impossible to place the links orthogonally with the specified minimal link distance on the node border. In this case, the algorithm places some link segments closer than the specified link offset.


Horizontal Node-Link Offset
Link Offset Violation Caused by a "Too-Small" Node
Vertical Link Offset
Horizontal Link Offset

Spacing parameters for orthogonal links

Incremental mode with HL

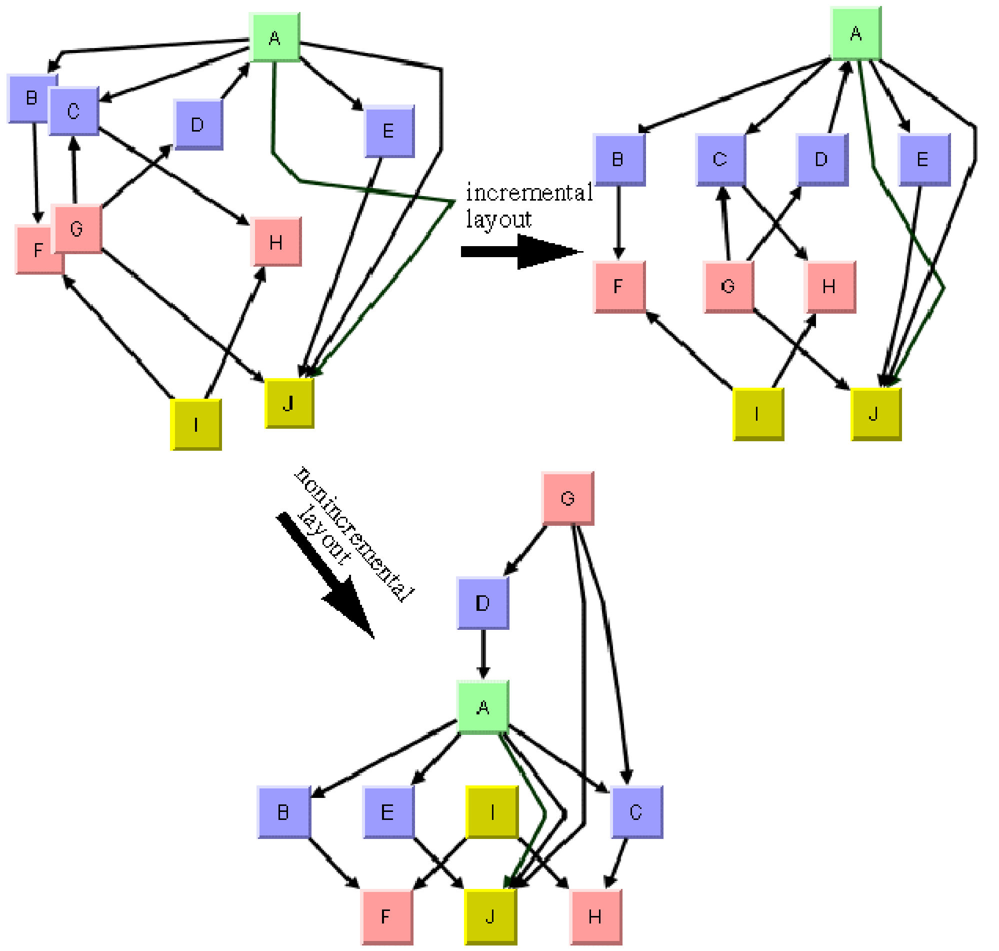
In some circumstances you may need to use a sequence of layouts on the same graph. For example:

- ♦ You work with graphs that have become out-of-date and you need to extend the graph. If you perform a layout on the extended graph, you probably want to identify the parts that were already laid out in the original graph. The layout should not change very much when compared with the layout of the original graph.
- ♦ The first layout results in a drawing with minor deficiencies. You want to solve these deficiencies manually and perform a second layout to clean up the drawing. The second layout probably should not greatly change the parts of the graph that were already acceptable after the first layout.

The Hierarchical Layout normally works nonincrementally. It performs a layout from scratch and moves all nodes to new positions and reroutes all links. The previous positions of nodes have no influence on the result of the layout. Hence, even a small change can cause a large effect on the next layout.

But the Hierarchical Layout also supports incremental sequences of layout that “do not change very much.” It can place the nodes close to their previous positions, so that you can more easily identify the parts that had already been laid out in the original graph. Incremental mode takes the previous positions of the nodes into account. In this mode the algorithm preserves the relative order of the levels and the nodes within the levels in the subsequent layout. It does not preserve the absolute positions of the nodes, but it tries to detect the structure of the previous layout by examining the node coordinates. For instance, if two nodes are in the same level, then they stay in the same level after an incremental layout. If a node is in a higher level than another node, it stays in the higher level.

The following figure illustrates the difference between an incremental and nonincremental layout.



Incremental and Nonincremental Layouts

Incremental mode is disabled by default.

To enable incremental mode:

In CSS
Add to the `GraphLayout` section:

```
incrementalMode: "true";
```

In Java™

```
layout.setIncrementalMode(true);
```

Important: Be aware of the difference between the incremental mode of the Hierarchical Layout and the incremental layout flag of the SDM Graph Layout Renderer.

The CSS statement `incrementalMode: "true";`

controls the incremental mode of Hierarchical Layout and specifies that all layouts are performed incrementally.

This feature of the Hierarchical Layout is described in the subsequent sections.

Conversely, the CSS statement

`incrementalLayout: "true";s`

controls the Node Layout Renderer and means that the Node Layout Renderer switches between incremental mode and nonincremental mode depending on whether an object is selected or not.

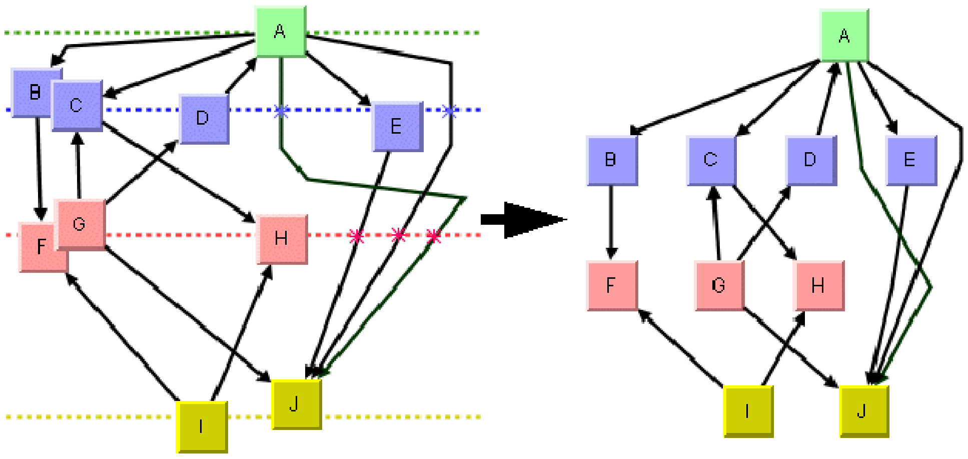
This advanced feature of the Node Layout Renderer is explained in The GraphLayout renderer in *Developing with the JViews Diagrammer SDK.*

Phases of the incremental mode

The layout algorithm analyzes the drawing in incremental mode in the following way:

1. First, it determines from the node coordinates which nodes must belong to the same level. For instance, if the flow direction is towards the bottom, it tries to detect horizontal reference lines at those vertical positions where many nodes are placed along a line. The specified vertical node offset helps to detect these lines because the horizontal reference lines should be approximately the vertical node offset apart. See the following figure.
2. All nodes that touch the same reference line are assigned to the same level.
3. It determines the order of the nodes within each level by analyzing where the node touches the reference line. For instance, if the flow direction is towards the bottom, it determines from the x coordinate of the nodes how they are ordered within the levels.
4. If long links span several levels, the algorithm can preserve the shape of a long link. It determines the point where a link crosses the level reference line. It creates a bend point for the long link inside the level. It tries to preserve the order of the bend points in each level. For instance, if in a flow direction towards the bottom, a long link bypasses another node on the right side, then the incremental layout tries to find a similar shape of the link that bypasses the node on the right side, as illustrated in the following figure.
5. Finally, the layout tries to calculate the absolute positions of the nodes that respect the levels and the ordering within the levels. It tries to balance the node positions. However, it also tries to place each node close to its previous position. Both criteria often compete with each other, because to get a perfect balance, nodes must sometimes move far from their original position. The Hierarchical Layout contains a parametrized heuristic to satisfy both criteria.

The following figure shows the result of the incremental phases.



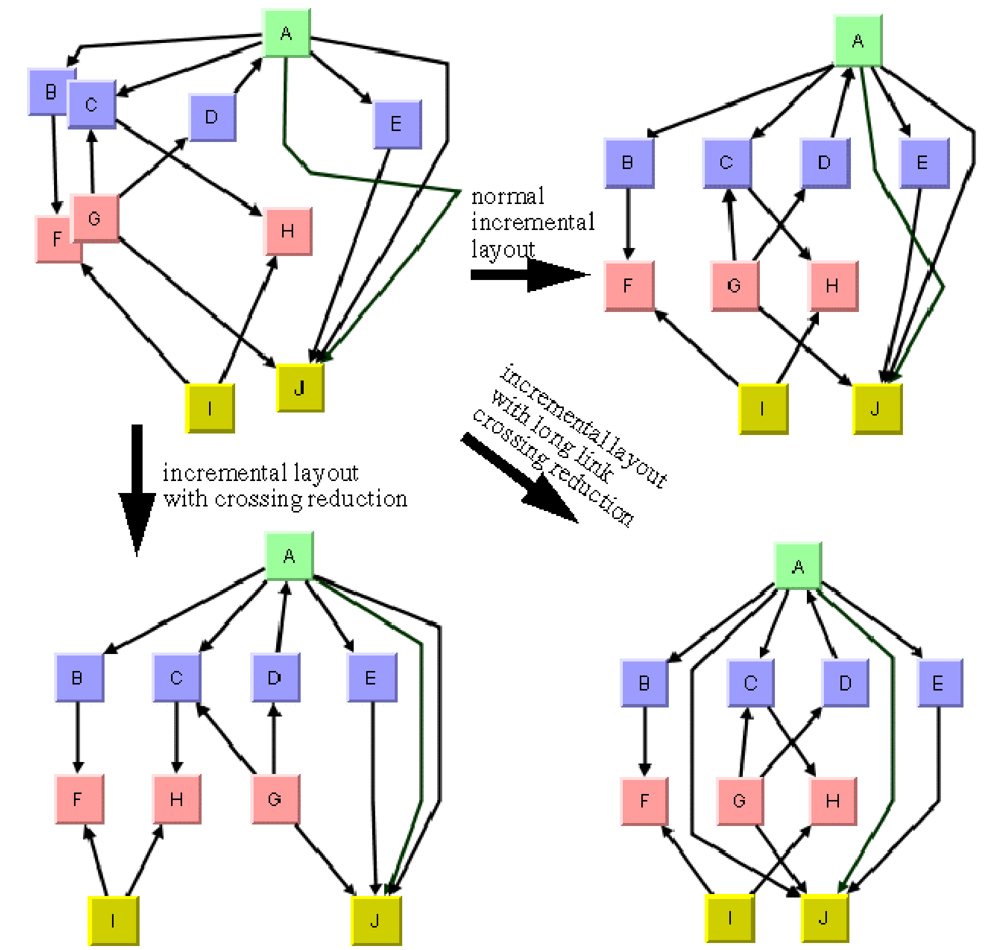
Incremental layout phases

Expert parameters of the incremental mode

Each phase of the incremental mode can be parameterized. These layout parameters have an effect only if incremental mode is switched on.

Minimizing long link crossings

The incremental layout tries to preserve the shape of long links that cross several levels. This implies that link crossings between long links are not resolved. If crossings of long links are not desired, it may be better to reroute long links from scratch. The following figure shows four hierarchy trees, with the original layout at the upper left. The bottom right shows the result if long links are rerouted, and the top right shows the result if the shape of long links is preserved.


A
B
C
D
E
F
G
H
I
J
normal incremental layout
incremental layout with crossing reduction
incremental layout with long link crossing reduction

Crossing Reduction During Incremental Layouts

To reroute long links from scratch, you must enable the crossing reduction mechanism for long links:

In CSS

```
longLinkCrossingReductionDuringIncremental: "true";
```

In Java

```
layout.setLongLinkCrossingReductionDuringIncremental(true);
```

The crossing reduction of long links determines only the shape of the links. It does not influence the order of the other nodes within the levels.

Minimizing all link crossings

Optionally, you can apply a crossing reduction to all nodes within each level. In this case, the incremental layout determines from the node coordinates which nodes belong to the same level, but it may reorder the nodes within the levels completely to avoid link crossings. It also reorders the long links in this case. The previous figure, bottom left shows the result. Notice that the order of the nodes “F,” “G,” and “H” have changed to resolve the link crossings.

To enable the crossing reduction for all nodes:

In CSS

```
crossingReductionDuringIncremental: "true";
```

In Java

```
layout.setCrossingReductionDuringIncremental(true);
```

Setting absolute level positioning

The incremental layout tries to place the nodes in absolute positions that are close to the previous positions. It tries to avoid nodes moving a large distance, because even if the relative order of the nodes within the levels does not change, large movement distances can be confusing for users. It is much easier to keep a mental map of the diagram if the nodes remain close to the previous positions.

The following figure illustrates node repositioning with and without taking the previous positions into account. The incremental layout of the original graph at the top left results in the graph at the top right, which is easier to recognize as the same graph than the graph at the bottom.

The absolute level positioning feature is enabled by default, but it can be disabled.

To disable the absolute level positioning feature:

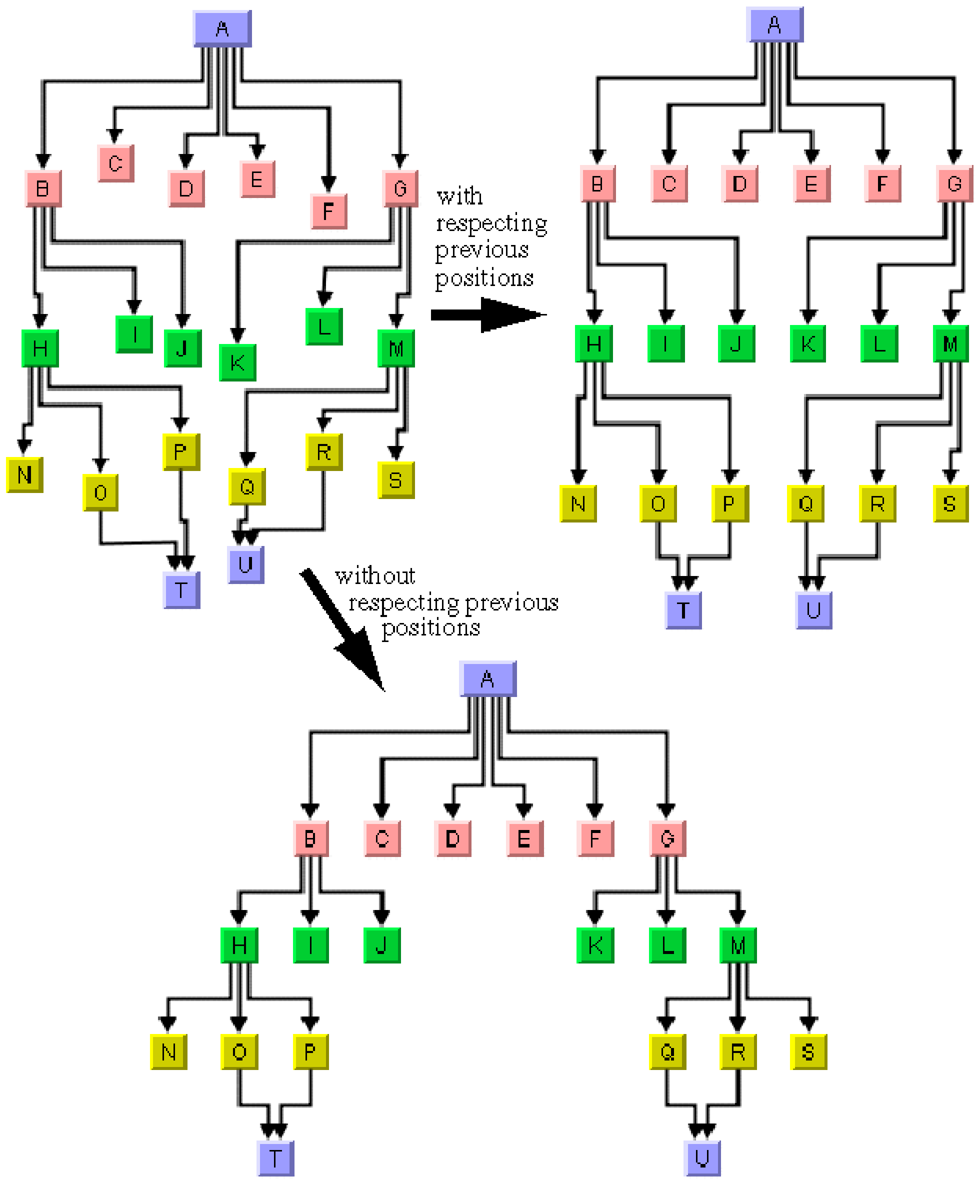
In CSS
Write the statement:

```
incrementalAbsoluteLevelPositioning: "false";
```

In Java
Call

```
layout.setIncrementalAbsoluteLevelPositioning(false);
```

With this statement, the layout does not try to place the nodes close to the previous positions. It places the nodes such that the layout is balanced. However, to create a perfect balance, the layout may need to move a few nodes so far apart that you can no longer recognize the diagram after the layout from the node positions that were shown in the previous layout (see the following figure, bottom).



with respecting previous positions
without respecting previous positions

Absolute Positioning During Incremental Layouts

Setting absolute level position range and tendency

If absolute level positioning is enabled, it competes with the aesthetic criteria to create a balanced layout. Due to the fact that nodes must stay close to their previous positions, the diagram after incremental layout may be somewhat unbalanced and unsymmetrical. The

Hierarchical Layout algorithm uses a heuristic that you can influence by two parameters, the absolute level position range and tendency.

The absolute level positioning feature is enabled by default, but it can be disabled.

To disable the absolute level positioning feature:

In CSS
Add in the `GraphLayout` section:

```
incrementalAbsoluteLevelPositioning: "100";
```

In Java
Call:

```
layout.setIncrementalAbsoluteLevelPositionRange(100);
```

This statement specifies that within the range of 100 coordinate units from the old position of the node, the balance is the only criteria for the placement. This means that a node whose optimal position is less than 100 coordinate units away from its previous position is placed exactly at its optimal position. Nodes whose optimal position is farther away are placed at a position that is a compromise between previous position and optimal position. This is illustrated in figure below, right.

To set the absolute level position tendency:

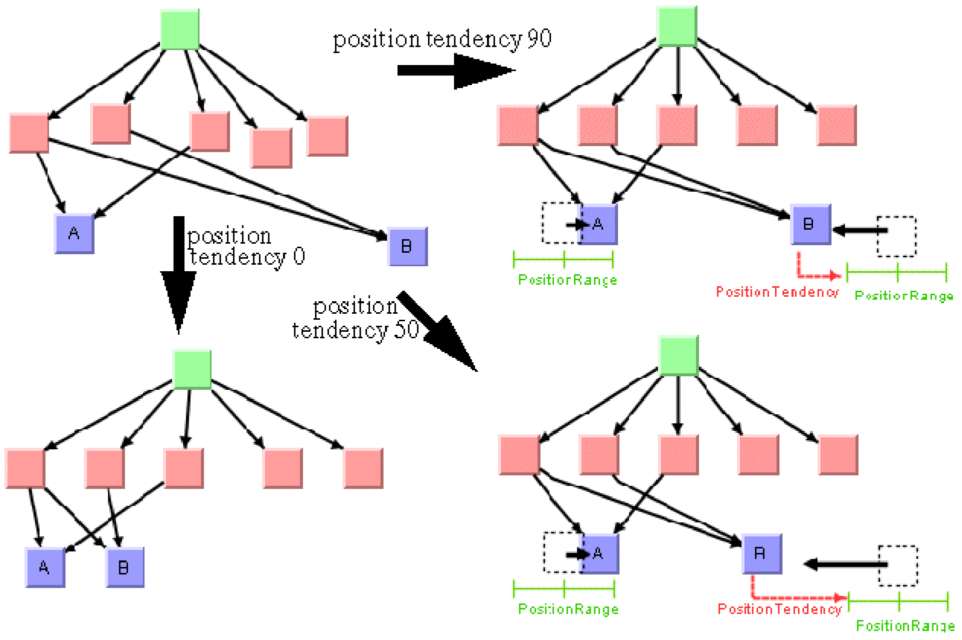
In CSS
Add in the `GraphLayout` section:

```
incrementalAbsoluteLevelPositionTendency: "70";
```

In Java
Call:

```
layout.setIncrementalAbsoluteLevelPositionTendency(70);
```

This statement specifies that positions of nodes whose optimal positions are far away from their previous position are 70% influenced by their previous position and 30% influenced by their optimal positions. Imagine a rubber band that tries to pull a node to its previous position, and another rubber band that tries to pull the same node to its optimally balanced position. The level position tendency `70` means that one rubber band pulls with 70% of the force towards the previous position, and the other rubber band pulls with 30% towards the optimal position. Increasing the tendency means that the node stays closer to its old position, decreasing it means that the node moves closer to its optimal position. If you set the tendency to 0%, this has the same effect as disabling the incremental absolute level positioning (see the following figure).



Absolute Positioning During Incremental Layouts

Marking nodes for incremental layout

Incremental layout normally treats all nodes and links of the drawing in the same way. However, you may have added nodes and links to the drawing programmatically, and the new nodes and links do not have meaningful coordinates yet. Perhaps you have placed them all at the origin (0,0), or at random coordinates. In this case, you need an incremental layout that takes the coordinates of all nodes into account that were previously laid out, while it ignores the coordinates of all new nodes. The incremental mode of the Hierarchical Layout allows you to specify in Java which nodes cannot be laid out incrementally by calling the method:

```
layout.markForIncremental(nodeOrLink);
```

If you call this statement, the node or link is marked such that its coordinates are ignored during the next incremental layout. The positions of marked nodes and links are calculated from scratch. The mark is valid only until the next layout and is automatically cleared afterwards.

Layout constraints for HL

The Hierarchical Layout algorithm supports *relative position constraints* on nodes. Such a constraint is a rule on how a particular node (or a group of nodes) must be placed with respect to the other nodes. The constraints influence the relative positions. For example, you can force node `A` to be on the left side of node `B`, so the position of `A` is expressed relative to the position of `B`. It is theoretically possible to specify contradicting constraints: if you specify that node `A` must be on the left side of `B` and `B` must be on the left side of `A`, then these constraints are not solvable at the same time. If `A` is on the left side of `B`, then `B` must be on the right side of `A`. The Hierarchical Layout algorithm tries to detect and resolve constraint conflicts automatically. It ignores those constraints that are infeasible. Since the automatic constraint resolution is time consuming, it is recommended to specify nonconflicting constraints when possible.

Constraints should be used only if the incremental mode is switched off. In fact, the incremental mode is implemented by means of additional constraints that are added internally. Hence, if you use constraints during the incremental mode, it is very likely that the system detects so many constraint conflicts that you get unexpected results.

Constraints should be used carefully. The more constraints are specified, the more difficult it is to calculate a layout. Therefore, this resulting layout may have more link crossings and be less balanced than a graph with no constraints.

Each type of constraint is represented by a subclass of `IlvHierarchicalConstraint`. The following constraint types are available:

`IlvLevelRangeConstraint`	Forces a node into a range of certain levels
`IlvSameLevelConstraint`	Forces two nodes to the same level.
`IlvRelativeLevelConstraint`	Forces a node to a lower/higher level than another node.
`IlvGroupSpreadConstraint`	Forces a group of nodes on levels that are no more than a specified spread value apart.
`IlvRelativePositionConstraint`	Forces a node to a lower/higher position than another node of the same level.
`IlvSideBySideConstraint`	Forces two nodes of the same level to be placed side by side.
`IlvExtremityConstraint`	Forces a node to the first or last level, or to the first or last position within a level.
`IlvSwimLaneConstraint`	Forces a group of nodes into the same rectangular swim lane area.

Specifying constraints in CSS for HL

There are two ways to specify constraints in a style sheet:

- In an external file that contains the constraints
- By means of style rules in the style file directly

When you do not use a diagram component with style sheet capabilities, you can still use constraints.

Writing style rules that specify constraints is very powerful but complex. It is illustrated in section *For experts: specifying constraints in CSS directly (HL)*. This topic concentrates on specifying constraints in an external file.

In the CSS file, you can write:

```
GraphLayout {
    graphLayout          : "Hierarchical";
    constraintsURL       : "url(Constraints.txt)";
}
```

This means that the layout constraints are specified in a separate file named `Constraints.txt`. This file contains the constraint in a very simple format, for example:

```
SameLevelConstraint {
  firstNode: node2
  secondNode: node5
}
RelativeLevelConstraint {
  lowerSubject: { node5 }
  higherSubject: { node9 }
  priority: 50.0
}
LevelRangeConstraint {
  subject: { node11, node12 }
  minLevel: 3
  maxLevel: 4
}
```

The complete example can be found at:

<installdir>/jviews-diagrammer86/codefragments/graphlayout/hierarchicallayout/constraints/resources/Constraints.txt

The style files are in the subdirectory `data`. The example shows how to specify constraints by loading an external file as well as how to specify constraints by writing style rules directly.

See also

Adding and removing constraints in Java for HL

Adding and removing constraints in Java for HL

You can add constraints to the Hierarchical Layout in Java™ . You allocate a new constraint object and call this method on the IlvHierarchicalLayout instance:

```
void addConstraint(IlvHierarchicalConstraint constraint);
```

You can add as many constraints as you want. The constraints will be respected during the subsequent layout calls until you remove them. To remove the most recent constraint, call:

```
void removeConstraint();
```

To remove a specific constraint, call:

```
void removeConstraint(IlvHierarchicalConstraint constraint);
```

To remove all existing constraints, call:

```
void removeAllConstraints();
```

You can retrieve the constraints that were added to a Hierarchical Layout with the method:

```
Enumeration getConstraints()
```

Node groups

Some constraints affect single nodes. Other constraints affect groups of nodes. The class IlvNodeGroup is a convenient way to specify a group of nodes in Java. You can create a group of nodes in the following way:

```
group = new IlvNodeGroup();
while (...) {
    group.add(node);
}
```

A node group has a similar functionality to a vector. You can ask for the size and elements of the group, remove elements from the group, or check whether a node already belongs to the group. You can also convert a vector of nodes into a group:

`group.add(node)`	Adds a node to the group.
`group.remove(node)`	Removes a node from the group.
`group.contains(node)`	Checks whether a node is in the group.
`group.size()`	Returns the number of nodes in the group.
`group.elements()`	Returns the nodes of the group as an Enumeration.
`group = new IlvNodeGroup(vector)`	Creates a new group that contains the nodes stored in the input vector.

Level range constraints (HL)

In Step 1 of the layout algorithm (the leveling phase), the nodes are partitioned into levels. These levels are indexed starting from 0. For instance, when the flow direction is to the bottom, the nodes of the level index 0 are placed at the topmost horizontal level line and the nodes with larger level index are placed at a position lower than the nodes with smaller level index (see *Level and position indices*). The layout algorithm calculates these level indices automatically.

You can choose how the levels are partitioned by specifying the range of the level index for some nodes. The nodes are placed in the levels whose index is in the specified range. You have to specify the minimum and maximum index of the level.

To specify the minimum and maximum index of the level:

In CSS
In the constraint file, specify:

```
LevelRangeConstraint {
  subject: { node11 }
  minLevel: 5
  maxLevel: 7
}
```

This specification forces the graphic of the model node with ID “node11” to be placed between the minimum level 5 and the maximum level 7, that is, either in level 5, level 6, or level 7.

In Java™
Call:

```
layout.addConstraint(new IlvLevelRangeConstraint(node, 5, 7));
```

Notice that in this case, `node` contains the graphic node (subclass of `IlvGraphic`). If you want to place the node exactly at level 5, call:

```
layout.addConstraint(new IlvLevelRangeConstraint(node, 5, 5));
```

Alternatively, you can call:

```
layout.setSpecNodeLevelIndex(node, 5);
```

which has exactly the same meaning.

If you want to force the node to level 5 and above, set `UNSPECIFIED` as the maximal level.

In CSS
In the constraint file, specify:

```
LevelRangeConstraint {
  subject: { node11 }
  minLevel: 5
  maxLevel: UNSPECIFIED
}
```

In Java
Call:

```
layout.addConstraint(
  new IlvLevelRangeConstraint(node, 5, IlvHierarchicalLayout.UNSPECIFIED));
```

If you want to force the node to level 5 and below (that is, level 0, ..., 5), set `UNSPECIFIED` as the minimal level; for example, in Java:

```
layout.addConstraint(
  new IlvLevelRangeConstraint(node, IlvHierarchicalLayout.UNSPECIFIED, 5));
```

In this particular case, you could also use zero (0) as the minimal level because the level indices start at 0.

You can apply the constraint to a group of several nodes at once. This has the same effect as specifying the constraint for each single node of the group, but it is more memory efficient and convenient. For instance, if you want to force the group of three nodes to the levels between 5 and 7:

To specify these parameters:

In CSS
In the constraint file, specify:

```
LevelRangeConstraint {
  subject: { node11, node12, node13 }
  minLevel: 5
  maxLevel: 7
}
```

In Java
Create a `IlvNodeGroup` object (see *Node groups*) of the three nodes and add it to the constraint in the following way:

```
layout.addConstraint(new IlvLevelRangeConstraint(nodeGroup, 5, 7));
```

Level index parameter (HL)

The level index is a special case of a level range constraint (see *Level range constraints (HL)*). It forces the node to one particular level. For your convenience, you can specify the level index of a node directly by the method:

```
void setSpecNodeLevelIndex(Object node, int index)
```

You pass a single node as the first argument (not a node group). The default index value is `-1`. If the default value is used, or if a node is set to a negative level index, the level index is considered to be unspecified. In this case the layout algorithm automatically calculates an appropriate level index during the leveling phase of the algorithm.

To obtain the specified level index for a node, use the method:

```
int getSpecNodeLevelIndex(Object node)
```

However, this method returns the value that was set by `setSpecNodeLevelIndex`. If the level index was specified by allocating a corresponding level range constraint that has the same meaning, `getSpecNodeLevelIndex` still returns `-1`.

Warning: Using arbitrarily large level indices is not recommended. For instance, if you set the level index of a node to `100000`, the layout algorithm creates 100,000 levels even if the graph has far fewer nodes. This causes the layout algorithm to become unnecessarily slow.

Same level constraints (HL)

If you want to force several nodes to the same level with fixed index, you can set the level index parameter of these nodes accordingly (see *Level index parameter (HL)*) or use a level range constraint (see *Level range constraints (HL)*). However, if you want to force several nodes to the same level *without* forcing them to a specific level index, you cannot use these mechanisms. You must use a same level constraint.

To set the same level constraint:

In CSS
In the constraint file, specify:

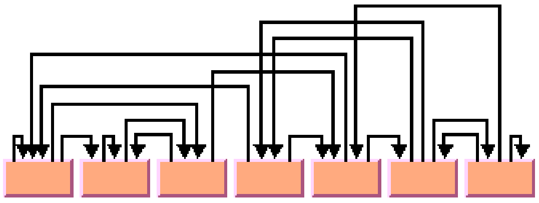
```
SameLevelConstraint {
  firstNode: node1
  secondNode: node2
}
```

In Java™
Call:

```
layout.addConstraint(new IlvSameLevelConstraint(node1, node2));
```

This forces `node1` and `node2` to be placed into the same level, but it does not constrain them to any particular level.

The following figure illustrates the placement of nodes on the same level.



All Nodes Fixed at Same Level

Group spread constraints (HL)

An alternative way to force a group of nodes to the same level is by specifying a group spread constraint with a spread size of zero (0). In general, the group spread constraint forces a group of nodes to *k+1* subsequent levels. The number *k* is the spread size. It does not select the lowest or highest level index of the group, but only requires that the nodes be placed no more than *k* levels apart. Hence, if *k=0*, all nodes of the group are placed at the same level.

To illustrate the general group spread constraint on nodes with ID “nodeA’, “nodeB” and “nodeC”:

In CSS
In the constraint file, specify:

```
GroupSpreadConstraint {
  group: { nodeA, nodeB, nodeC }
  spreadSize: 2
}
```

In Java™
To use the group spread constraint on graphic nodes (subclasses of `IlvGraphic`), call:

```
IlvNodeGroup nodeGroup = new IlvNodeGroup();
nodeGroup.add(nodeA);
nodeGroup.add(nodeB);
nodeGroup.add(nodeC);
layout.addConstraint(new IlvGroupSpreadConstraint(nodeGroup, 2));
```

The constraint is satisfied if the highest level index for `nodeA`, `nodeB`, and `nodeC` is no more than two levels apart from the smallest level index of the nodes. For instance, the constraint is satisfied if the level indices for `nodeA`, `nodeB`, and `nodeC` are 1, 2, 3; or if they are 7, 8, 9; or if they are 16, 14, 15. The constraint is also satisfied if all three nodes are placed at level 5, or if two of the nodes are placed at level 15 and the third node at level 13. The constraint is not satisfied if the level indices for `nodeA`, `nodeB`, and `nodeC` are 3, 5, 6, because in this case the highest index (6) is more than two levels away from the lowest index (3).

Relative level constraints (HL)

If the flow direction is towards the bottom, level 0 is topmost in the drawing. In this layout you can specify by relative level constraints that a node be above or below another node. If the flow direction is towards the right, level 0 is leftmost in the drawing. Here you can specify by relative level constraints that a node be left or right of another node.

In CSS
In the constraint file, specify:

```
RelativeLevelConstraint {
  lowerSubject: { nodeA }
  higherSubject: { nodeB }
  priority: 1000.0
}
```

In Java
In Java™ , call:

```
layout.addConstraint(
    new IlvRelativeLevelConstraint(nodeA, nodeB, priority));
```

This forces `nodeA` to be placed at a level with a smaller index than `nodeB`. Since relative level constraints compete with each other, you must specify the priority of the constraint. In fact, links also impose constraints on the system, and the link priority has the same impact as the constraint priority. A link with priority 10 forces its (usually) source node (unless ports are specified) into a lower level than its target node. To force the source node into a higher level than the target node, you need to create a constraint with a higher priority than the link. For instance, to ensure that the constraints are satisfied even if there are many links, you can use link priorities between 0 and 10 and constraint priorities between 1000 and 10,000.

You can also create a relative level constraint between groups of nodes.

In CSS
In the constraint file, specify a comma-separated list of nodes, such as

```
lowerSubject: { node1, node2, node3 }
```

In Java
Call:

```
layout.addConstraint(
    new IlvRelativeLevelConstraint(nodeGroup1, nodeGroup2, priority));
```

Position index parameter (HL)

In Step 2 of the layout algorithm (the crossing reduction phase), the nodes are ordered within the levels. All nodes that belong to the same level get a position index starting from 0. For instance, when the flow direction is to the bottom, the node with the position index 0 is placed in the leftmost position within its level. The nodes with a larger position index are placed farther to the right than the nodes with a smaller position index in the same level. The nodes of different levels are independent. The node of the first level with the position index 0 is to the left of the node of the first level with the position index 1, but not necessarily to the left of a node of another level with position index 0. Note that long links crossing a level also obtain a position index (see *Level and position indices*). The layout algorithm calculates these position indices automatically.

You can affect how the nodes are positioned within each level by specifying the position index of some nodes. The nodes are placed at the specified position within their level.

To specify the position index of a node in Java™ , use the method:

```
void setSpecNodePositionIndex(Object node, int index)
```

The default value is `-1`. If the default value is used, if a node is set to a negative position index, or if a node is set to a position index that is larger than the number of nodes of its level, the layout automatically calculates an appropriate position index during the crossing reduction step.

To obtain the current position index of a node, use the method:

```
int getSpecNodePositionIndex(Object node)
```

Relative position constraints (HL)

Working with absolute node position indices is inconvenient in certain situations. For example, if two nodes belong to the same level, you may want to force one node to a position with a lower index than the other node without fixing the absolute positions of the nodes. You can achieve this by using a relative position constraint.

The relative position constraint forces a specific order upon the nodes of a level, but it does not specify which nodes are directly neighbored. For instance, a relative position constraint may force `nodeA` to be placed somewhere at a lower position than `nodeB`, but there may be many nodes between `nodeA` and `nodeB`.

In CSS
In the constraint file, specify:

```
RelativePositionConstraint {
  lowerSubject: { nodeA }
  higherSubject: { nodeB }
  priority: 1000.0
}
```

In Java™
Call:

```
layout.addConstraint(
    new IlvRelativePositionConstraint(nodeA, nodeB, priority));
```

This forces `nodeA` to a lower position than `nodeB`. If the flow direction is towards the bottom, the nodes are in horizontal levels; hence the constraint means that `nodeA` is placed at the left side of `nodeB`. If the flow direction is towards the right, the nodes are in vertical levels; hence the constraint means that `nodeA` is placed below `nodeB`.

The relative position constraint has an effect only if both nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition. There is no way to influence the relative position of nodes that belong to different levels.

Similar to the relative level constraint, the relative position constraint can be applied to node groups. These constraints also have priorities that indicate which constraints dominate if a constraint conflict occurs. The higher the priority, the more likely the constraint is satisfied when resolving constraint conflicts.

Side-by-side constraints (HL)

To force nodes to be directly neighbored, use the side-by-side constraint.

In CSS
In the constraint file, specify:

```
SideBySideConstraint {
  group: { nodeA, nodeB, nodeC }
  priority: 100.0
}
```

In Java™
You can create a side-by-side constraint on a group of type `IlvNodeGroup` (see *Node groups*):

```
layout.addConstraint(
    new IlvSideBySideConstraint(nodeGroup, priority));
```

If the node group consists of just two nodes, it forces the two nodes to be placed side by side. However, it does not specify which node is at the lower node position and which node is at the higher node position. If the group consists of more than two nodes, it forces the nodes to be placed at consecutive positions such that all nodes are clustered together. A node that does not belong to the group cannot be placed between the nodes of the group.

For example, assume that the group contains the three nodes `A`, `B`, `C`. The constraint is satisfied if the position indices of `A`, `B`, and `C` are 3, 4, 5 or 9, 7, 8. However, if node `D` is placed between `A` and `B` (say, `D` has position 4, `A` has position 3, and `C` has position 5), then the constraint is not satisfied because `D` does not belong to the same group.

The side-by-side constraint has an effect only if the nodes actually belong to the same level. To achieve this, you can, for instance, use a same level constraint in addition.

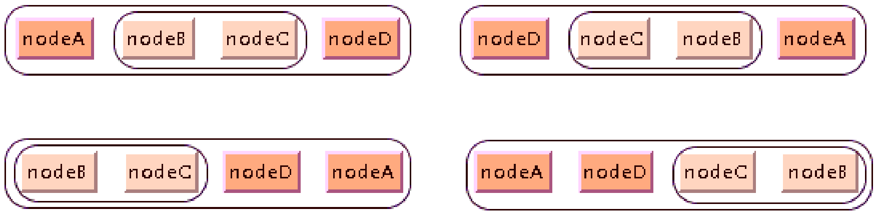
Side-by-side constraints have priorities that decide how to resolve constraint conflicts. The higher the priority, the more likely the constraint is satisfied.

You can use side-by-side constraints to create nested clusters. For example, in Java:

```
IlvNodeGroup group1 = new IlvNodeGroup();
group1.add(nodeA);
group1.add(nodeB);
group1.add(nodeC);
group1.add(nodeD);
layout.addConstraint(
    new IlvSideBySideConstraint(group1, 10.0f));
IlvNodeGroup group2 = new IlvNodeGroup();
group2.add(nodeB);
group2.add(nodeC);
layout.addConstraint(
    new IlvSideBySideConstraint(group2, 10.0f));
```

The first constraint specifies that `nodeA`, `nodeB`, `nodeC`, and `nodeD` must be clustered. The second constraint specifies that `nodeB` and `nodeC` are clustered inside the larger cluster.

This means that no other node can be placed between the four nodes and, furthermore, neither `nodeA` nor `nodeD` can be placed between `nodeB` and `nodeC`. The following figure shows four solutions that satisfy both constraints.



Sketch of Solutions for Side-By-Side Constraints

Extremity constraints (HL)

To force a node to the first level, you can specify:

```
layout.setSpecNodeLevelIndex(node, 0);
```

However, you cannot specify a level index for the last level because it is unknown at the beginning of layout how many levels will be created. It is unwise to specify:

```
layout.setSpecNodeLevelIndex(node, java.lang.Integer.MAX_VALUE);
```

because this will create many empty levels between the levels actually used and the last one. Even though these empty levels are removed in postprocessing steps, this influences the speed and quality of the layout. (In fact, the algorithm will run out of memory if you set the specified level index unreasonably high.)

By using constraints you can achieve the same effect more efficiently.

To force a node to the first level:

In CSS
In the constraint file, specify:

```
ExtremityConstraint {
  node: node
  side: NORTH
}
```

In Java
In Java™ , call:

```
layout.addConstraint(
    new IlvExtremityConstraint(node, IlvHierarchicalLayout.NORTH));
```

To force a node to the last level:

In CSS
In the constraint file, specify:

```
ExtremityConstraint {
  node: node
  side: SOUTH
}
```

In Java
Call:

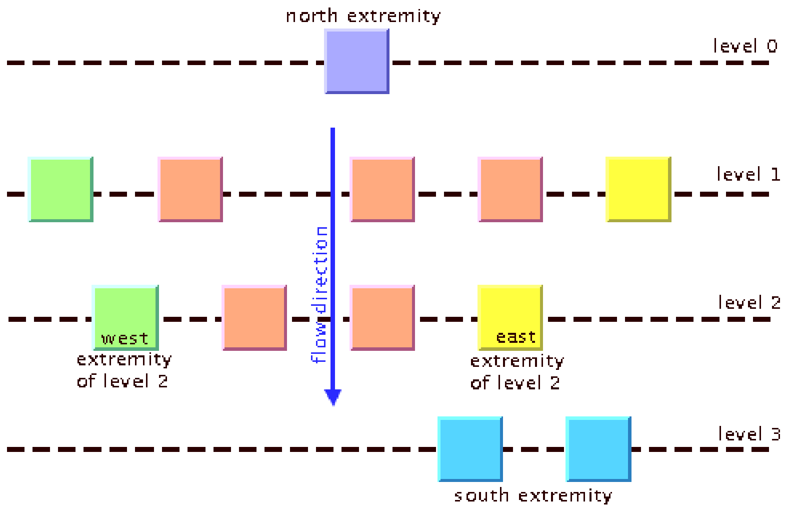
```
layout.addConstraint(
    new IlvExtremityConstraint(node, IlvHierarchicalLayout.SOUTH));
```

With compass directions as a convenient reference (see *Port sides parameter (HL)*), the first level indicates the north pole and the last level indicates the south pole. You can also specify extremity constraints for the east and west sides:

```
layout.addConstraint(
    new IlvExtremityConstraint(node1, IlvHierarchicalLayout.EAST));
layout.addConstraint(
    new IlvExtremityConstraint(node2, IlvHierarchicalLayout.WEST));
```

The west extremity constraint forces the node to the lowest position index within its level, and the east extremity constraint forces the node to the highest position index within its level. The position indices specify the relative position within the level. For instance, a node with west extremity constraint will be the leftmost node within its level, if the flow direction is towards the bottom. However, this does not affect other levels; there may be a node in another level that is still placed farther to the left.

The following figure illustrates some extremity constraints.



Sketch of Extremity Constraints

Swim lane constraints (HL)

Swim lanes are rectangular areas orthogonal to the levels.

- ♦ If the link flow direction is towards the bottom or top, the levels are horizontal rows and the swim lanes are vertical columns.
- ♦ If the flow direction is towards the left or right, the levels are vertical columns and the swim lanes are horizontal rows.

Swim lanes can be used if the nodes are partitioned into groups, to indicate which nodes belong to a certain group. The nodes of the same swim lane are placed so that it is possible to draw a surrounding rectangle around them. Swim lanes allow you to organize the graph in a table-like manner. For instance, you may have a workflow diagram where nodes represent actions; then the swim lanes could represent the departments that perform these actions. Each node can belong to only one swim lane.

To associate a group of nodes with the same swim lane:

In CSS
In the constraint file, specify:

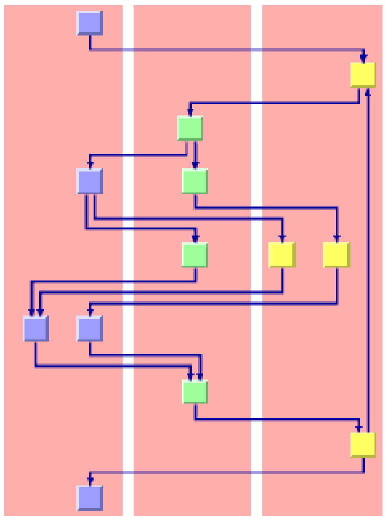
```
SwimLaneConstraint {
  group: { node1, node2, node3 }
  relativeSize: -1.0
  positionIndex: -1
  minMargin: 0.0
}
```

In Java
In Java™ , call:

```
layout.addConstraint(new IlvSwimLaneConstraint(new IlvNodeGroup(nodeVector)))
;
```

All nodes of the node vector will be placed in the same swim lane rectangle. If a graph has many swim lane rectangles, the relative order of these swim lanes is determined automatically. The size of the swim lane rectangle depends on the nodes that belong to the swim lane. However, you can specify the relative order, relative size, and the margins of the swim lane as well by using the constructor:

```
public IlvSwimLaneConstraint(IlvNodeGroup group,
                             float relativeSize,
                             int positionIndex,
                             float minMargin)
```



The red background rectangle indicates where the swim lane is.

Swim Lanes

The relative size indicates how large this swim lane is compared to the other swim lanes. Assume that the flow direction is towards the bottom. In this case, the relative size indicates the width of the swim lane. All swim lanes with the same relative size will have the same width. A swim lane with a relative size that is twice the value of another swim lane will have twice the width of the other swim lane. The actual number of this parameter does not matter, only how large the value is compared to the other swim lanes. If you do not want to restrict the size of the swim lane, set the value to 0. In this case, the width of the swim lane will be independent of the other swim lanes.

The minimal margin is the margin of the swim lane in absolute coordinates. If the flow direction is towards the bottom, then it is the minimal horizontal distance between the leftmost or rightmost node of the swim lane and the swim lane border.

The position index indicates the order of the swim lanes. Just as nodes have position indices, the swim lanes are placed sequentially at relative positions numbered from 0 to n. In a top-down layout, the swim lane with position 0 is the leftmost swim lane, and the swim lanes with higher position indices are placed farther to the right. If the swim lanes have the position index -1, the layout algorithm determines the appropriate position automatically.

A swim lane constraint is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.

Tip: The automatic conflict resolution can handle conflicting constraints. However, to speed up the layout, it is recommended that you specify constraints in such a way that there are no conflicts.

Summary of constraints file as opposed to constraints in Java (HL)

The following is a summary of how to specify constraints:

- ♦ Applications based on the class `IlvDiagrammer` which use style sheets (CSS files) can specify constraints in a constraint file.
- ♦ Applications that do not use style sheets but work on `IlvGrapher` and `IlvHierarchicalLayout` instances can add constraints directly to the layout instance in Java™ .
- ♦ In the constraint file, nodes are specified by their model ID (see `getID(java.lang.Object)`).
- ♦ Constraints specified in Java take the graphic nodes (subclasses of `IlvGraphic`) directly as arguments.
- ♦ Relative level constraints, relative position constraints, level range constraints, group spread constraints, side-by-side constraints, and swim lane constraint can work in node groups. To specify a node group:
  - In the constraint file, use a comma separated list of node IDs:

    { node1, node2, node3 }
  - In Java, use the API of the class `IlvNodeGroup`:

    ```
    IlvNodeGroup group1 = new IlvNodeGroup();
    group1.add(node1);
    group1.add(node2);
    group1.add(node3);
    ```

Warning: Mixing style sheets and constraint specifications in Java is not recommended. In an application that uses style sheets (CSS files), the constraints specified in the CSS file override those created in Java, when the layout is performed.

Constraint priorities (HL)

A set of constraints may cause conflicts. This means that not all of the constraints can be satisfied at the same time. For instance, it is impossible to force two nodes into the same level by an `IlvSameLevelConstraint` while at the same time forcing one of the nodes to a higher level than the other node by an `IlvRelativeLevelConstraint`. In this case, one of the two constraints must be ignored during layout.

In general, constraint conflicts are resolved by ignoring the constraints with the lowest priority while the constraints with the highest priority get satisfied. The following rules explain the constraint priorities in detail.

- The constraints that influence into which level a node is placed are applied before the constraints that influence the position of the node within a level.
- The `IlvExtremityConstraint` is translated into a sequence of constraints with high priority. For instance, the extremity constraint with the south side is translated into several same level constraints and several relative level constraints.
- The `IlvSameLevelConstraint` and the `IlvGroupSpreadConstraint` have the highest priority. They are never in conflict with each other. They dominate all other constraints, even the specified level index.
- The `IlvLevelRangeConstraint` (and the direct level index specification) has the second highest priority. If two nodes are forced to the same level but have disjoint specified level ranges, then the level range is ignored. In the following example:

  ```
  layout.addConstraint(new IlvSameLevelConstraint(node1, node2));
  layout.setSpecNodeLevelIndex(node1, 5);
  layout.setSpecNodeLevelIndex(node2, 10);
  ```

  both `node1` and `node2` will be placed at level 5. The conflicting specification: layout.setSpecNodeLevelIndex(node2, 10) is ignored.
- The `IlvRelativeLevelConstraint` is dominated by the same level constraint, by the level range constraint, and by the direct specification of level indices. If several relative level constraints conflict each other, the one with the highest specified priority dominates. However, note that all links are implicitly considered relative level constraints as well. If links with high priority force a node to a certain level, then a relative level constraint with lower priority will be ignored.
- The `IlvSwimLaneConstraint` is always evaluated, even if the incremental mode is enabled. The constraint has a higher priority than the relative position constraint and the side-by-side constraint. You can specify side-by-side constraints for a group of nodes that belong to the same swim lane, but side-by-side constraints of nodes of different swim lanes are ignored. You can specify relative position constraints between nodes of the same swim lane. You can also specify relative position constraints between one entire swim lane group and another swim lane group, which effectively orders the swim lanes. But relative position constraints are ignored if they would require breaking the swim lanes apart. The swim lane constraint dominates the specified position indices and the extremity constraints, that is, if a swim lane constraint is used, you cannot specify position indices or east/west extremity constraints for any node.

- The `IlvSideBySideConstraint` is evaluated only if the corresponding nodes belong to the same level. Typically you will use a same level constraint to force the nodes to the same level, and then a side-by-side constraint to force the nodes to a certain ordering. The side-by-side constraints dominate the relative position constraints. If several side-by-side constraints are conflicting, the one with the highest specified priority dominates the other constraints.
- The `IlvRelativePositionConstraint` is also evaluated only if the corresponding nodes belong to the same level. It is dominated by the side-by-side constraint; however, conflicts with side-by-side constraints are rare. If several relative position constraints are conflicting, the one with the highest specified priority dominates the other constraints.

For experts: constraint validation (HL)

Constraints that you specify in Java™ may become invalid. For instance, if you add a constraint that node `A` must be to the left side of node `B`, but you remove `A` from the graph, then this constraint becomes invalid. It simply does not make sense any more, even though it does not conflict with any other constraint. The layout instance automatically removes invalid constraints from time to time because they are a waste of memory. The validation check is done during layout. Forcing a validation check is normally not necessary but if you want to do this, call:

```
layout.validateConstraints();
```

This removes all invalid constraints from the Hierarchical Layout and cleans up the memory. The constraint validation does not check which constraints have conflicts. The main effect of the validation is that the constraint system uses less memory afterwards.

Note: A constraint is valid if it is meaningful. Two valid constraints are conflicting if the system cannot satisfy them both at the same time. Invalid constraints cannot be conflicting because they are meaningless.

Hence, constraint validation and constraint resolution are different phases. Constraint validation performs a quick local test. It removes invalid constraints from the layout instance completely. It does not affect conflicting constraints.

Constraint resolution checks whether a set of valid constraints are in conflict with each other. Thus, constraint resolution is a complex process on a network of multiple related constraints. Constraint resolution decides which constraints can be solved and which cannot. But the constraint resolution does not remove conflicting constraints from the layout instance, it just delivers a solution that may ignore some constraints.

For experts: specifying constraints in CSS directly (HL)

SDM applications that use style files (CSS files) can specify an external constraint file that refers to the constrained nodes by their node IDs. Alternatively, it is possible to specify the constraints in the CSS file directly.

The general mechanism for specifying constraints in CSS directly is:

```
<node selector> {
   <constraint property>: "<constraint name>,<additional parameters>";
}
```

The meaning is that the selected node participates in the constraint with given name. The constraint name is an arbitrary string that identifies the constraint. The name is needed because usually, several nodes participate at the same constraint, therefore several such style rules are required to entirely specify the constraint.

For instance, to specify a relative level constraint between the nodes with ID `activity1` and `participant1`, use the following style rules:

```
#activity1 {
   LowerRelativeLevelConstraint: "ConstraintA,5000";
}

#participant1 {
   HigherRelativeLevelConstraint: "ConstraintA,5000";
}
```

The constraint property `LowerRelativeLevelConstraing` indicates that this rule specifies the lower node of a relative level constraint. The property `HigherRelativeLevelConstraint` indicates that this rule specifies the higher node of a relative level constraint. The constraint name “`ConstraintA`” is used to distinguish this relative level constraint from other constraints of the same type. The parameter 5000 is the priority of this constraint.

All constraint types can be specified similarly. The constraint name is arbitrary, except for the extremity constraint: Here, only the names “`EAST`”, “`WEST`”, “`NORTH`” and “`SOUTH`” are allowed as constraint names, and they indicate at the same time the side of the constraint.

A sample CSS sheet with various constraint specifications can be found at:

<installdir>/jviews-diagrammer86/codefragments/graphlayout/hierarchicallayout/constraints/resources/SampleCSS.css.

For further details about specifying constraints in Java™ , see the class `IlvGraphLayoutRenderer`.

The following table lists all constraint properties that are available in CSS.

CSS Constraint properties

Constraint Property	Corresponding Specification in Constraint File
LevelRangeConstraint	LevelRangeConstraint { subject: {...} }
FirstSameLevelConstraint	SameLevelConstraint { firstNode: ... }
SecondSameLevelConstraint	SameLevelConstraint { secondNode: ... }
GroupSpreadConstraint	GroupSpreadConstraint { group: {...} }
LowerRelativeLevelConstraint	RelativeLevelConstraint { lowerSubject: {...} }
HigherRelativeLevelConstraint	RelativeLevelConstraint { higherSubject: {...} }
LowerRelativePositionConstraint	RelativePositionConstraint { lowerSubject: {...} }
HigherRelativePositionConstraint	RelativePositionConstraint { higherSubject: {...} }
SideBySideConstraint	SideBySideConstraint { group: {...} }
ExtremityConstraint	ExtremityConstraint { node: ... }
SwimLaneConstraint	SwimLaneConstraint { group: {...} }

Note: If constraints are specified in the CSS file directly, each node can participate only in one constraint of each type at most. For instance, you can specify an extremity constraint “EAST” for a node, but you cannot specify an extremity constraint “SOUTH” for the same node at the same time. You can however specify another constraint type (e.g. a level range constraint) for the same node at the same time.

This limitation is due to technical reasons pertaining to the mechanism whereby CSS files are processed by the SDM engine. If you specify constraints by an external constraint file, there is no such limitation: for instance., in the external constraint file, you can specify as many extremity constraints for the same node as you want.

For experts: more indices (HL)

The Hierarchical Layout allows you to specify the level index and the position index of a node.

In CSS
Specify the level and position index of a node with ID “`node1`” in the following way:

```
#node1 {
   SpecNodeLevelIndex: "5";
   SpecNodePositionIndex: "33";
}
```

In Java™
You specify the level and position index of a graphic node in the following way:

```
layout.setSpecNodeLevelIndex(node, 5);
layout.setSpecNodePositionIndex(node, 33);
```

How these indices are used depends on the graph topology and the additional constraints. For example, the specified level index can be in conflict with some `IlvLevelRangeConstraint` or `IlvSameLevelConstraint`. In this case, the constraint priorities determine how the conflict is resolved (see *Constraint priorities (HL)*). If the incremental mode is switched on, the specified node level and position index are ignored, since the incremental mode tries to preserve old node positions. It is also possible to obtain the indices of nodes that were calculated during layout.

Calculated level index

The layout algorithm allows you to access the level index that was calculated for a node by a previous layout. To do this, use the method:

```
int getCalcNodeLevelIndex(Object node)
```

If the node was never laid out, this method returns `-1`. Otherwise, it returns the previous level index of the node.

In an application that specifies layout parameters entirely in Java, the method can be used to specify the level index for the next layout in the following way:

```
int index = layout.getCalcNodeLevelIndex(node);
layout.setSpecNodeLevelIndex(node, index);
```

When this is done, it ensures that the node is placed at the same level as in the previous layout.

If the graph is detached from the layout algorithm, the calculated level index of a node is set back to `-1`.

Note: You should be aware of the difference between the methods `getCalcNodeLevelIndex(java.lang.Object)` and `getSpecNodeLevelIndex(java.lang.Object)`. The first one returns the level index calculated by the previous layout. The second one returns the specified level index, even if there was no previous layout.

For instance, consider two nodes A and B. Node A has no specified level index and node B has a specified level index 5. Before the first layout, the method `getCalcNodeLevelIndex` returns -1 for both nodes because the levels have not been calculated yet. However, `getSpecNodeLevelIndex` returns -1 for A and 5 for B. After the first layout, node A may be placed at level 4. Now, `getCalcNodeLevelIndex` returns 4 for node A and 5 for node B and `getSpecNodeLevelIndex` still returns -1 for A and 5 for B.

Calculated position index

The layout algorithm allows you to access the position index within a level that was calculated for a node by a previous layout. To do this, use the method:

`getCalcNodePositionIndex(java.lang.Object)`

If the node was never laid out, this method returns -1. Otherwise, it returns the previous position index of the node within its level.

To ensure that the node is placed at the same level at the same relative position as in the previous layout, use in an application that specifies layout parameters entirely in Java the following:

```
layout.setSpecNodeLevelIndex(node,
    layout.getCalcNodeLevelIndex(node));
```

This example code works only if the generic connected component layout is disabled and the port sides `EAST` or `WEST` are not used in the layout.

If the graph is detached from the layout algorithm, the calculated position index of a node is set back to -1.

Note: You should be aware of the difference between the methods `getCalcNodePositionIndex` and `setSpecNodePositionIndex`.The first one returns the position index calculated by the previous layout and -1 if there was no previous layout. The second one returns the specified position index even if there was no previous layout. This behavior is similar to the behavior of the specified and calculated level index (see *Calculated level index*).

Recursive layout

JViews Diagrammer supports nested graphs, that is, it can render graphs containing nodes that are graphs. A graph that is a node in another graph is called a subgraph. Links that connect nodes of different subgraphs are called intergraph links. In *Recursive hierarchical layout on nested graph with polyline link style*, all red links are intergraph links and all black links are normal links. This is explained in detail in *Nested layouts*.

The hierarchical layout can treat a nested graph at once, placing all nested nodes and routing all links including the intergraph links. It can even place the labels in the nested graph.

To enable the recursive mode:

In CSS
Add to the `GraphLayout` section:

```
recursiveLayoutMode: "true";
```

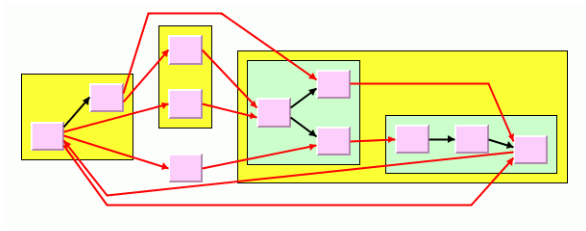
In Java™
Use this method:

```
void setRecursiveLayoutMode(boolean enable);
```

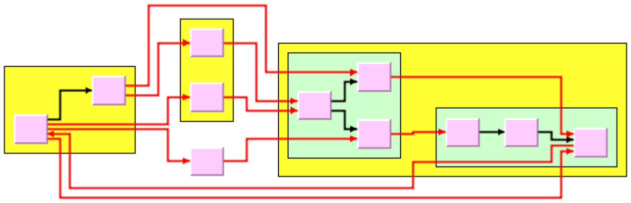
and call `performLayout` with the third parameter set to `true` in the following way:

```
layout.performLayout(force, redraw, true);
```

The recursive layout mode requires that all subgraphs are laid out in the same style (for example, they must all use the same flow direction). This is automatically the case when calling `layout.performLayout(force, redraw, true)` and it is also the case when using CSS without specifying individual graph layouts per subgraph. If different layout styles are needed per subgraph, you must specify an individual layout per subgraph as described in *Individual layout styles per subgraph* and in *Advanced recursion: mixing different layouts in a nested graph*. In this case the hierarchical layout cannot route the intergraph links and you have to use a Link Layout algorithm to route the intergraph links.



Recursive hierarchical layout on nested graph with polyline link style



Recursive hierarchical layout on nested graph with orthogonal link style

Setting layout parameters in recursive mode in Java code

When you use CSS, the layout parameters are specified in CSS as usual, and the SDM engine handles the internal details automatically. However, when you want to specify layout parameters in Java™ code, note that in recursive layout mode, the hierarchical layout is attached to the top-level graph. Global layout parameters must be set on this layout instance. Layout parameters per node or per link must be set in the following way:

```
// node is directly contained in subgraph
IlvHierarchicalLayout sublayout = (IlvHierarchicalLayout)topLevelLayout.
getRecursiveLayout().getLayout(subgraph);
sublayout.setSpecNodeLevelIndex(node, 5);
```

This means that layout parameters per node or per link cannot be set on the top-level layout, but on a sublayout retrieved by means of the `IlvRecursiveLayout` from the top-level layout.

Label layout

If the recursive layout mode is enabled, the hierarchical layout can also place the node and link labels. This is useful, because placing labels after a recursive layout may change the bounds of subgraphs again, and hence would require the hierarchical layout to rerun. Therefore, an annealing label layout is integrated into the hierarchical layout which is executed during the recursive layout mode. In order to set label descriptors, you can access this label layout by using the following code:

```
public IlvAnnealingLabelLayout getLabelLayout()
```

If the labels are contained in a subgraph, use the following code:

```
// node and label are directly contained in subgraph
  IlvHierarchicalLayout sublayout =
        (IlvHierarchicalLayout)topLevelLayout.getRecursiveLayout().getLayout
(subgraph);
  IlvAnnealingLabelLayout labellayout = sublayout.getLabelLayout();
  lvAnnealingPointLabelDescriptor d =
    new IlvAnnealingPointLabelDescriptor(label, node,
IlvAnnealingPointLabelDescriptor.RECTANGULAR,
                                         IlvDirection.Bottom);
  labellayout.setLabelDescriptor(label, d);
```

When the recursive layout mode is used, the label layout is automatically used. It is recommended to keep it enabled if nodes or links in subgraphs have labels. In can be disabled if there are no labels.

To disable the label layout:

In CSS

```
labelLayoutEnabledDuringRecursiveLayoutMode: "false";
```

In Java

```
layout.setLabelLayoutEnabledDuringRecursiveLayoutMode(false);
```

For more details on how to use the `IlvAnnealingLabelLayout` see *Annealing label layout*. For more details on how to use the `IlvRecursiveLayout` see *Recursive layout*.

Link layout (LL)

Describes the *Link Layout* algorithm (class `IlvLinkLayout` from the package `ilog.views.graphlayout.link`).

In this section

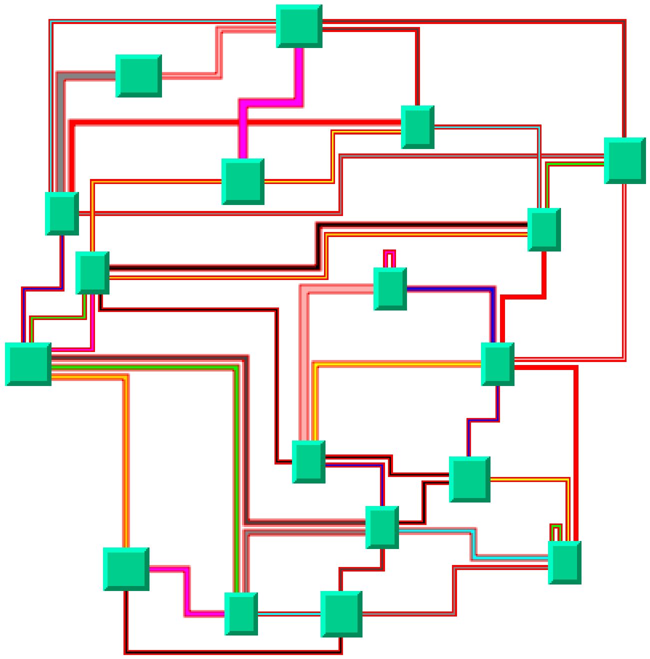
For experts: special options of the Short LL
Describes the options of the Short Link Layout for expert use.

For experts: special options of the Long LL
Describes the options of the Long Link Layout for expert use.

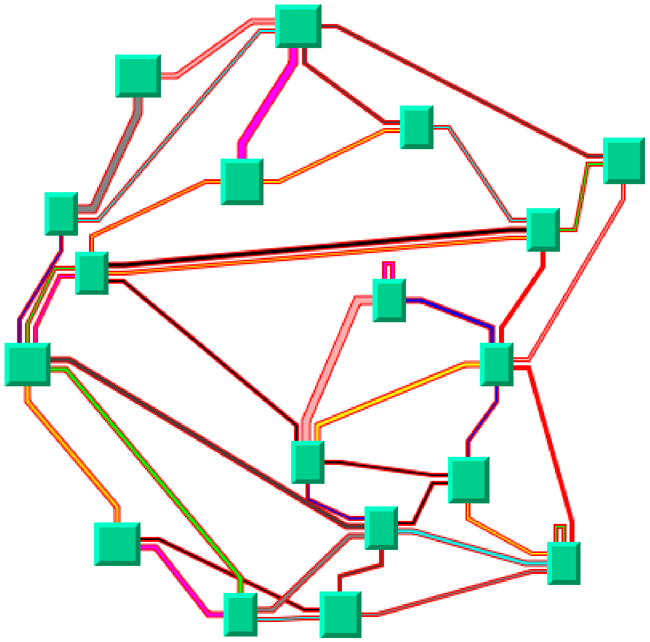
General information on the LL

LL samples

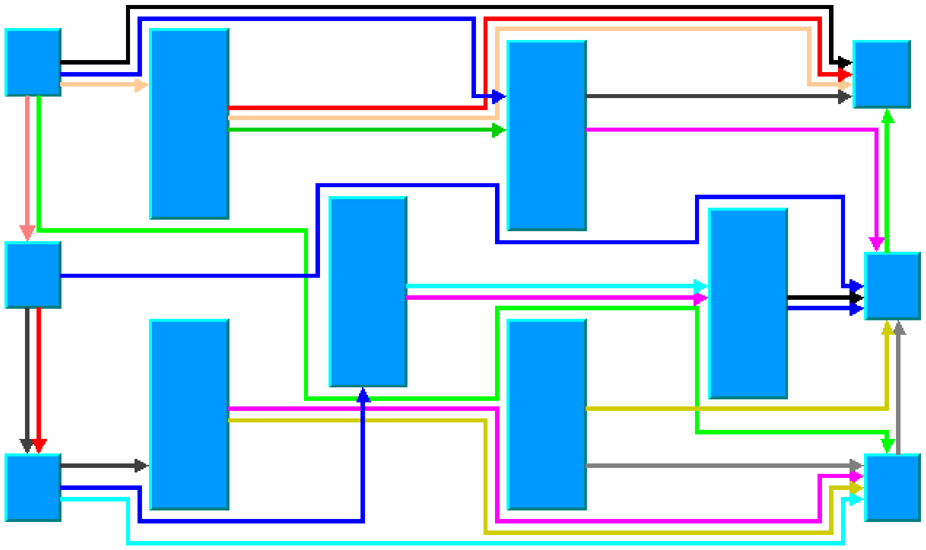
These sample drawings were produced with the Link Layout algorithm:



Link layout in short link mode with orthogonal links



The same graph in short link mode with direct links



Link layout in long link mode with orthogonal links

What types of graphs suit the LL?

Any type of graph where nodes are fixed and links need to be routed:

- connected graphs and disconnected graphs
- planar graphs and nonplanar graphs.
- nested graphs with intergraph links

Application domains for the LL

Application domains of the Link Layout include:

- Electrical engineering (circuit block diagrams)
- Industrial engineering (schematic design diagrams, equipment/resource control charts)
- Business processing (entity relation diagrams)
- Software management/software (re-)engineering (data inspector diagrams)
- Database and knowledge engineering (sociology, genealogy)
- CASE tools (design diagrams)

Features and limitations of the LL

Features of both modes (LL)

♦ Reshapes the links of a graph in either an orthogonal or a direct style, without moving the nodes. Orthogonal and direct style links can be combined in the same layout.

♦ Allows you to specify which side of the node (top, bottom, left, or right) a link can be connected to, or to preserve the existing connection points of the links.

♦ Supports self-links (that is, links with the same origin and destination node).

♦ Supports multiple links (that is, more than one link between the same origin and destination nodes).

♦ Allows you to specify pinned (fixed) links that the layout algorithm cannot reshape.

♦ Supports intergraph links of nested graphs. An intergraph link is a link whose end nodes belong to different subgraphs of a nested graph.

♦ Supports an incremental mode: If new links are added to a drawing, the next layout takes the shapes of the old links into account.

♦ Two layout modes: *short links* with a limited number of bends or *long links* with unlimited number of bends.

Features of short link mode (LL)

♦ Links are placed freely in the space.

♦ Link-to-link and link-to-node crossings are reduced, if this is possible with link shapes that have a maximum of 4 bends.

♦ Links of different width are supported.

♦ Link bundles between the same pair of nodes are supported. Optionally, the algorithm can ensure that multiple links are bundled together by giving them parallel shapes.

♦ Automatically arranges the final segments of the links (the segments near the origin or destination node) to obtain a bundle of parallel links.

♦ Provides two optional shapes for the self-links.

♦ Very fast algorithm with low memory footprint.

Features of long link mode (LL)

♦ Links are placed on a grid.

♦ Link-to-node crossings of orthogonal links are avoided, even if this introduces many bends.

♦ Orthogonal link segments do not overlap.

- Does not bundle the final segments. Instead, it distributes the links on the border of each end node according to which border has more free space.
- Fast algorithm: speed and memory footprint depend on the grid spacing.

Limitations

- Since the Link Layout algorithm reshapes the links, it works preferably with links of type `IlvPolylineLinkImage` and link connectors of type `IlvFreeLinkConnector`.
- When routing intergraph links, the incremental mode cannot be used. Due to the complexity of intergraph link routing, more crossings and overlappings may occur than when routing normal links.
- In short link mode, crossings and overlapping of links with other links and nodes cannot always be avoided because the algorithm uses link shapes with a limited number of bends. This happens in particular when there are many obstacles between the end points of a link.
- In long link mode, link crossings cannot always be avoided. Segment overlappings of orthogonal links are always avoided unless there is no free space remaining on the border of the end nodes. Any overlapping of nodes and links is always avoided unless one end nodes is inside an enclave. An enclave is an area that is surrounded by other nodes such that the area cannot be reached from the other end node. (See *A Node inside an enclave*.)
- In long link mode, segment overlapping or overlapping between nodes and links cannot always be avoided if the direct link style is used.
- The long link mode is slower and uses more memory if the grid spacing is very small.

The LL algorithms

The Link Layout algorithm utilizes two sublayout classes:

- `IlvShortLinkLayout` in short link mode.
- `IlvLongLinkLayout` in long link mode.

They implement different strategies to find the link shapes.

Short Link Layout algorithm

The Short Link Layout algorithm is based on a combinatorial optimization that chooses the "optimal" shape of the links to minimize a cost function. This cost function is proportional to the number of link-to-link and link-to-node crossings.

For efficiency reasons, the basic shape of each link is chosen from a set of predefined shapes. These shapes are different for each link style option. For the orthogonal link style, the links are reshaped to a polygonal line of up to five alternating horizontal and vertical segments (see *Link layout in short link mode with orthogonal links*). For the direct link style, the links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with small horizontal or vertical segments (see *The same graph in short link mode with direct links*).

The shape of a link also depends on the relative position of the origin and destination nodes. For instance, when two nodes are very close or they overlap, the shape of the link is chosen to provide the best visibility of the link.

The exact shape of a link is computed taking into account additional constraints. The layout algorithm tries to do the following:

- Minimize the number of crossings between the links incident to a given side of a node.
- Space the final segments of the links incident to a given side of a node equally on the node border.

Long Link Layout algorithm

The Long Link Layout algorithm first treats each link individually. For each link, it first calculates the connection points at the end nodes that are on the grid and orders them according to a penalty value. Connection points on used grid points have a very high penalty and, therefore, are very unlikely to be used.

For the orthogonal links (see *Link layout in long link mode with orthogonal links*), the Long Link Layout algorithm then uses a grid traversal to search a route over free grid points from the start connection point to the end connection point. Therefore, in contrast to the short link mode, orthogonal links can have any shape with a large number of bends if this is necessary to bypass obstacle nodes to avoid overlappings. For the direct links (see *The same graph in short link mode with direct links*), it shortens the search by using a direct segment between the connection points.

After all links are placed, a crossing reduction phase examines pairs of links and eliminates link crossings by exchanging parts of the routes between both links.

The Long Link Layout algorithm relies on the fact that links fit to the grid spacing and parts of the routes between different links can be exchanged. Therefore, the Long Link Layout algorithm does not take the link width into account because it would be too difficult to find the parts of two links that can be exchanged. It is recommended to set the grid spacing larger than the largest link width.

Example of Link Layout

In CSS

Since the Link Layout only reshapes the links without placing the nodes, an additional layout algorithm can be specified in CSS for placing the nodes. The specification can be loaded as style file into an application that uses the `IlvDiagrammer` class (see *Graph layout in IBM® ILOG® JViews Diagrammer*).

The following example performs only the Link Layout, without using an additional layout for placing the nodes.

```
SDM {
    GraphLayout : "false";
    LinkLayout  : "true";
}

LinkLayout {
    layoutMode           : "SHORT_LINKS";
    globalLinkStyle      : "ORTHOGONAL_STYLE";
    globalConnectorStyle : "CLIPPED_PINS";
    linkOffset           : "3";
}
```

In some situations, you may need a separate node layout renderer. The following example uses the Uniform Length Edges Layout to place the nodes and the Link Layout to reshape the links:

```
SDM {
    GraphLayout          : "true";
    LinkLayout           : "true";
}

GraphLayout {
    graphLayout          : "UniformLengthEdges";
    linkStyle            : "NO_RESHAPE_STYLE";
    preferredLinksLength : "60";
}

LinkLayout {
    layoutMode           : "SHORT_LINKS";
    globalLinkStyle      : "ORTHOGONAL_STYLE";
    globalConnectorStyle : "CLIPPED_PINS";
    linkOffset           : "3";
}
```

Notice the Link Layout renderer has a “Hierarchical Link Layout” mode. The Hierarchical Link Layout is not a separate layout algorithm. It is merely the feature of the Diagrammer

Link Layout renderer to reuse the Hierarchical Layout as Link Layout, instead of the standard Link Layout. The following example shows how to activate this mode:

```
LinkLayout {
    hierarchical        : "true";
}
```

In Java™

Below is a code sample using the `IlvLinkLayout` class. This code sample shows how to perform a Link Layout on a grapher directly without using a diagram component or any style sheet:

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.link.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /* Fill in the grapher with nodes and links here */

IlvLinkLayout layout = new IlvLinkLayout();
layout.attach(grapher);

/* Specify the layout mode */
layout.setLayoutMode(IlvLinkLayout.SHORT_LINKS);

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic features and parameters of the LL

The `IlvLinkLayout` class supports the following generic parameters as defined in the class `IlvGraphLayout` (see *Base class parameters and features*):

- *Allowed time (LL)*
- *Animation (LL)*
- *Automatic layout (LL)*
- *Preserve fixed links (LL)*
- *Spline routing (LL)*
- *Stop immediately (LL)*
- *Save parameters to named properties (LL)*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed time (LL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) If the layout stops early because the allowed time has elapsed, some links may not be routed in the best possible way. The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Animation (LL)

The layout algorithm can show the temporary positions of the links during the routing in an animated way. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Animation*.)

Note: If this option is enabled, the layout of large graphs can be very time consuming.

Automatic layout (LL)

The Link Layout routes the links so that they bypass the nodes and cross each other as few times as possible. It does not position any nodes. However, if the user moves, adds, or resizes nodes, or adds or removes links, the Link Layout drawing usually becomes invalid; that is, the links no longer look orthogonal, overlap the moved nodes, or cross other links.

Using the automatic layout feature of the `IlvGraphLayout` class, the layout is reperformed whenever a change of the graph occurs. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Automatic layout*.)

Preserve fixed links (LL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.) The fixed links are taken into account when computing the optimal layout of the nonfixed links.

Spline routing (LL)

The layout algorithm supports the generic spline routing mechanism (see *Spline routing*). If the style of a link is direct or orthogonal and the link is a spline, it is routed by the generic spline routing mechanism when it is enabled.

Save parameters to named properties (LL)

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*.)

Stop immediately (LL)

The layout algorithm stops if the method `IlvLinkLayout` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early, some links may not be routed in the best possible way. The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters for both LL modes

Layout mode (LL)

The Link Layout algorithm has two layout modes.

To select a layout mode:

In CSS
Add to the `LinkLayout` section:

```
layoutMode : "SHORT_LINKS";
```

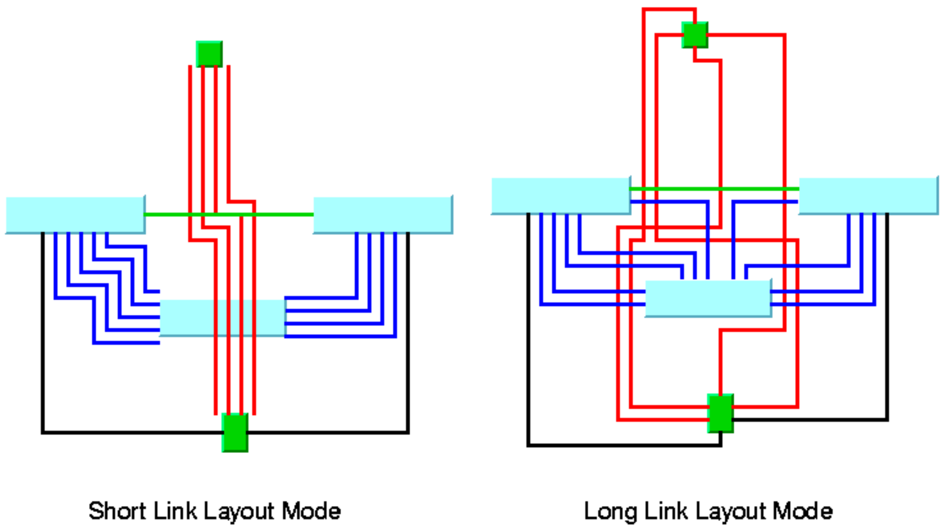
In Java™
Use the method:

```
void setLayoutMode(int mode);
```

The valid values for `mode` are:

- ♦ `IlvLinkLayout.SHORT_LINKS` (the default)
- ♦ `IlvLinkLayout.LONG_LINKS`

Short and Long Link Modes with Orthogonal Links shows a small sample graph in short and long link mode. The short link mode bundles the links very well. However, due to the bundling, some red links appear to be unconnected to the green nodes. Furthermore, the algorithm cannot find a route for the long red links without overlapping some nodes or without overlapping the green link. The long link mode works on a grid. It is specialized for long links and avoids overlapping any nodes or link segments. It can connect to the green nodes by choosing connection points on different sides of the end nodes. This advantage, however, is paid for by a less regular structure that does not bundle the links and a larger number of link crossings.



Short and Long Link Modes with Orthogonal Links

Choosing the appropriate layout mode (LL)

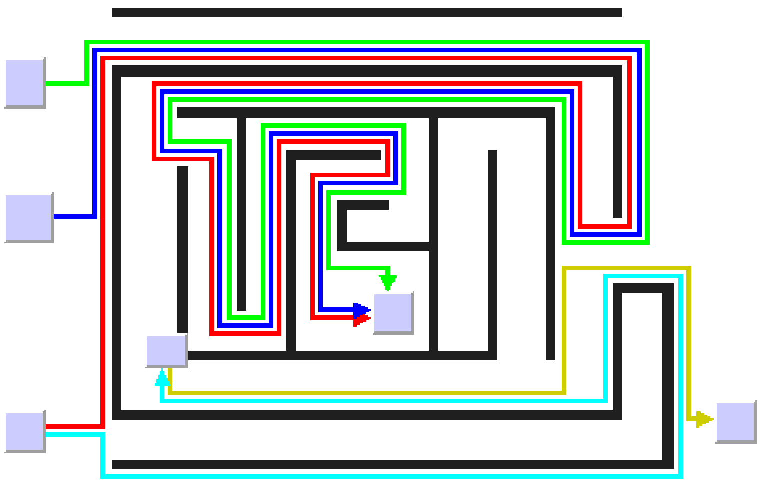
The short link mode should be used if any of the following conditions apply:

- The majority of links is short and it is not fatal if long links overlap obstacles.
- The link routes must be placed freely and cannot be restricted to a grid.
- It is important to limit the number of bends.

The long link mode should be used if any of the following conditions apply:

- Many links are long and it is important that long links do not overlap obstacles.
- There is a preferred routing because the nodes are already placed on the grid.
- It is important to have a guaranteed minimal distance between link segments.
- An increasing number of bends is acceptable if it avoids any overlappings.

Labyrinth routing with the long link mode shows how the long link mode can be used to find an orthogonal route without overlappings in a labyrinth of node obstacles.



Labyrinth routing with the long link mode

Link style (LL)

The layout algorithm provides two link styles. You can set the link style globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Note: The layout algorithm may raise an `IlvInappropriateLinkException` if layout is performed on an `IlvGrapher`, but inappropriate link classes or link connector classes are used. See *Layout exceptions* for details and solutions to this problem.

Global link style

Example of setting global link style (Link Layout algorithm)

To set the global link style:

In CSS

Add to the `LinkLayout` section:

```
globalLinkStyle: "ORTHOGONAL_STYLE";
```

In Java

Use the method:

```
void setGlobalLinkStyle(int style);
```

The valid values for `style` are:

- `IlvLinkLayout.ORTHOGONAL_STYLE` (the default)

  The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See *Link layout in short link mode with orthogonal links* and *Link layout in long link mode with orthogonal links* as examples.

- `IlvLinkLayout.DIRECT_STYLE`

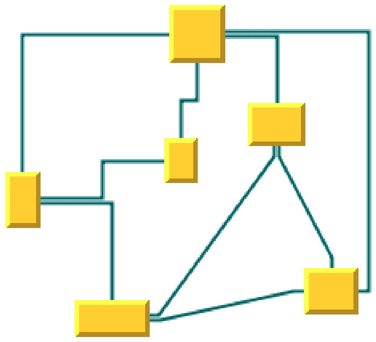
  The links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with a small horizontal or vertical segment. See *The same graph in short link mode with direct links* as an example.

- `IlvLinkLayout.MIXED_STYLE`

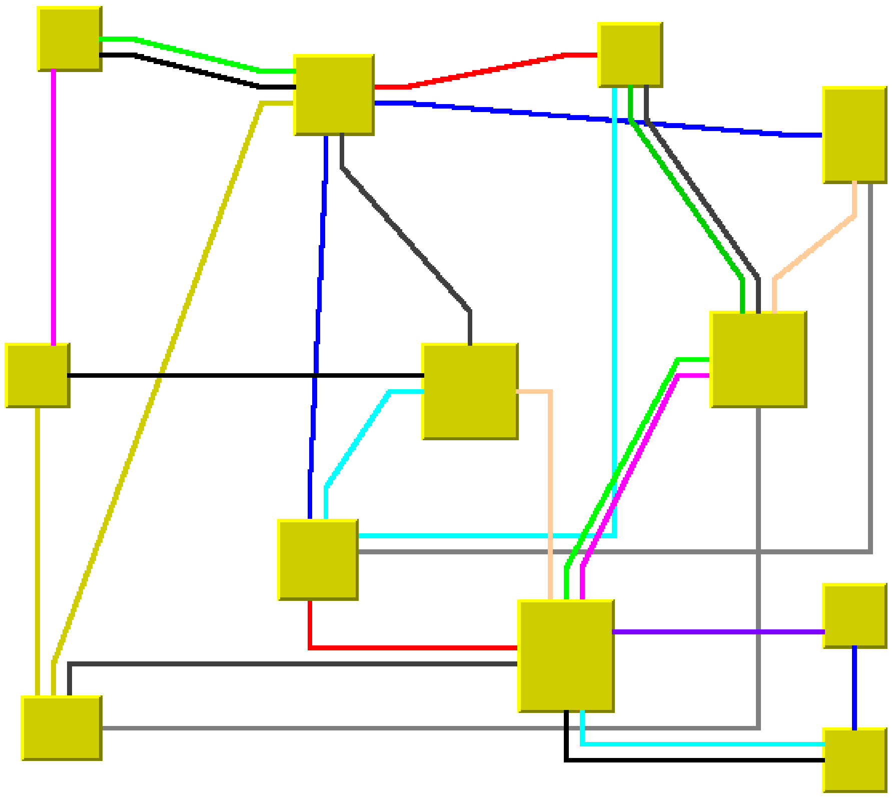
  Each link can have a different link style. The style of each individual link can be set to have different link shapes occurring on the same graph.

Individual link style

All links have the same style of shape unless the global link style is `IlvLinkLayout.MIXED_STYLE`. Only when the global link style is set to `MIXED_STYLE` can each link have an individual link style.



Different link styles mixed in the same drawing (short link mode)



Different link styles mixed in the same drawing (long link mode)

Example of specifying individual link style (Link Layout algorithm)
To set and retrieve the style of an individual link:

In CSS
First set the global link style to MIXED_STYLE, then write a rule to select the link:

```
LinkLayout {
   globalLinkStyle : "MIXED_STYLE";
}
#link1 {
  LinkStyle: "DIRECT_STYLE ";
}
```

In Java
Use the methods:

```
void setLinkStyle(Object link, int style);
```

```
int getLinkStyle(Object link);
```

The valid values for `style` are:

- ♦ `IlvLinkLayout.ORTHOGONAL_STYLE` (the default)
- ♦ `IlvLinkLayout.DIRECT_STYLE`
- ♦ `IlvLinkLayout.NO_RESHAPE_STYLE` (that is, the link is not reshape in any manner)

Note: The link style of a Link Layout graph requires links in an `IlvGrapher` that can be reshaped. Links of type `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage`, and `IlvDoubleSplineLinkImage` cannot be reshaped. You should use the class `IlvPolylineLinkImage` or `IlvSplineLinkImage` instead

End points mode (LL)

Normally, the layout algorithm is free to choose the termination points of each link. However, if fixed-link connectors are used (for instance, `IlvPinLinkConnector`), the user can specify that the current fixed termination pin of a link should be used.

The layout algorithm provides two end point modes. You can set the end point mode globally, in which case all end points have the same mode, or locally on each link, in which case different end point modes occur in the same drawing.

Global end point mode

Example of specifying global end point mode (Link Layout algorithm)
To set the global end point mode:

In CSS
Add to the `LinkLayout` section:

```
globalOriginPointMode : "FIXED_MODE";
globalDestinationPointMode : "FIXED_MODE";
```

In Java

```
void setGlobalOriginPointMode(int mode);
```

```
void setGlobalDestinationPointMode(int mode);
```

The valid values for `mode` are:

- ♦ `IlvLinkLayout.FREE_MODE` (the default)

  The layout is free to choose the appropriate position of the connection point on the origin/destination node.

♦ `IlvLinkLayout.FIXED_MODE`

  The layout must keep the current position of the connection point on the origin/destination node.

♦ `IlvLinkLayout.MIXED_MODE`

  Each link can have a different end point mode.

The connection points are automatically considered as fixed if they are connected to grapher pins.

Individual end point mode

All links have the same end point mode unless the global end point mode is `IlvLinkLayout.MIXED_MODE`. Only when the global end point mode is set to `MIXED_MODE` can each link have an individual end point mode.

Example of specifying individual end point mode (Link Layout algorithm)
To set the mode of an individual link:

In CSS
First set the global origin and destination point mode to MIXED_MODE, then write a rule that selects the link:

```
LinkLayout {
   globalOriginPointMode      : "MIXED_MODE";
   globalDestinationPointMode : "MIXED_MODE";
}
#link1{
  OriginPointMode      : "FREE_MODE";
  DestinationPointMode : "FREE_MODE";
}
```

In Java
Use the methods:

```
void setOriginPointMode(Object link, int mode);
```

```
int getOriginPointMode(Object link);
```

```
void setDestinationPointMode(Object link, int mode);
```

```
int getDestinationPointMode(Object link);
```

The valid values for `mode` are:

♦ `IlvLinkLayout.FREE_MODE` (the default)

♦ `IlvLinkLayout.FIXED_MODE`

The connection points are automatically considered as fixed if they are connected to grapher pins.

Note: The layout algorithm may raise an `IlvInappropriateLinkException` if layout is performed on an `IlvGrapher`, but inappropriate link classes or link connector classes are used. See *Layout exceptions* for details and solutions to this problem.

Incremental mode (LL)

The Link Layout algorithm normally routes all links from scratch. If the graph changes incrementally because you add or remove links or nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode.

Example of enabling incremental mode (Link Layout algorithm)
To enable the incremental mode:

In CSS
Add to the `LinkLayout` section:

```
incrementalMode : "true";
```

In Java
Call:

```
layout.setIncrementalMode(true);
```

In incremental mode, the layout tries to minimize the changes to the layout. A link is only rerouted if it is new, if a link bend moved, if its layout parameters have changed, or if a node was moved such that it overlaps the link.

In short link mode, if the next layout is incremental, the links preserve the connection side and the general shape calculated by a previous layout, except if one of their end nodes has been moved or resized.

In the long link mode, a new route is searched for the links that are no longer on the grid or that overlap with nodes. The shape and the connection side of the rerouted links can change completely. However, links that are already on the grid and do not overlap nodes or other links are not rerouted in incremental mode. It is also possible to specify which link must be rerouted by the next incremental layout even though the layout has not changed.

Example of specifying which link must be rerouted by the next incremental layout (Link Layout algorithm)
To select an individual link to be used for incremental rerouting:

In CSS
Write a rule to select the link:

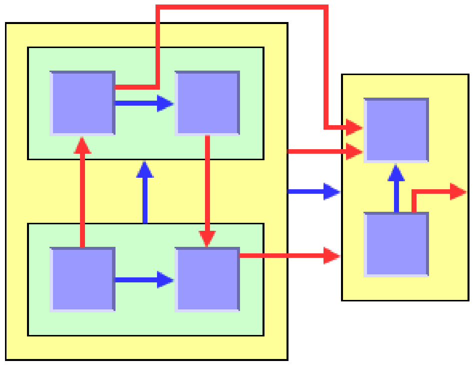
```
#link1 {
  MarkForIncremental: "true";
}
```

In Java
Use the method:

```
void markForIncremental(Object link);
```

Intergraph link routing (LL)

A nested graph is a graph with nodes that are subgraphs. In a nested graph, normal links and intergraph links can occur (see Nested managers and nested graphers in *Advanced Features of JViews Framework*). Normally, both end nodes of a link belong to the same subgraph. Intergraph links are those links whose end nodes belong to different subgraphs. Intergraph links belong to the lowest common grapher in the nesting structure that contains both end nodes. The following figure shows a nested graph with blue normal links and red intergraph links.



Nested Graph With Normal Links (blue) and Intergraph Links (red)

By default, the Link Layout routes both the normal links and the intergraph links.

Example of routing only normal links (Link Layout algorithm)
In order to route only normal links, disable intergraph link routing:

In CSS
Add to the `LinkLayout` section:

```
interGraphLinksMode : "false" ;
```

In Java™
Call:

```
layout.setInterGraphLinksMode(false);
```

Example of routing intergraph and/or normal links (Link Layout algorithm)
If the intergraph links mode is enabled, you can select whether only the intergraph links are routed, or whether the intergraph links and the normal links are routed at the same time.

In CSS
If you set:

```
combinedInterGraphLinksMode : "false";
```

the next layout routes the intergraph links but does not reshape any normal links.

If you set:

```
combinedInterGraphLinksMode : "true"
```

the next layout routes both the normal links and the intergraph links.

In Java
If you call:

```
layout.setCombinedInterGraphLinksMode(false);
```

the next layout routes the intergraph links but does not reshape any normal links. If you call:

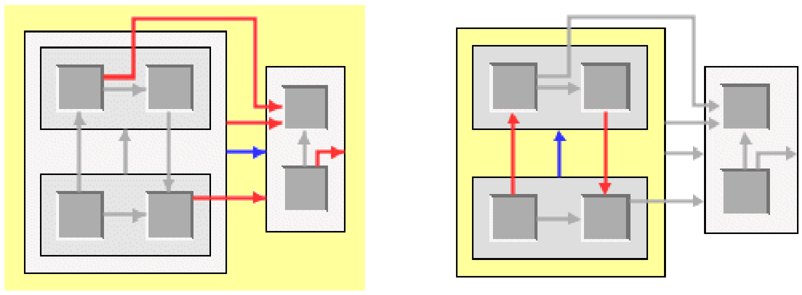
```
layout.setCombinedInterGraphLinksMode(true);
```

the next layout routes both the normal links and the intergraph links.

When the intergraph links mode is enabled, the layout cannot route the links incrementally (see *Incremental mode (LL)*) and the layout animation is disabled (see *Animation*).

Notice that the layout routes only those links that belong to the attached graph. In a nested graph, each subgraph is attached to a different layout instance. Therefore, when starting a normal (nonrecursive) layout for the top-level graph (see *Nested Graph With Normal Links (blue) and Intergraph Links (red)*) not all links are routed that are shown in this figure, but only those links that belong to the top-level graph.

The following figure shows two situations: the yellow subgraph indicates the subgraph where the nonrecursive layout is currently applied, and color of the links indicate which links are currently routed. Depending on the intergraph links mode, the red and/or blue links are routed, but the grey links are not reshaped.



Routed Link in a Nested Graph when Layout is Performed for the Yellow Subgraph

To route *all links* of a nested graph, you need to apply the Link Layout recursively. Details of the recursive layout mechanism are explained in *Recursive layout*. For instance:

```
layout.setInterGraphLinksMode(true);
layout.performLayout(force, redraw, true);
```

routes the intergraph links recursively in all subgraphs. If you use a layout provider (a class that implements the interface `IlvLayoutProvider`), you need to set the intergraph links mode for all subgraphs explicitly:

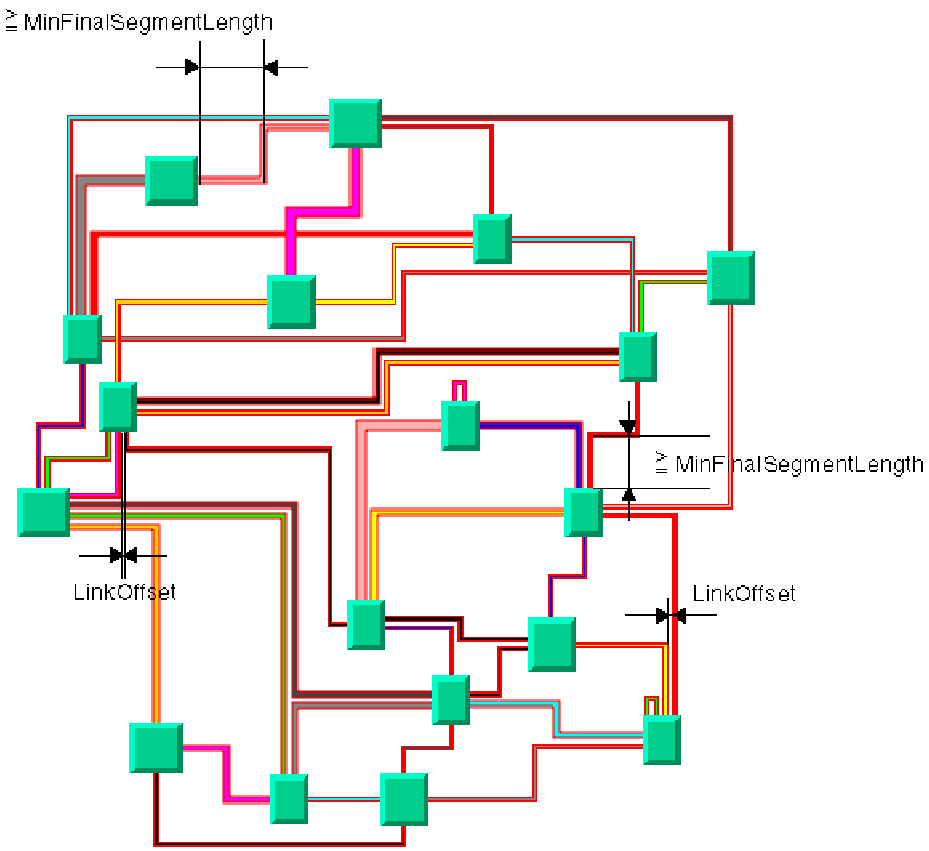
```
IlvLayoutProvider layoutProvider = ...
// first, set the intergraph mode for all layouts
Enumeration e = graphModel.getLayouts(layoutProvider, true);
while (e.hasMoreElements()) {
    IlvGraphLayout layout = (IlvGraphLayout) e.nextElement();
    if (layout instanceof IlvLinkLayout)
        ((IlvLinkLayout) layout).setInterGraphLinksMode(true);
}
// then perform layout recursively using the provider
graphModel.performLayout(layoutProvider, force, redraw, true);
```

If you want to recursively perform the intergraph link routing in combination with a layout that places the nodes or that arranges labels, we recommend that you use an instance of the class `IlvMultipleLayout` to encapsulate the Link Layout and the other layouts, and then perform the Multiple Layout recursively all at once. For details, see *Recursive layout*.

Spacing parameters in short link mode

Since the short link mode places the links freely in the space, only two parameters are necessary to control the spacing: the minimal distance between links and the minimal length of the final segment.

Spacing Parameters for the Short Link Mode shows the spacing parameters used in the short link mode.



Spacing Parameters for the Short Link Mode

Link offset

The layout algorithm computes the final connecting segments of the links (that is, the segments near the origin and destination nodes) to obtain parallel lines spaced at a user-defined distance. In short link mode, the algorithm takes into account the width of the links when computing the offset.

Example of specifying link offset (Link Layout algorithm)
To specify the offset:

In CSS
Add to the LinkLayout section:

```
layoutMode : "SHORT_LINKS";
linkOffset : "3.00";
```

In Java™
Use the method:

```
void setLinkOffset(float offset)
```

The offset is measured from the border of one link to the nearest border of the other link. Therefore, if the specified offset is zero, the border of a link touches the border of its neighbor link.

Minimum final segment length

You can specify a minimum value for the length of the final connecting segments of the links (that is, the segments near the origin and destination nodes).

Example of specifying minimum final segment length (Link Layout algorithm)
In CSS
Add to the LinkLayout section:

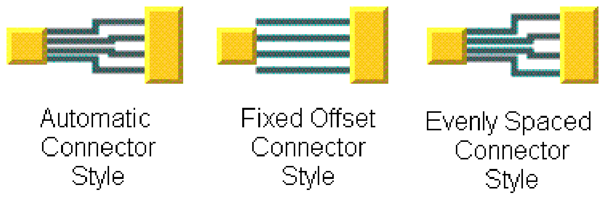
```
layoutMode : "SHORT_LINKS";
minFinalSegmentLength : "15.0";
```

In Java
Use the method:

```
void setMinFinalSegmentLength(float length)
```

Connector style

The layout algorithm positions the end points of links (the connector pins) at the nodes automatically. The connector style parameter specifies how these end points are calculated.


Automatic Connector Style
Fixed Offset Connector Style
Evenly Spaced Connector Style

Connector styles

The layout algorithm provides two connector styles. You can set the connector style globally, in which case all the nodes (hence, all the links) have the same kind of connector style, or locally on each node (that is, for all the links connected to the node), in which case different connector styles occur in the same drawing.

Global connector style

Example of specifying the global connector style (Link Layout algorithm)

To specify the global connector style:

In CSS

Add to the `LinkLayout` section:

```
globalConnectorStyle: "EVENLY_SPACED_PINS";
```

In Java

Use the following method:

```
void setGlobalConnectorStyle(int style);
```

The valid values for `style` are:

♦ `IlvShortLinkLayout.FIXED_OFFSET_PINS`

  The connection pins are spaced along the node border at a distance equal to the link offset parameter. See *Spacing Parameters for the Short Link Mode* as an example.

♦ `IlvShortLinkLayout.EVENLY_SPACED_PINS`

  The connector pins are evenly spaced along the node border, preserving a margin which is determined by the `evenlySpacedPinsMarginRatio` parameter (see the accessor `getEvenlySpacedPinsMarginRatio()`). See *Spacing Parameters for the Short Link Mode* as an example.

♦ `IlvShortLinkLayout.AUTOMATIC_PINS` (the default)

  Uses the connector style `FIXED_OFFSET_PINS` except if this pushes a connection point outside the border the link is attached to, in which case it uses the connector style `EVENLY_SPACED_PINS`. See *Spacing Parameters for the Short Link Mode* as an example.

♦ `IlvShortLinkLayout.MIXED_STYLE`

  Each node can have a different connector style. The style of each individual node can be set to have different connector styles occurring on the same graph.

In CSS, you omit the prefix `IlvShortLinkLayout` when specifying the value of the connector style.

Individual connector style

All nodes have the same connector style unless the global connector style is `IlvShortLinkLayout.MIXED_STYLE`. Only when the global connector style is set to `MIXED_STYLE` can each node have an individual connector style.

Example of specifying individual node connector style (Link Layout algorithm)

To specify the connector style of an individual node:

In CSS
Specify a rule that selects the node, for instance:

```
LinkLayout {
   globalConnectorStyle : "MIXED_STYLE";
}
#node1
{
  ConnectorStyle : "EVENLY_SPACED_PINS";
}
```

In Java
Use the following methods:

```
void setConnectorStyle(Object node, int style);
```

```
int getConnectorStyle(Object node);
```

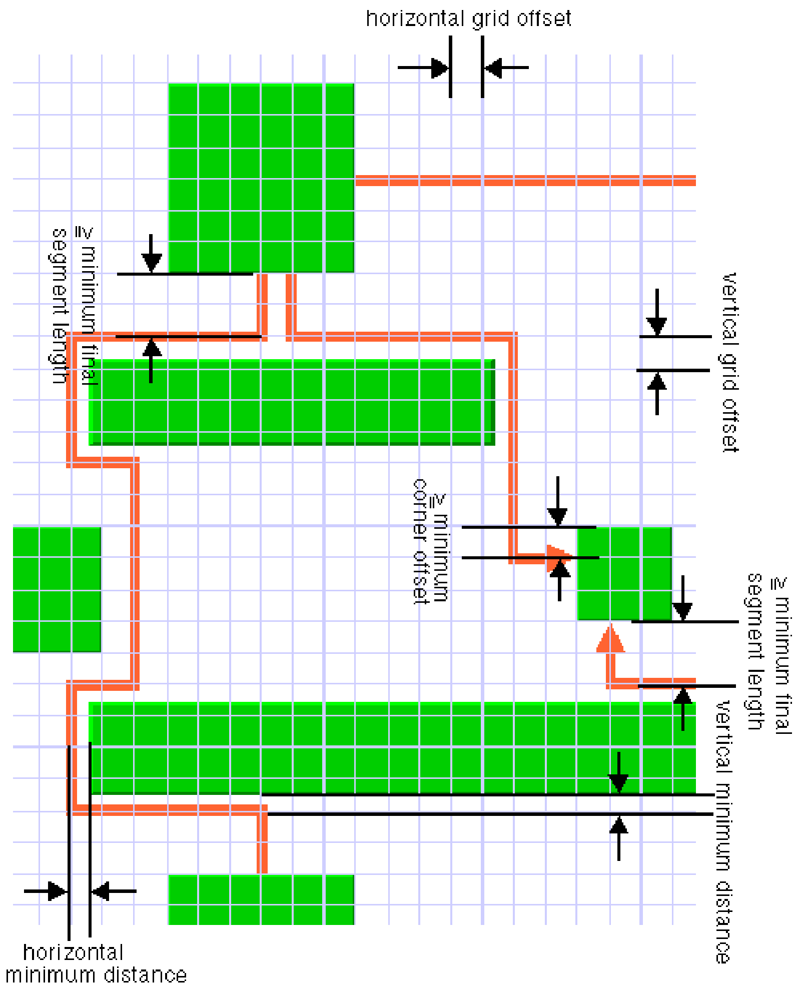
The valid values for `style` are:

- `IlvShortLinkLayout.FIXED_OFFSET_PINS`
- `IlvShortLinkLayout.EVENLY_SPACED_PINS`
- `IlvShortLinkLayout.AUTOMATIC_PINS` (the default).

The default value is 10.

Spacing parameters in long link mode

The long link mode places the links on a grid. Four parameters control the grid offsets and five parameters control the spacing of links in relation to other objects. *Spacing parameters for the long link mode* shows the spacing parameters used in the long link mode.


horizontal grid offset
≥ minimum final segment length
vertical grid offset
≥ minimum corner offset
≥ minimum final segment length
vertical minimum distance
horizontal minimum distance

Spacing parameters for the long link mode

Grid offset parameters

The grid offset parameters control the spacing between grid lines. Links are routed such that the center of the orthogonal link segments are on the grid lines. The grid offsets should be set to a value larger than the largest link width value to avoid links that visually overlap.

Example of specifying grid offset parameters (Link Layout algorithm)
To set the horizontal and vertical grid offset:

In CSS
Add these statements to the `LinkLayout` section:

```
horizontalGridOffset : "5.0";
verticalGridOffset : "5.0";
```

In Java
In Java™ , use the methods:

```
void setHorizontalGridOffset(float offset);
```

```
void setVerticalGridOffset(float offset);
```

The grid offset is the critical parameter for the long link mode. If the grid offset is too large, there may be no grid lines between nodes even though some free space exists between the nodes. In this case, the link routings cannot use the free space. However, if the grid offset is too small, the algorithm needs a long time to traverse the grid.

Grid base parameters

Sometimes it is necessary to shift the whole grid by a small amount because the nodes are not aligned on the grid. For instance, to have grid lines at positions 3, 13, 23, 33, and so on, you can set the grid offset to 10 and the grid base to 3.

Example of specifying grid base parameters (Link Layout algorithm)
To adjust the grid base:

In CSS
Add these statements to the `LinkLayout` section:

```
horizontalGridBase : "3.0";
verticalGridBase : "3.0";
```

In Java
Use the methods:

```
void setHorizontalGridBase(float coordinate);
void setVerticalGridBase(float coordinate);
```

Minimum distance parameters

The minimum distance controls how closely a link can be placed to the border of a node that needs to be bypassed. If the node border is not aligned to the grid, the minimum distance specifies the next grid line close to the border that can be used. For instance, if a node covers the x-coordinates 25 to 65 on a grid with offset 10 and base 0, the next grid lines used to bypass the node would normally be at 20 and 70. If you specify a minimum distance of 8, these grid lines are too close to the node and then the grid lines at 10 and 80 would be used.

Example of specifying minimum distance parameters (Link Layout algorithm)
To set the minimum distance:

In CSS
Add these statements to the `LinkLayout` section:

```
horizontalMinOffset : "10.25";
verticalMinOffset : "10.25";
```

In Java
Use the methods:

```
void setHorizontalMinOffset(float offset);
```

```
void setVerticalMinOffset(float offset);
```

Minimum node corner offset parameter

The minimum corner offset is the minimum distance between a node corner and a link that connects to the node. This parameter is used to avoid having a link that connects exactly to the corner or outside the border of the node (see *Minimal corner offset*).

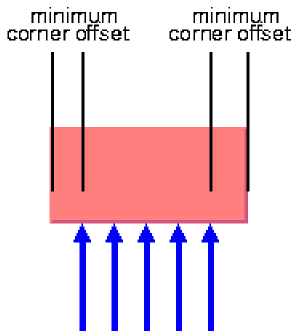
Example of specifying minimum node corner offset parameter (Link Layout algorithm)
To set the minimum corner offset:

In CSS
Add to the `LinkLayout` section:

```
minNodeCornerOffset : "5.2";
```

In Java
Use the method:

```
void setMinNodeCornerOffset(float offset);
```



Minimal corner offset

Minimum final segment length

As with the short link mode, the long link mode respects the minimum value for the length of the final connecting segments of the links.

Example of specifying minimum final segment length (Link Layout algorithm)
To set the minimal length of the final segment:

In CSS
Add to the `LinkLayout` section:

```
minFinalSegmentLength : "15.0";
```

In Java
Use the method:

```
void setMinFinalSegmentLength(float length)
```

For experts: additional features of LL

Using a node-side filter

Some applications require that links are not connected to specific sides of certain nodes. The Link Layout algorithm allows you to restrict to which node side a link can connect by using a node-side filter. A node-side filter is any class that implements the interface `IlvNodeSideFilter`. This interface defines the following method:.

```
public boolean accept(IlvGraphModel graphModel,
                      Object link,
                      boolean origin,
                      Object node,
                      int side);
```

This method allows you to let the input `link` to connect its `origin` or destination to the input `side` of the input `node`.

As an example, assume that the application requires that for end nodes of type `IlvShadowRectangle`, links can connect their origin only at the top and bottom side.

For end nodes of type `IlvReliefRectangle`, links can connect their destination only at the left and right side. You can obtain this result with the following node-side filter:

```
class MyFilter implements IlvNodeSideFilter
{
     public boolean accept(IlvGraphModel graphModel,
                           Object link,
                           boolean origin,
                           Object node,
                           int side)
    {
        if (node instanceof IlvShadowRectangle && origin)
            return(side == IlvDirection.Top || side == IlvDirection.Bottom);
        if (node instanceof IlvReliefRectangle && !origin)
            return(side == IlvDirection.Left || side == IlvDirection.Right);
        return true;
    }
}
```

Example of setting node-side filter (Link Layout algorithm)
To set this node-side filter:

In CSS
SDM allows you to specify the node-side constraints using the `NodeSideForOrigin` and `NodeSideForDestination` properties. For more information, see Per-object properties of the LinkLayout renderer in *Developing with the JViews Diagrammer SDK*.

In Java
In Java™ , call:

```
layout.setNodeSideFilter(new MyFilter());
```

To remove the node-side filter, call:

```
layout.setNodeSideFilter(null);
```

Using a node box interface

Some applications require that effective area of a node is not exactly its bounding box. For instance, if the node has a shadow, the shadow is included in the bounding box. However, the shadow may not be considered as an obstacle for the links. In this case, the effective bounding box of a node is smaller than the bounding box returned by `IlvGraphic.boundingBox`.

Example of using a node box interface (Link Layout algorithm)

In CSS

It is not possible to set the node box interface.

In Java™

You can modify the effective bounding box of a node by implementing a class that implements the `IlvNodeBoxInterface`.

This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective bounding box. For instance, to set a node box interface that returns a smaller bounding box for all nodes of type `IlvShadowRectangle`, call:

```
layout.setNodeBoxInterface(new IlvNodeBoxInterface() {
        public IlvRect getBox(IlvGraphModel graphModel, Object node) {
            IlvRect rect = graphModel.boundingBox(node);
            if (node instanceof IlvShadowRectangle) {
                // need a rect that is 4 units smaller
                rect.resize(rect.width-4.f, rect.height-4.f);
            }
            return rect;
        }
    });
```

Using a link connection box interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes. Sometimes, it may be necessary to place the connection points on a rectangle that is smaller or larger than the bounding box. For instance, this can happen when labels are displayed below or above nodes.

Example of using a link connection box interface to modify position of connection points (Link Layout algorithm)

In CSS

It is not possible to set the link connection box interface.

In Java

You can modify the position of the connection points of the links by implementing a class that implements the `IlvLinkConnectionBoxInterface`. This is a subinterface of `IlvNodeBoxInterface` (see *Using a node box interface*). It defines again the method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

Additionally, the interface `IlvLinkConnectionBoxInterface` defines a second method:

```
public float getTangentialOffset(IlvGraphModel graphModel, Object node, int
nodeSide);
```

This method is used only in the short link mode. For details, see *Using a link connection box interface*. When using the Link Layout in long link mode, just implement the method by returning the value 0.

For experts: special options of the Short LL

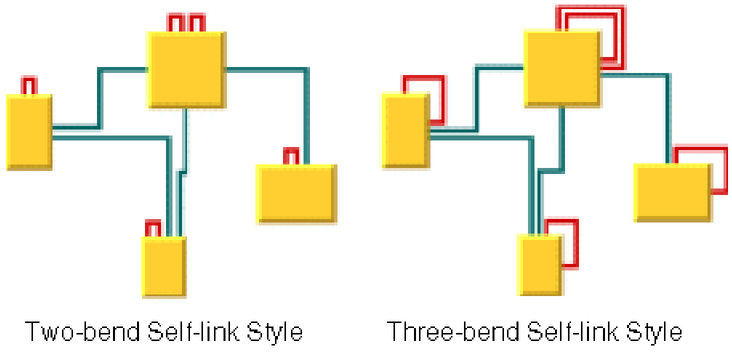
The Link Layout algorithm utilizes the class `IlvShortLinkLayout` as a subalgorithm. `IlvShortLinkLayout` is a subclass of `IlvGraphLayout` and can be used a stand-alone as well. To access the instance of `IlvShortLinkLayout` that is used by the Link Layout algorithm, call:

```
IlvShortLinkLayout getShortLinkLayout();
```

Using this accessor, you can control many special features of the Short Link Layout that are not made available by the `IlvLinkLayout` class because these features are for experts only.

Self-link style

Self-links are links whose origin and destination is the same node. The Short Link Layout provides two optional shapes for self-links.



Self-link Style Options

Example of setting the style of the self-links (Link Layout algorithm)
To set the style of the self-links:

In CSS
Add to the `LinkLayout` section:

```
layoutMode : "SHORT_LINKS";
globalSelfLinkStyle : "TWO_BENDS_ORTHOGONAL_STYLE";
```

In Java™
Call `layout.getShortLinkLayout(). setGlobalSelfLinkStyle(int)`

The valid values for `style` are:

♦ `IlvShortLinkLayout.TWO_BENDS_ORTHOGONAL_STYLE`

♦ `IlvShortLinkLayout.THREE_BENDS_ORTHOGONAL_STYLE`

Number of optimization iterations

The link shape optimization is stopped if the time exceeds the allowed time (see *Allowed time (LL)*) or if the number of iterations exceeds the allowed number of iterations.

Example of specifying the number of optimization iterations (Link Layout algorithm)
To set the allowed number of iterations to 3:

In CSS
Add to the `LinkLayout` section:

```
layoutMode : "SHORT_LINKS";
allowedNumberOfIterations : "3";
```

In Java
Call:

```
layout.getShortLinkLayout().setAllowedNumberOfIterations(3);
```

Note: You may want to disable the link shape optimization by setting the number of iterations to zero to increase the speed of the layout process.

Evenly spaced pins margin ratio

The margin ratio allows you to customize the way connection points are computed when the connector style (see *Connector style*) is `EVENLY_SPACED_PINS`, and when the `AUTOMATIC_STYLE` places the connection points using the `EVENLY_SPACED_PINS` style. This option has no effect if the connector style `FIXED_OFFSET_PINS` is used.

In the “evenly spaced pins” connector style, the connection points of the links are evenly spaced along the node border, preserving a margin to each extremity of the node border. The size of this margin is controlled by the margin ratio and is computed by multiplying the offset between the links by the ratio.

Example of specifying margin ratio (Link Layout algorithm)
To specify this option

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "SHORT_LINKS";
evenlySpacedMarginRatio: "0.2";
```

In Java
Call `layout.getShortLinkLayout(). setEvenlySpacedPinsMarginRatio(float)`

The input value must be a positive or zero value. The default value is `0.5`. *Evenly Spaced Pins Margin Ratio* shows examples of values with their meaning.

Evenly Spaced Pins Margin Ratio

Ratio value	Meaning
0	No margin
0.5 (default value)	The margin is equal to half the offset between the links.
1	The margin is equal to the offset between the links.
2	The margin is equal to twice the offset between the links.

Link overlap nodes forbidden

This option allows you to ask the layout algorithm to avoid strictly to reshape links such that they overlap some nodes. If overlaps are not forbidden, the algorithm tries to avoid overlaps anyway, but may create overlaps, for instance for the link to cross other links.

Note: Forbidding overlaps may slow down the layout and may increase the number of bends for those links that would overlap nodes if overlaps were not strictly forbidden.

Example of specifying link overlap nodes forbidden (Link Layout algorithm)
To specify this option:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "SHORT_LINKS";
linkOverlapNodesForbidden: "true";
```

In Java
Call

`layout.getShortLinkLayout(). setLinkOverlapNodesForbidden(boolean)`

The default value of this option is `false`.

When overlaps are forbidden, the Short Link Layout algorithm uses the Long Link Layout as an auxiliary algorithm for laying out only the links that would otherwise overlap nodes.

Example of specifying Long Link Layout when overlaps forbidden (Link Layout algorithm)
To retrieve the auxiliary instance of Long Link Layout:

In CSS
It is not possible to access the auxiliary Long Link Layout, nor to tailor this auxiliary Long Link Layout.

In Java
Call this method on the `IlvShortLinkLayout` instance:

```
IlvLongLinkLayout getAuxiliaryLongLinkLayout()
```

This method allows you to get this auxiliary layout instance and to customize its parameters if needed. Notice that you should neither modify the origin and destination point mode, nor disable the preservation of fixed links. Notice also that an `IlvGraphModel` instance is attached to the `IlvLongLinkLayout` instance only if needed, therefore the method `getAuxiliaryLongLinkLayout().getGraphModel()` may return `null`.

Incremental link reshape mode

In incremental mode, it is possible to customize the rules used by the Short Link Layout to determine which links should keep their current shape as much as possible, as computed by the previous layout execution. The incremental link reshape mode allows you to customize these rules separately for two categories of links. See the methods:

```
IlvShortLinkLayout.getLinkConnectionBoxInterface()
```

and

```
IlvShortLinkLayout.getNodeBoxInterface()
```

- ♦ The “modified links”: the links that have either a different “link connection box” or are connected to nodes which have a different bounding box as during the previous layout execution.
- ♦ The “unmodified links”: the links that have the same “link connection box” and are connected to nodes which have the same bounding box as during the previous layout execution.

The mode can be customized either for both or for only one of these categories of links.

The incremental link reshape mode has no effect if the incremental mode is disabled.

The layout algorithm provides two incremental link reshape modes. You can set the mode globally, in which case all the links have the same mode, or locally on each link, in which case different modes occur in the same drawing.

Global incremental link reshape mode

Example of specifying global incremental link reshape mode (Link Layout algorithm)
To specify the global incremental link reshape mode:

In CSS
Add, for instance, these statements to the `LinkLayout` section:

```
globalIncrementalModifiedLinkReshapeMode: "FIXED_NODE_SIDES_MODE";
globalIncrementalUnmodifiedLinkReshapeMode: "FIXED_SHAPE_TYPE_MODE";
```

In Java
Use the following methods:

```
layout.getShortLinkLayout(). setGlobalIncrementalModifiedLinkReshapeMode
```

```
layout.getShortLinkLayout(). setGlobalIncrementalUnmodifiedLinkReshapeMode
```

The valid values for `mode` are:

- `IlvShortLinkLayout.FIXED_SHAPE_TYPE_MODE` (the default)

  The incremental layout preserves the shape type of the link. This means that both the number of bends and the node sides to which the link is connected are preserved.

- `IlvShortLinkLayout.FIXED_NODE_SIDES_MODE`

  The incremental layout preserves the node sides to which the links are connected.

- `IlvShortLinkLayout.FIXED_CONNECTION_POINTS_MODE`

  The incremental layout preserves the connection points of the links.

- `IlvShortLinkLayout.FIXED_MODE`

  The links are not reshaped at all during incremental layout. Only newly added links are rerouted.

- `IlvShortLinkLayout.FREE_MODE`

  The incremental layout is allowed to freely reshape the links. This is equivalent to a non-incremental behavior for all the links, hence it is recommended to disable the incremental mode instead of using `FREE_MODE` as global incremental reshape mode.

  Of course, the settings that may have been done by "fixing" links (see *Preserve fixed links (LL)*) or by customizing the origin or destination point mode (see *End points mode (LL)*) are still respected.

- `IlvShortLinkLayout.MIXED_MODE`

  Each link can have a different mode.

In CSS, you can omit the prefix `IlvShortLinkLayout` when specifying the value of the mode.

Individual incremental link reshape mode

All links have the same incremental link reshape mode unless the global incremental link reshape mode is `IlvShortLinkLayout.MIXED_MODE`. Only when the global mode is set to `MIXED_MODE` can each link have an individual mode.

Example of specifying individual incremental link reshape mode (Link Layout algorithm)
To specify the mode of an individual link:

In CSS
Write a rule that selects the link, for instance:

```
LinkLayout {
  layoutMode: "SHORT_LINKS";
  globalIncrementalModifiedLinkReshapeMode: "MIXED_MODE";
  globalIncrementalUnmodifiedLinkReshapeMode: "MIXED_MODE";
}
#link1{
  IncrementalModifiedLinkReshapeMode: "FIXED_NODE_SIDES_MODE";
  IncrementalUnmodifiedLinkReshapeMode: "FIXED_SHAPE_TYPE_MODE";
}
```

In Java
Use the following methods on the `IlvShortLinkLayout` instance:

```
void setIncrementalModifiedLinkReshapeMode(Object link, int mode);
```

```
void setIncrementalUnmodifiedLinkReshapeMode(Object link, int mode);
```

```
int getIncrementalModifiedLinkReshapeMode(Object link);
```

```
int getIncrementalUnmodifiedLinkReshapeMode(Object link);
```

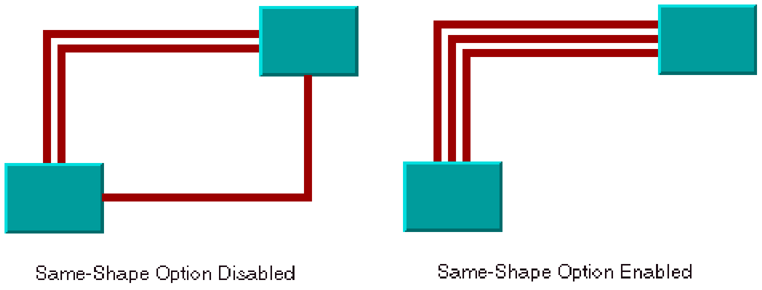
The valid values for `mode` are:

- `IlvShortLinkLayout.FIXED_SHAPE_TYPE_MODE` (the default)
- `IlvShortLinkLayout.FIXED_NODE_SIDES_MODE`
- `IlvShortLinkLayout.FIXED_CONNECTION_POINTS_MODE`
- `IlvShortLinkLayout.FREE_MODE`
- `IlvShortLinkLayout.FIXED_MODE`

Same shape for multiple links

You can force the layout algorithm to compute the same shape for all the links having common origin and destination nodes. The links will have parallel shapes.

When this option is disabled, the layout is free to compute different shapes for links connecting the same pair of nodes. Generally, different shapes are chosen to avoid some overlaps.



Self-link style options

Example of specifying same shape for multiple links (Link Layout algorithm)
To enable same shape for multiple links:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "SHORT_LINKS";
sameShapeForMultipleLinks : "true";
```

In Java
Use the method:

```
layout.getShortLinkLayout().setSameShapeForMultipleLinks(true);
```

The default value is `false`.

Link crossing penalty

The computation of the shape of the links is driven by the objective to minimize a cost function, which is proportional to the number of link-to-link crossings and link-to-node crossings. By default, these two types of crossings have equal weights of `1`. You can increase the weight of the link-to-node crossings.

Example of specifying link-to-node crossing penalty (Link Layout algorithm)
To increase the weight of the link-to-node crossings:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "SHORT_LINKS";
linkToNodeCrossingPenalty  : "5.0";
```

In Java
Use the method:

```
layout.getShortLinkLayout().setLinkToNodeCrossingPenalty(5.f);
```

This increases the possibility of obtaining a layout with no link-to-node crossings (or with only a few crossings), with the expense that there may be more link-to-link crossings.

Alternatively, you can increase the weight of the link-to-link crossings.

Example of specifying link-to-link crossing penalty (Link Layout algorithm)
To increase the weight of the link-to-link crossings,for instance, to a value of `3`s:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "SHORT_LINKS";
linkToLinkCrossingPenalty  : "3.0";
```

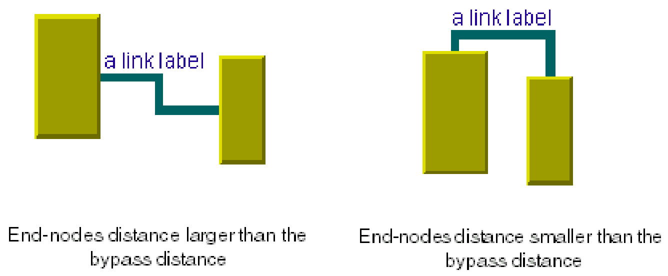
In Java
Use the method:

```
layout.getShortLinkLayout().setLinkToLinkCrossingPenalty(3.f);
```

This increases the possibility of obtaining a layout with no link-to-link crossings (or with only a few crossings), with the expense that there may be more link-to-node crossings.

Bypass distance

If the origin and destination nodes are too close, there may not be enough space for routing the link directly between the end nodes. Therefore, by default, if the end nodes are closer than a threshold distance, the layout chooses link shapes that bypass the interval between close nodes. (See *End nodes and bypass distance*.)


a link label
a link label
End-nodes distance larger than the bypass distance
End-nodes distance smaller than the bypass distance

End nodes and bypass distance

The bypass distance is the minimum distance between the origin and destination nodes for which a link shape going directly from one node to another is allowed. The algorithm tries to avoid link shapes that connect directly the sides of the end nodes that are closer than the bypass value.

Example of specifying the bypass distance (Link Layout algorithm)
To set the bypass distance:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "SHORT_LINKS";
bypassDistance  : "3.0";
```

In Java
Call

```
void setBypassDistance(float dist)
```

The default value is a strictly negative value. If the bypass distance is strictly negative, the value of the minimum final segment length (see *Minimum final segment length*) parameter is used as the bypass distance. This allows the automatic adjustment of the bypass distance according to the current value of the minimum final segment length. This behavior is suitable in most cases. However, you can specify a non-negative value to override the default behavior.

Using a link connection box interface

By default, the connection points of the links are distributed on the border of the bounding box of the nodes, symmetrically with respect to the middle of each side. Sometimes, it may be necessary to place the connection points on a rectangle smaller or larger than the bounding box, eventually in a nonsymmetric way. For instance, this can happen when labels are displayed below or above nodes.

Example of using a link connection box interface to modify the position of the connection points (Link Layout algorithm)

You can modify the position of the connection points of the links by implementing a class that implements the `IlvLinkConnectionBoxInterface`.

In CSS

It is not possible to set the link connection box interface.

In Java

This interface defines the following method:

```
public IlvRect getBox(IlvGraphModel graphModel, Object node);
```

This method allows you to return the effective rectangle on which the connection points of the links are placed.

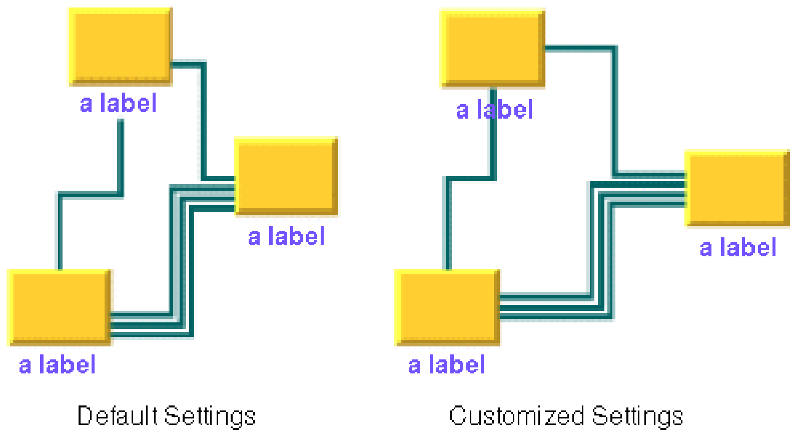
A second method defined on the interface allows the connection points to be “shifted” tangentially, in a different way for each side of each node:

```
public float getTangentialOffset(IlvGraphModel graphModel, Object node, int
nodeSide);
```

For instance, to set a link connection box interface that returns a link connection rectangle that is smaller than the bounding box for all nodes of type `MyNodeEditPart` and shifts up the connection points on the left and right side of all the nodes, call:

```
layout.setLinkConnectionBoxInterface(new IlvLinkConnectionBoxInterface() {
    public IlvRect getBox(IlvGraphModel graphModel, Object node) {
        IlvRect rect = graphModel.boundingBox(node);
        if (node instanceof MyNodeEditPart) {
           // for example, the size of the bounding box is reduced by 4 units

            rect.resize(rect.width-4.f, rect.height-4.f);
        }
        return rect;
    }
    public float getTangentialOffset(IlvGraphModel graphModel,
                                     Object node, int nodeSide) {
        switch (nodeSide) {
          case IlvDirection.Left:
          case IlvDirection.Right:
            return -10; // shift up with 10 for both left and right side
          case IlvDirection.Top:
          case IlvDirection.Bottom:
          default:
            return 0; // no shift for top and bottom side
        }
    }
});
```

Self-link Style Options shows the effects of customizing the connection box. On the left is the result using the default settings: the connection points are distributed on the bounding box of the node (which includes the label) and are symmetric with the middle of each node side (including the label). On the right, is the result after specifying a link connection box interface. On the bottom side of the nodes, the links are now connected to the node (passing over the label), while on the left and right side the nodes are now symmetric to the middle of the node (without the label).



Customization of the link connection box

For experts: special options of the Long LL

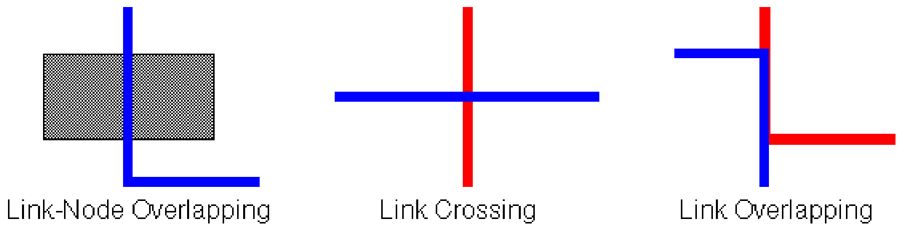
The Link Layout algorithm utilizes the class `IlvLongLinkLayout` as subalgorithm. `IlvLongLinkLayout` is a subclass of `IlvGraphLayout` and can be used a stand-alone as well. To access the instance of `IlvLongLinkLayout` that is used by the Link Layout algorithm, use the method:

```
IlvLongLinkLayout getLongLinkLayout();
```

Using this accessor, you can control many special features of the Long Link Layout that are not made available by the `IlvLinkLayout` class because these features are for experts only.

Specifying additional obstacles

The Long Link Layout algorithm considers nodes to be obstacles that cannot be overlapped and links to be obstacles that can be crossed at an angle of 90 degree (approximately, if the link style is direct), but that cannot be overlapped.



Crossings and Overlappings

Example of specifying additional obstacles (Link Layout algorithm)
If an application requires additional obstacles that are not links or nodes, these can be specified as follows:

In CSS
It is not possible to specify additional obstacles in CSS.

In Java™
Call:

```
layout.getLongLinkLayout().addRectObstacle(ilog.views.IlvRect)

layout.getLongLinkLayout().addLineObstacle(ilog.views.IlvRect)

layout.getLongLinkLayout().addLineObstacle
```

Rectangular obstacles behave like nodes: links cannot overlap the rectangles. Line obstacles behave like link segments: other links can cross the line segments, but cannot overlap the segments. These obstacle settings can be removed by the following:

```
layout.getLongLinkLayout().removeAllRectObstacles()

layout.getLongLinkLayout().
```

`removeAllLineObstacles()`

Penalties for variable end points

If the termination points of the links are not fixed, the algorithm uses a heuristic to calculate the termination points of each link. It examines all free grid points that are close to the border of the start and end node and assigns a penalty to each grid point. If a node-side filter is installed, the penalty depends on whether the node side is allowed or rejected.

A more precise way to affect how the termination points are chosen is the termination point filter. This enables the user to specify the penalty for each grid point.

Example of specifying the termination point filter (Link Layout algorithm)
In CSS
It is not possible to specify the termination point filter in CSS.

In Java
A termination point filter is a class that implements the interface `IlvTerminationPointFilter` that defines the following method:

```
public int getPenalty(IlvGraphModel graphModel, Object link,
boolean origin, Object node, IlvPoint point,
int side, int proposedPenalty);
```

To select the `origin` or destination point of the input `link`, the input `point` (a grid point on the input `side` of the `node`) is examined. The `proposedPenalty` is calculated by the default heuristic of the algorithm. You can return a changed penalty or you can return `java.lang.Integer.MAX_VALUE` to reject the grid point. If the grid point is rejected, it is not chosen as termination point of the link.

The termination point filter can be set as follows:

Call on the `IlvLongLinkLayout` instance: `setTerminationPointFilter`

Manipulating the routing phases

As mentioned in *Long Link Layout algorithm*, the algorithm first treats each link individually and then applies a crossing reduction phase to all links. To find a route for an individual link, the algorithm first checks whether a routing (such as a straight line or with only one bend) is possible. If this kind of routing is not possible, it uses a sophisticated, but more time consuming, grid search algorithm with backtracking to find a route with many bends.

Example of manipulating the routing phases (Link Layout algorithm)
To switch off the phase that finds a straight-line or one-bend routing:

In CSS
Add to the LinkLayout section:

```
layoutMode: "LONG_LINKS";
straightRouteEnabled: "false";
```

In Java
Call:

```
layout.getLongLinkLayout(). setStraightRouteEnabled(boolean)
```

The backtrack search for a route with many bends can be affected in the several ways.

A more convenient way is to specify the maximum time available to search for the route for each link.

Example of specifying backtrack steps (Link Layout algorithm)
You can specify the maximum number of backtrack steps by using the following:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "LONG_LINKS";
maxBackTrack: "1000";
```

In Java
In Java, call:

```
layout.getLongLinkLayout(). setMaxBacktrack(int)
```

The default maximum backtrack number is 30000.

Example of specifying maximal time for route search (Link Layout algorithm)
To specify the maximum time available to search for the route for each link.

In CSS

```
layoutMode: "LONG_LINKS";
allowedTimePerLink: "4000";
```

In Java
Call:

```
setAllowedTimePerLink(long)
```

The default allowed time per link is 2000 milliseconds (2 seconds).

Finally, you can specify how many steps should be done during the crossing reduction phase.

Example of specifying number of steps in crossin reduction phase (Link Layout algorithm)
To specify how many steps should be done during the crossing reduction phase:

In CSS
Add to the LinkLayout section

```
layoutMode: "LONG_LINKS";
numberCrossingReductionIterations: "5";
```

In Java
Call

```
setNumberCrossingReductionIterations(int)
```

Example of disabling crossing reduction (Link Layout algorithm)
You can disable the crossing reduction completely by using the following:

In CSS
Add to the LinkLayout section:

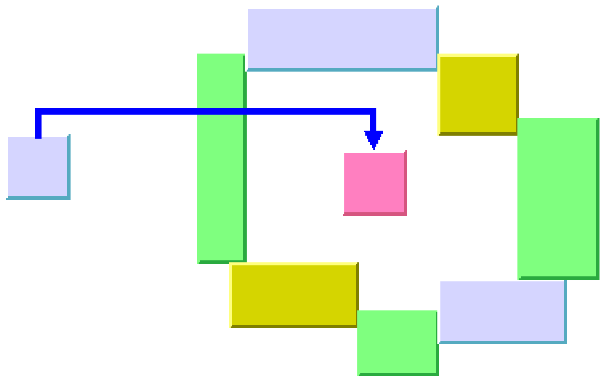
```
layoutMode: "LONG_LINKS";
crossingReductionEnabled: "false";
```

In Java
Call

```
setCrossingReductionEnabled(boolean)
```

Fallback mechanism

The Long Link Layout algorithm may not be able to find a routing for a link, if one of the end nodes is inside an enclave. In *A Node inside an enclave*, the red node is inside an enclave. In this case, the backtrack search algorithm fails to find a routing without overlapping nodes. The backtrack search algorithm may also fail if the situation is so complex that the search exceeds the allowed time per link.



A Node inside an enclave

When the backtrack search algorithm fails to find a routing, a simple fallback mechanism is applied that creates a routing with node overlappings.

Example of disabling fallback mechanism (Link Layout algorithm)
To disable the fallback mechanism:

In CSS
Add to the `LinkLayout` section:

```
layoutMode: "LONG_LINKS";
fallbackRouteEnabled: "false";
```

In Java

```
layout.getLongLinkLayout().setFallbackRouteEnabled(false);
```

If the fallback mechanism is disabled, these links are not routed at all and remain in the same shape as before the layout. In Java code, you can retrieve the links that could not be routed in the usual way without the fallback mechanism.

Example of retrieving links without the fallback mechanism (Link Layout algorithm)
To retrieve the links that, without the fallback mechanism, could not be routed in the usual way :

In Java

```
Enumeration e = layout.getLongLinkLayout().getCalcFallbackLinks();
```

For instance, you can iterate over these links and apply your own specific fallback mechanism instead of the default fallback mechanism of the Long Link Layout algorithm.

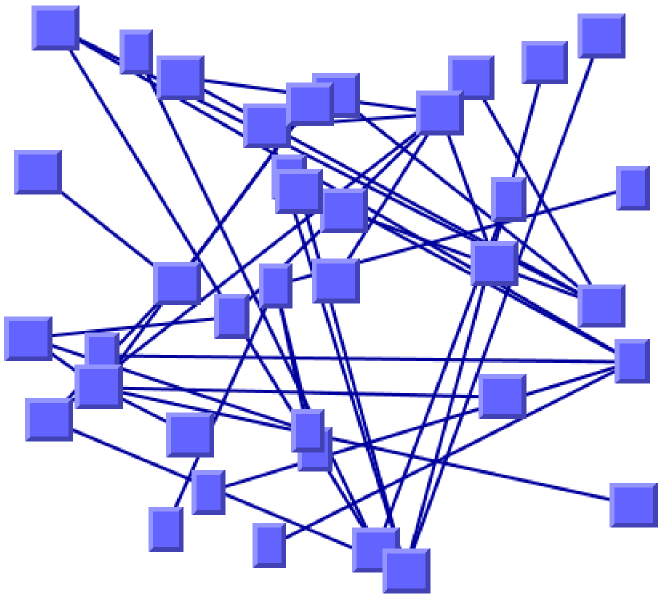
Random layout (RL)

Describes the *Random Layout* algorithm (class `IlvRandomLayout` from the package `ilog.views.graphlayout.random`).

In this section

RL sample

The following figure shows a sample drawing produced with the Random Layout (RL).



Graph drawing produced with the Random Layout

What types of graphs suit the RL?

Any type of graph:

- ♦ connected graphs and disconnected graphs
- ♦ planar graphs and nonplanar graphs

Features and limitations of the RL

Features

Random placement of the nodes of a grapher inside a given region.

Limitations

- The algorithm computes random coordinates for the upper-left corner of the graphic objects representing the nodes. In some cases, this may not be appropriate.
- To ensure that the nodes do not overlap the margins of the layout region, the algorithm computes the coordinates randomly inside a region whose width and height are smaller than the width and height of the layout region. The difference is the maximum width and the maximum height of the nodes, respectively. In some cases, this may not be appropriate.

The RL algorithm

The Random Layout (RL) algorithm is not really a layout algorithm. It simply places the nodes at randomly computed positions inside a user-defined region. Nevertheless, the Random Layout algorithm may be useful when a random, initial placement is needed by another layout algorithm or in cases where an aesthetic, readable drawing is not important.

Example of RL

In CSS

Below is a sample CSS specification using the Random Layout algorithm. The CSS specification can be loaded as style file into an application that uses the `IlvDiagrammer` class (see *Graph layout in IBM® ILOG® JViews Diagrammer*).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

GraphLayout {
    linkStyle            : "STRAIGHT_LINE_STYLE";
}
```

In Java™

The following code sample uses the `IlvRandomLayout` class. This code sample shows how to perform a Random Layout on a grapher directly without using a diagram component or any style sheet:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.random.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*  Fill in here the grapher with nodes and links in */

IlvRandomLayout layout = new IlvRandomLayout();
layout.attach(grapher);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic features and parameters of the RL

The `IlvRandomLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- *Layout region (RL)*
- *Percentage of completion calculation (RL)*
- *Preserve fixed links (RL)*
- *Preserve fixed nodes (RL)*
- *Random generator seed value (RL)*
- *Save Parameters to Named Properties (RL)*
- *Stop immediately (RL)*

The following sections describe the particular way in which these parameters are used by this subclass.

Layout region (RL)

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout region*.)

Percentage of completion calculation (RL)

The layout algorithm calculates the estimated percentage of completion. This value can be obtained from the layout report during the run of the layout. (For a detailed description of this features, see *Percentage of completion calculation* and *Graph layout event listeners*.)

Preserve fixed links (RL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.)

Preserve fixed nodes (RL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.)

Random generator seed value (RL)

The Random Layout uses a random number generator to compute the coordinates. You can specify a particular value to be used as a seed value. (See *Random generator seed value*) For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

Save Parameters to Named Properties (RL)

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*.)

Stop immediately (RL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the RL

Link style (RL)

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

Example of removing intermediate link points (RL algorithm)
To specify that the layout algorithm to automatically removes all the intermediate points of the links (if any):

In CSS
Add to the `GraphLayout` section:

```
linkStyle : "STRAIGHT_LINE_STYLE ";
```

In Java™
Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

♦ `IlvRandomLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

♦ `IlvRandomLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

Note: The layout algorithm may raise an `IlvInappropriateLinkException` if layout is performed on an `IlvGrapher`, but inappropriate link classes or link connector classes are used. See *Layout exceptions* for details and solutions to this problem.

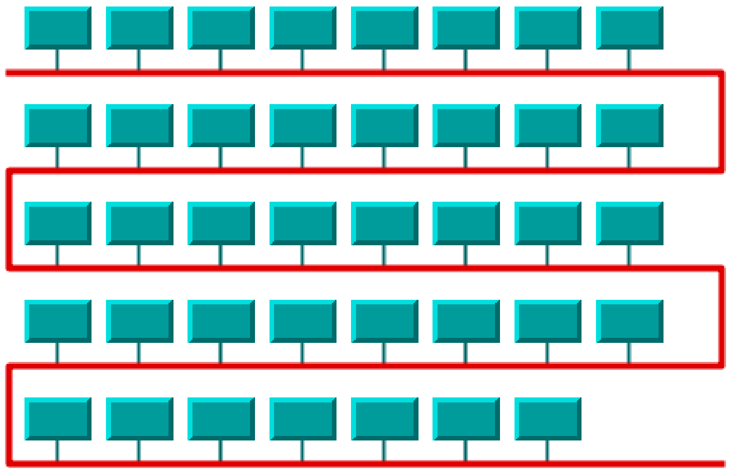
Bus layout (BL)

Describes the *Bus Layout* algorithm (class `IlvBusLayout` from the package `ilog.views.graphlayout.bus`).

In this section

BL - sample

The following figure shows a sample drawing produced with the Bus Layout (BL).


Bus topology produced with the Bus Layout

What types of graphs suit the BL?

♦ Bus network topologies (a set of nodes connected to a bus object)

Application domains for the BL

Application domains of the Bus Layout include:

♦ Telecom and networking (LAN diagrams)

♦ Electrical engineering (circuit block diagrams)

♦ Industrial engineering (equipment/resource control charts)

Features of the BL

- Displays bus topologies.
- Takes into account the size of the nodes so that no overlapping occurs.
- Provides several ordering, alignment, and flow direction options.
- Allows easy customization of the dimensional parameters.

The BL algorithm

Bus topology is well known in network management and telecommunications fields. The Bus Layout class can display these topologies nicely. It represents the “bus” as a “serpent” polyline. The width of the “serpent” is user-defined (via the width of the layout region parameter) and the height is computed so that enough space is available for all the nodes.

BL - CSS Sample

BL example
In CSS
Below is a sample CSS specification using the Bus Layout algorithm. Since the Bus Layout places nodes and reshapes the links, it is usually not necessary to specify an additional link layout in CSS. The CSS specification can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see *Graph layout in IBM® ILOG® JViews Diagrammer*.

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

GraphLayout {
    graphLayout   : "Bus";
    flowDirection : "LEFT_TO_RIGHT";
    nodeComparator: "DESCENDING_HEIGHT";
}
```

To be laid out by the Bus Layout, the graph needs to contain a bus node connected with links to several other nodes. When you specify the Bus Layout in CSS, the data model must respect this condition. Moreover, the CSS statements of the graphic objects used for the nodes must specify a graphic object implementing the `IlvPolyPointsInterface` to allow the Bus Layout to discover the bus node automatically when it is not explicitly specified (for details, see *Bus node (BL)*). Typically, you can do this by specifying a node rule in which the selector uses an attribute of the node in the data model that identifies the bus node.

In Java™
The following code sample uses the `IlvBusLayout` class. This code sample shows how to perform a Bus Layout on a grapher directly without using a diagram component or any style sheet:

```
 ...
import ilog.views.*;
import ilog.views.graphic.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.bus.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*  Fill in the grapher with nodes and links here */
```

```
/* Create the bus node; the number of points and
   the coordinates are not important */
IlvPoint[] points = {new IlvPoint(10, 10)};
IlvPolyline bus = new IlvPolyline(points);
grapher.addNode(bus, false);

 ... /* Fill in the grapher with links between each node
       and the bus here */

IlvBusLayout layout = new IlvBusLayout();
layout.attach(grapher);

/* Specify the bus node */
layout.setBus(bus);

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic features and parameters of the BL

The `IlvBusLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

♦ *Allowed time (BL)*

♦ *Layout of connected components (BL)*

♦ *Layout region (BL)*

♦ *Link clipping (BL)*

♦ *Preserve fixed links (BL)*

♦ *Preserve fixed nodes (BL)*

♦ *Stop immediately (BL)*

Extra feature for JViews Diagrammer:

♦ *Save parameters to named properties (BL)*

The following sections describe the particular way in which these parameters are used by this subclass.

Allowed time (BL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components (BL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*.)

Layout region (BL)

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass (See *Layout region*.)

The size of the layout is chosen with respect to the layout region (see *Dimensional Parameters for the Bus Layout Algorithm*). The height of the layout region is not taken into account. The height of the layout will be smaller or larger, depending on the number of nodes, the size of the nodes, and the other specified parameters.

Link clipping (BL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the Bus Layout* for details of the link clipping mechanism.

Preserve fixed links (BL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.)

Preserve fixed nodes (BL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.)

Save parameters to named properties (BL)

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*.)

Stop immediately (BL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the BL

The following parameters are specific to the `IlvBusLayout` class.

Order parameter (BL)

The order parameter specifies how to arrange the nodes.

Example of specifying node ordering option (BL algorithm)
To specify the ordering option for the nodes:

In CSS
Add to the GraphLayout section:

```
nodeComparator : "DESCENDING_HEIGHT";
```

In Java™
Use the method:

```
void setNodeComparator(Comparator comparator)
```

The valid values for `comparator` are:

♦ `IlvBusLayout.DESCENDING_HEIGHT`

  The nodes are ordered in the descending order of their height.

♦ `IlvBusLayout.ASCENDING_HEIGHT`

  The nodes are ordered in the ascending order of their height.

♦ `IlvBusLayout.DESCENDING_WIDTH`

  The nodes are ordered in the descending order of their width.

♦ `IlvBusLayout.ASCENDING_WIDTH`

  The nodes are ordered in the ascending order of their width.

♦ `IlvBusLayout.DESCENDING_AREA`

  The nodes are ordered in the descending order of their area.

♦ `IlvBusLayout.ASCENDING_AREA`

  The nodes are ordered in the ascending order of their area.

♦ `IlvBusLayout.ASCENDING_INDEX`

  The nodes are ordered in the ascending order of their index (see `setIndex(java.lang.Object, int)`).

♦ `IlvBusLayout.DESCENDING_INDEX`

The nodes are ordered in the descending order of their index (see `setIndex(java.lang.Object, int)`).

- ♦ `null`

  The nodes are ordered in an arbitrary way.

- ♦ Any other implementation of the Comparator interface.

  The nodes are ordered according to this custom comparator.

The default is `null`.

The ordering of the nodes starts at the upper-left corner of the bus.

Note that in incremental mode (see `setIncrementalMode(boolean)`) or when nodes are fixed (see `setFixed(java.lang.Object, boolean)`), the order is not guaranteed to obey the comparator, because it competes with the other constraints.

More about the ASCENDING_INDEX and DESCENDING_INDEX options (BL)

These options allow you to specify the order of the nodes according to a user-defined index value specified for each node. If this option is chosen, the algorithm sorts the nodes in ascending order according to their index values.

Example of specifying index options (BL algorithm)
The index is an integer value associated with a node. To specify the index:

In CSS
Write a rule to select the node:

```
#node1 {
  Index: "3";
}
```

In Java
Use the method:

```
void setIndex(Object node, int index)
```

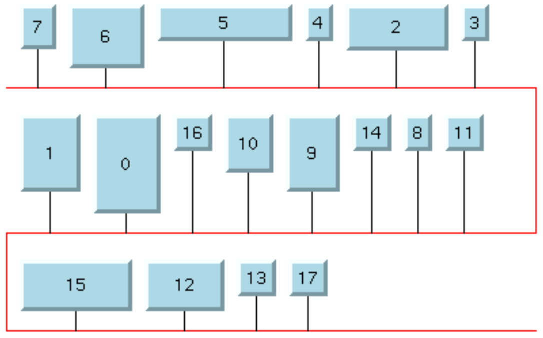
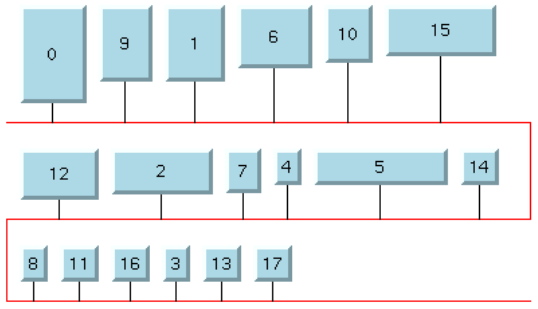
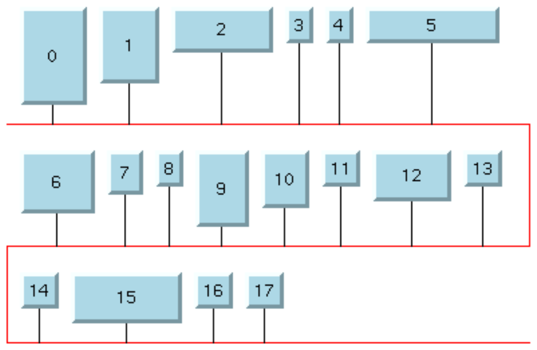
The values of the indices cannot be negative. To obtain the current index of a node, use the method:

```
int getIndex(Object node)
```

If no index is specified for the node, the value `IlvBusLayout.NO_INDEX` is returned.

The following table shows the ordering options for the Bus Layout algorithm.

Examples of ordering options for the nodes for the Bus Layout algorithm

Ordering	Layout
No order	 7 6 5 4 2 3 1 0 16 10 9 14 8 11 15 12 13 17
`DESCENDING_HEIGHT`	 0 9 1 6 10 15 12 2 7 4 5 14 8 11 16 3 13 17
`ASCENDING_INDEX`	 0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17

Bus node (BL)

To represent bus topologies, the algorithm reshapes a special node, called the “bus node”, and gives it a “serpent” form. This bus node must be an instance of the `IlvPolyPointsInterface` class. Usually, you use its subclass `IlvPolyline`. Before performing the layout, you must create this object and add it to the grapher as a node.

(The number of points in the object you create is not important.) Then, you must specify the node as “bus node” using the method:

```
void setBus(IlvPolyPointsInterface bus)
```

If none is specified, the Bus layout automatically tries to find an appropriate node that can be used as bus object.

The bus object must implement the interface `IlvPolyPointsInterface` and it must allow the insertion and removal of points (see the methods `allowsPointInsertion()` and `allowsPointRemoval()` defined by the interface). The initial number of points is not significant.

When a bus object is specified or automatically discovered in an `IlvGrapher`, the appropriate link connector is automatically installed on it. By default, the link connector is of type `IlvBusLinkConnector`.

Usually, the class `IlvPolyline` is used for the bus object. The bus object must be added to the `IlvGrapher` as a node (using the method `addNode(ilog.views.IlvGraphic, boolean)`). The links between the bus and the nodes connected to the bus must be created before performing the layout. (See the Java code provided in *BL - CSS Sample*.)

Link style (BL)

When the layout algorithm moves the nodes, straight-line links (such as instances of `IlvLinkImage`) will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can specify that the layout algorithm automatically removes all the intermediate points of the links (if any).

Example of specifying BL to automatically remove all intermediate points of the link (BL algorithm)
To specify that the layout algorithm automatically removes all the intermediate points of the links (if any):

In CSS
Add to the `GraphLayout` section:

```
linkStyle : "STRAIGHT_LINE_STYLE";
```

In Java
Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- ♦ `IlvBusLayout.NO_RESHAPE_STYLE`

  None of the links are reshaped in any manner.

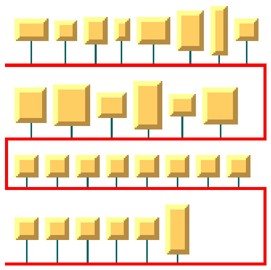
- ♦ `IlvBusLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

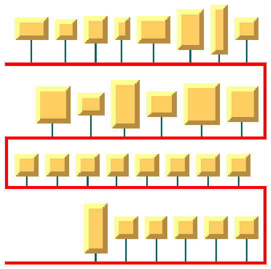
Note: The layout algorithm may raise an `IlvInappropriateLinkException` if layout is performed on an `IlvGrapher`, but inappropriate link classes or link connector classes are used. See *Layout exceptions* for details and solutions to this problem.

Flow direction (BL)

The flow direction options control the horizontal alignment of each row (bus level) with respect to the left and right sides of the layout region. The rows can be either all left-aligned on the left border of the layout region or can alternate between the left and right alignment.



Bus layout with left-to-right flow direction



Bus layout with alternate flow direction

Example of setting the flow direction (BL algorithm)
To set the flow direction:

In CSS
Add to the `GraphLayout` section:

```
flowDirection : "ALTERNATE";
```

In Java
Use the method:

```
void setFlowDirection(int direction);
```

The valid values for `direction` are:

- ♦ `IlvBusLayout.LEFT_TO_RIGHT` (the default)

  All the rows (bus levels) are left-aligned.

- ♦ `IlvBusLayout.ALTERNATE`

  The even rows (bus levels) are left-aligned and the odd rows are right-aligned.

Maximum number of nodes per level (BL)

By default, the layout places as many nodes on each level as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per level (see *Bus width adjusting disabled and bounded number of nodes per level* and *Bus width adjusting enabled and bounded number of nodes per level*).

Example of setting the maximum number of nodes per level (BL algorithm)
To set the maximum number of nodes per level:

In CSS
Add to the `GraphLayout` section:

```
maxNumberOfNodesPerLevel : "5";
```

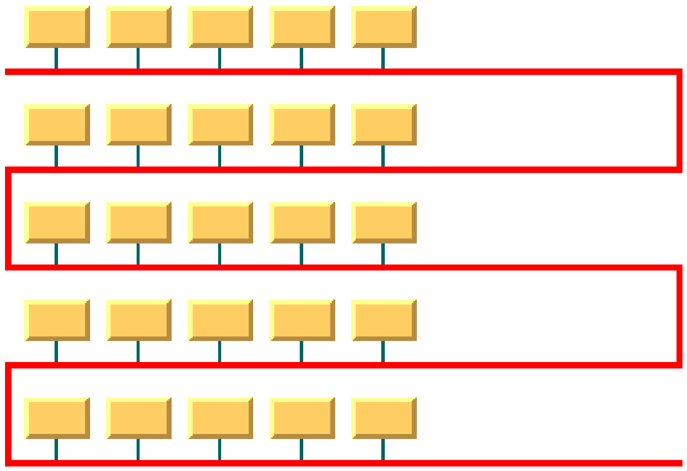
In Java
Use the method:

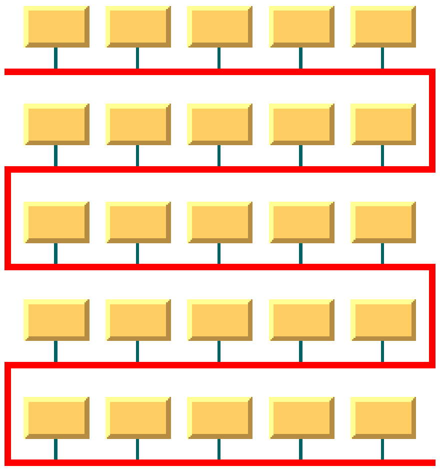
```
void setMaxNumberOfNodesPerLevel(int nNodes);
```

The default value is `Integer.MAX_VALUE`. This means that the number of nodes placed in each level is only bounded by the size of the nodes and the dimensional parameters. The specified value must be at least `1`.

Bus width adjusting (BL)

By default, the width of the bus object, that is the difference between the maximum and minimum x-coordinates, depends on the width of the layout region and the other dimensional parameters (see *Dimensional Parameters for the Bus Layout Algorithm*). Optionally, the width of the bus object can be automatically adjusted to the total width of the nodes, plus the offsets and the margins. This option can be particularly useful in conjunction with the customization of the maximum number of nodes per level (see *Maximum number of nodes per level (BL)*).


Bus width adjusting disabled and bounded number of nodes per level


Bus width adjusting enabled and bounded number of nodes per level

Example of enabling/disabling the bus width adjustment (BL algorithm)
To enable or disable bus width adjusting:

In CSS
Add to the graph layout section:

```
busWidthAdjustingEnabled: "true";
```

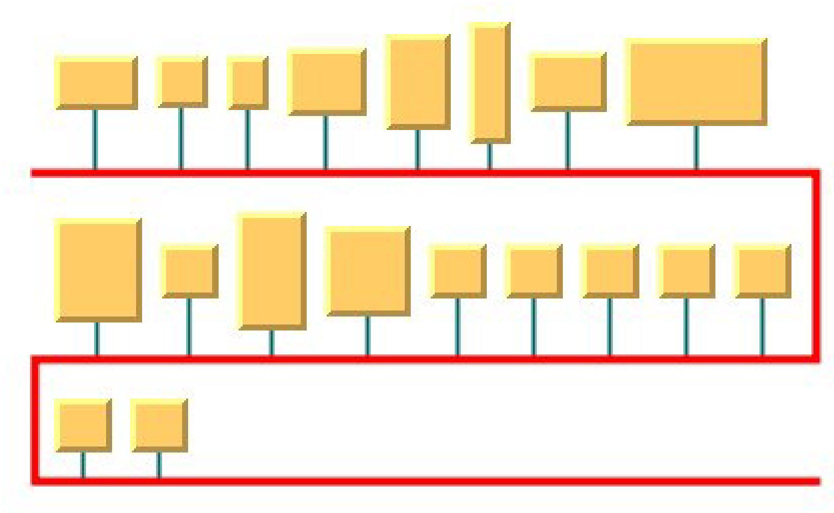
In Java
Use the method:

```
void setBusWidthAdjustingEnabled(boolean enable);
```

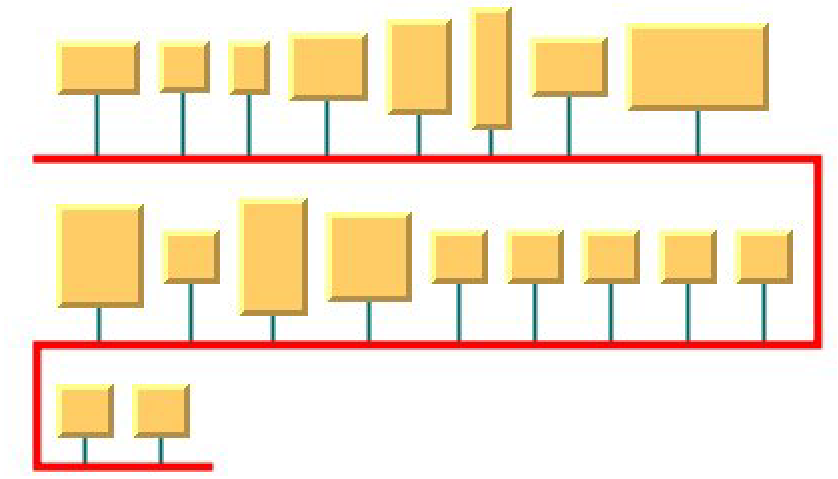
The bus width adjusting is disabled by default.

Bus line extremity adjusting (BL)

If necessary, the bus line can be adjusted to stop where the nodes stop (plus the margins). This can make a difference when there is only one horizontal bus line, or when the flow direction is `ALTERNATE`.



Bus Layout with bus line extremity disabled



Bus Layout with bus line extremity enabled

Example of enabling/disabling the bus line extremity adjustment (BL algorithm)
To enable or disable the adjustment of the bus line extremity:

In CSS
Add to the graph layout section:

```
busLineExtremityAdjustingEnabled: "true";
```

In Java
Use the method:

```
void setBusLineExtremityAdjustingEnabled (boolean enable);
```

The adjustment of the bus line extremity is disabled by default.

Alignment parameters (BL)

The alignment options control how a node is placed above its row (bus level). The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Global alignment parameters

Example of setting global alignment (BL algorithm)
To set the global alignment:

In CSS
Add to the `GraphLayout` section:

```
globalVerticalAlignment : "TOP";
```

In Java

Use the method:

```
void setGlobalVerticalAlignment(int alignment);
```

The valid values for `alignment` are:

- ♦ `IlvBusLayout.CENTER` (the default)

  The node is vertically centered over its row (bus level).

- ♦ `IlvBusLayout.TOP`

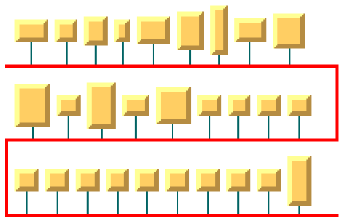
  The node is vertically aligned on the top of its row (bus level).

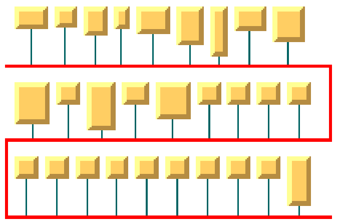
- ♦ `IlvBusLayout.BOTTOM`

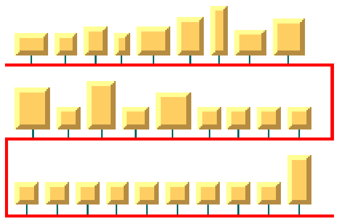
  The node is vertically aligned on the bottom of its row (bus level).

- ♦ `IlvBusLayout.MIXED`

  Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.


Bus Layout with center vertical alignment


Bus Layout with top vertical alignment


Bus Layout with bottom vertical alignment

Alignment of individual nodes

All nodes have the same alignment unless the global alignment is set to `IlvBusLayout.MIXED`. Only when the global alignment is set to `MIXED` can each node have an individual alignment style.

Example of setting the alignment of an individual node (BL algorithm)
To set the alignment of an individual node:

In CSS
Write a rule to select the node:

```
GraphLayout {
   globalVerticalAlignment : "MIXED";
}
#node1
{
  VerticalAlignment : "BOTTOM ";
}
```

In Java
Use the methods:

```
void setVerticalAlignment(Object node, int alignment);
```

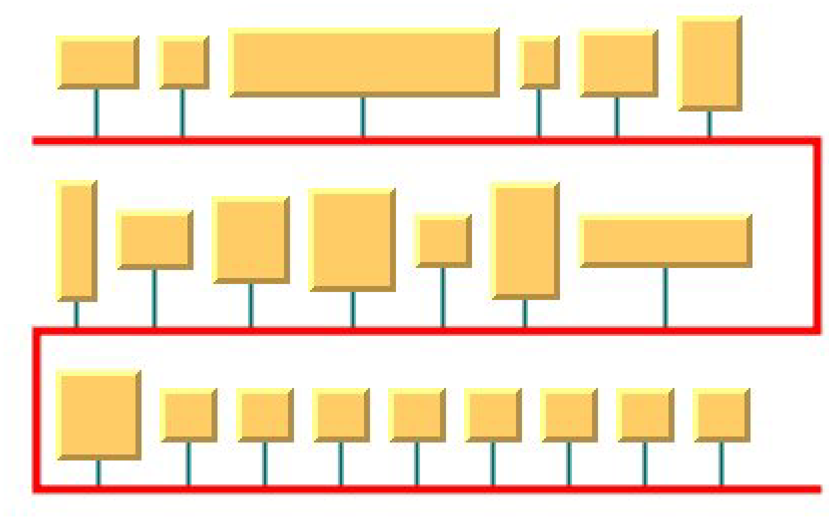
```
int getVerticalAlignment(Object node);
```

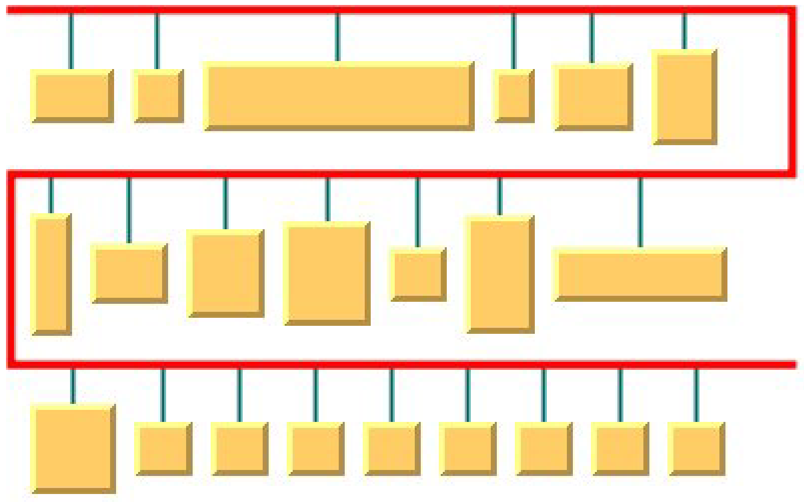
The valid values for node alignment are:

- `IlvBusLayout.CENTER` (the default)
- `IlvBusLayout.TOP`
- `IlvBusLayout.BOTTOM`

Node position (BL)

The nodes can be placed either above or below the corresponding bus line.


Bus Layout with nodes above the bus


Bus Layout with Nodes Below the Bus

Example of setting node position (BL algorithm)
To set the node position:

In CSS
Add to the GraphLayout section:

```
nodePosition : "NODES_BELOW_BUS";
```

In Java
Use the method:

```
void setNodePosition(int position);
```

The valid values for node positions are:

♦ `IlvBusLayout.NODES_ABOVE_BUS` (the default)

  The nodes are placed above the corresponding bus line.

♦ `IlvBusLayout.NODES_BELOW_BUS`

  The nodes are placed below the corresponding bus line.

Incremental mode (BL)

The Bus Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode. In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible

Example of enabling incremental mode (BL algorithm)
To enable the incremental mode:

In CSS
Add to the `GraphLayout` section:

```
incrementalMode : "true";
```

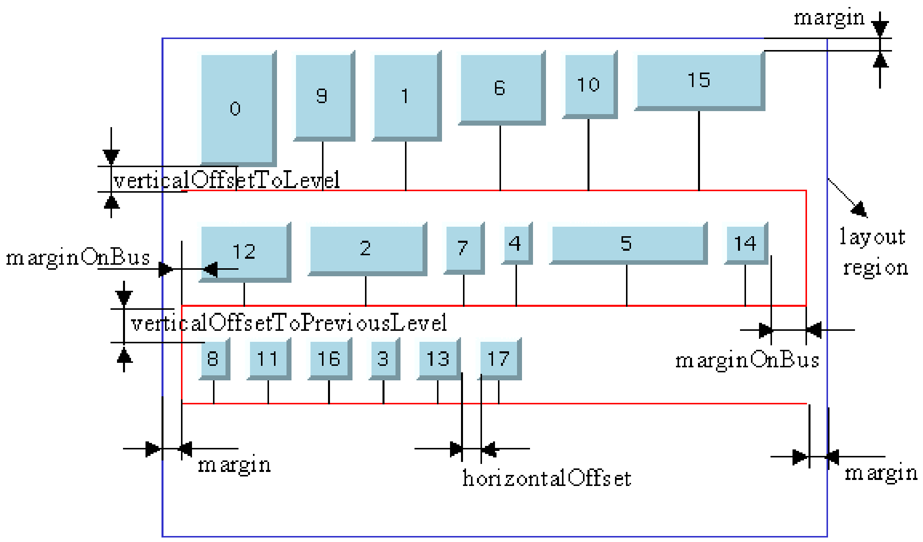
In Java
Call:

```
layout.setIncrementalMode(true);
```

Note: To preserve stability, the incremental mode can keep some regions free. Therefore, the total area of the layout can be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.

Dimensional parameters (BL)

Dimensional Parameters for the Bus Layout Algorithm illustrates the dimensional parameters used in the Bus Layout algorithm. These parameters are explained in the subsequent sections.



Dimensional Parameters for the Bus Layout Algorithm

Horizontal offset (BL)

This parameter represents the horizontal distance between two nodes.

Example of specifying the horizontal offset (BL algorithm)
To specify the horizontal offset:

In CSS
Add to the `GraphLayout` section:

```
horizontalOffset : "30.0";
```

In Java
Use the method:

```
void setHorizontalOffset(float offset)
```

Vertical offset to level (BL)

This parameter represents the vertical distance between a row of nodes and the next horizontal segment of the bus node.

Example of specifying vertical offset (BL algorithm)
To specify this parameter:

In CSS
Add to the `GraphLayout` section:

```
verticalOffsetToLevel : "40.0";
```

In Java
Use the method:

```
void setVerticalOffsetToLevel(float offset)
```

Vertical offset to previous level (BL)

Example of setting vertical offset to the previous level (BL algorithm)
To set the vertical offset to the previous level:

In Java
This parameter represents the vertical distance between a row of nodes and the previous horizontal segment of the bus node. To specify this parameter, use the method:

```
void setVerticalOffsetToPreviousLevel(float offset)
```

Margin (BL)

This parameter represents the offset distance between the layout region and the bounding rectangle of the layout.

Example of specifying the margin (BL algorithm)
To specify the margin:

In CSS
Add to the `GraphLayout` section:

```
margin : "5.0";
```

In Java
Use the method:

```
void setMargin(float margin)
```

Margin on bus (BL)

On the odd horizontal levels (first, third, fifth, and so on) of the bus, starting from the top, this parameter represents the offset distance between the left side of the first node on the left and the left side of the bus object.

On the even horizontal levels (second, fourth, sixth, and so on) of the bus, starting from the top, this parameter represents the offset distance between the right side of the last node on the right and the right side of the bus object. (See *Dimensional Parameters for the Bus Layout Algorithm* for an illustration of the margin-on-bus parameter.)

Example of specifying the margin on bus (BL algorithm)
To specify this parameter:

In CSS
Add to the `GraphLayout` section:

```
marginOnBus : "5.0";
```

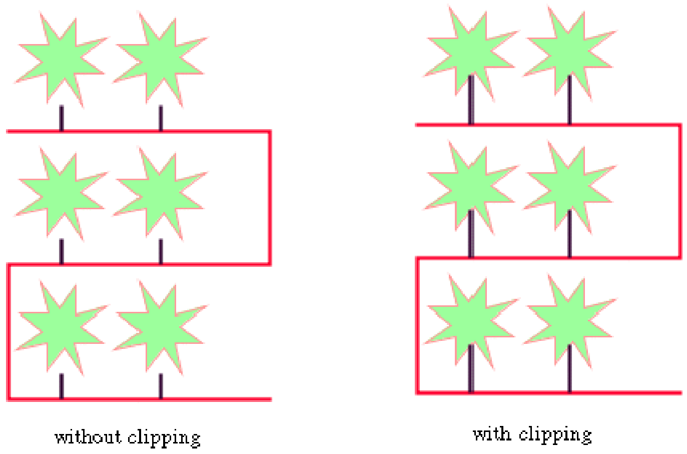
In Java
Use the method:

```
void setMarginOnBus(float margin)
```

Using a link clipping interface with the Bus Layout

By default, the Bus Layout does not place the connection points of links at the nodes. At the bus node, it installs a bus link connector that is responsible for the connection points. At the other nodes, it relies on their link connectors to determine the connection points. If no link connectors are installed at these nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes.

If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape. The following figure shows an example.



Effect of Link Clipping Interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*.

Example of setting a link clipping interface with the Bus Layout
To set a link clip interface:

In CSS
It is not possible to set the link clip interface.

In Java

Call:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: The link clip interface requires link connectors at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you use `IlvFreeLinkConnector` or `IlvClippingLinkConnector` for link connectors to be used in combination with `IlvGrapher` objects. The clip link connector updates the clipped connection points automatically during interactive node movements.

The special bus node is an exception: it always uses the bus link connector.

Circular layout (CL)

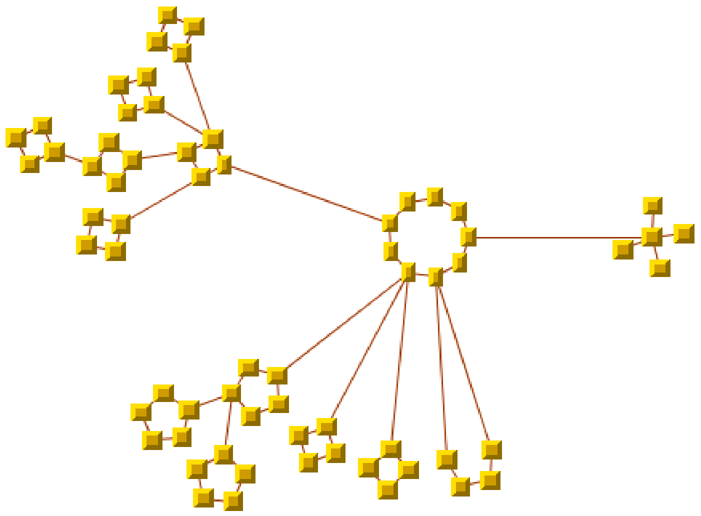
Describes the *Circular Layout* algorithm (class `IlvCircularLayout` from the package `ilog.views.graphlayout.circular`).

In this section

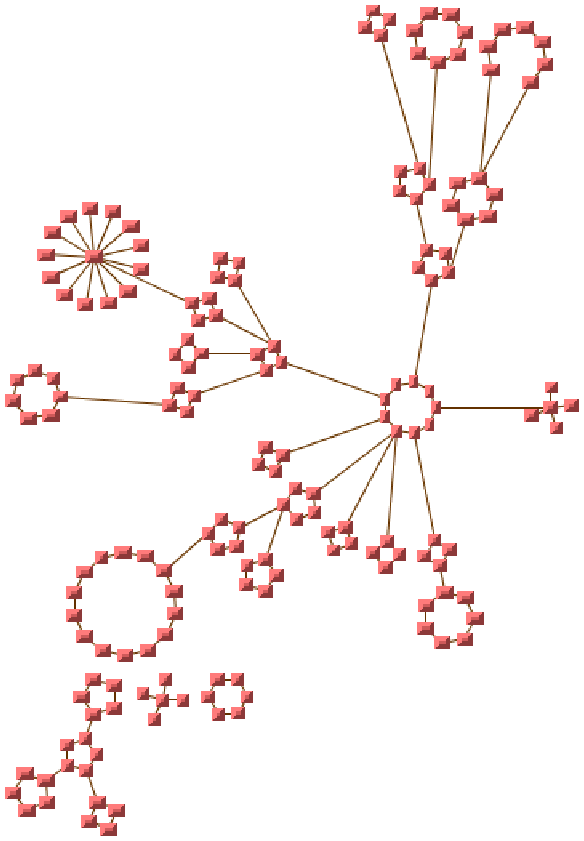
General information on the CL

CL samples

The following figures show sample drawings produced with the Circular Layout.



Ring-and-star topology drawing produced with the Circular Layout



Large ring-and-star topology drawing produced with the Circular Layout

What types of graphs suit the CL?

♦ Graphs representing interconnected ring and/or star network topologies

Application domains for the CL

Application domains for the Circular Layout include:

♦ Telecom and networking (LAN diagrams)

♦ Business processing (organization charts)

♦ Database and knowledge engineering (sociology, genealogy)

♦ The World Wide Web (Web hyperlink neighborhood)

Features and limitations of the CL

Features

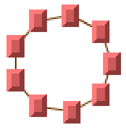
- ♦ Displays network topologies composed of interconnected rings and/or stars.
- ♦ Provides two clustering modes (see *Clustering mode (CL)*). The first mode lays out clusters as circles and places the clusters. This mode is designed for rings/stars that are interconnected in a tree structure, but it can produce acceptable results even if the graph contains cycles. The second mode lays out the clusters as circles of nodes, minimizing the link crossings while keeping the clusters at their initial position.
- ♦ Takes into account the size of the nodes so that no overlapping occurs. (See also *The CL algorithm*).

Limitations

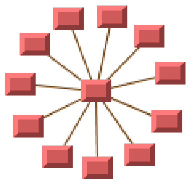
Link crossings cannot always be avoided.

The CL algorithm

Ring and star topologies are similar in several ways. Take a look at *Ring topology* and *Star topology* to get an idea of their similarities.



Ring topology

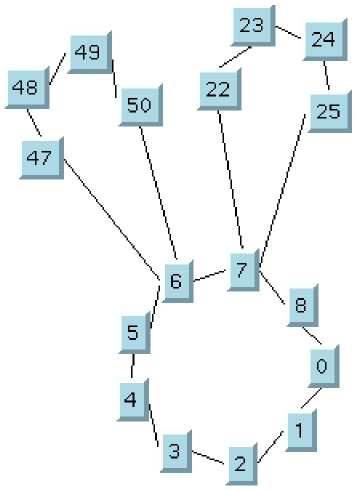


Star topology

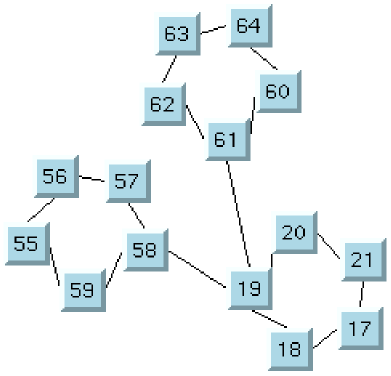
Both topologies are composed of nodes drawn on a circle. For the Circular Layout algorithm, the only difference between the ring and star topologies is that the star has a special node, called the star center, that is drawn at the center of the circle. The user must specify the node that is the star center. (See *Star center (CL)* for information on how to specify the node.)

For each ring or star (generically called a cluster), the Circular Layout algorithm, in one of its modes (see *Clustering mode (CL)*), allows you to specify the order of the nodes on the circle (this is discussed in *Cluster membership and order of the nodes on a cluster (CL)*). Otherwise, an arbitrary order is automatically chosen. In another mode, the order is computed automatically such that the number of link crossings is small.

The network topology can be composed of more than one ring or star. These rings and stars can be partially interconnected; that is, two or more clusters can have a common node as shown in *Rings interconnected by common nodes*. They can also be interconnected by links between nodes of two different clusters as shown in *Rings interconnected by links*.



Rings interconnected by common nodes



Rings interconnected by links

The Circular Layout algorithm lays out the ring/star topologies in a way that preserves the visual identity of each cluster and avoids overlapping nodes and clusters. (See the sample drawings in *CL samples*.)

To understand how the layout is performed in the clustering mode `BY_CLUSTER_IDS`, consider a graph in which each node represents a ring or star cluster of a network topology. Add a link between two nodes each time there is an interconnection between the corresponding clusters. The Circular Layout algorithm is designed for the case where the graph obtained in this manner is a tree (that is, a graph with no cycles). If cycles exist, the layout is performed using a spanning tree of the graph.

Starting from a root cluster (either a ring or a star), the clusters that are connected to the root cluster are drawn on a circle that is concentric to the root cluster. The radius of the circle is computed to avoid overlapping clusters. Next, the algorithm lays out the clusters connected to these last clusters on a larger circle, and so on. Each circle is called a level.

For networks that are not connected (that is, disconnected groups of clusters exist in the graph), more than one spanning tree exists. Each spanning tree is laid out separately and placed near the others. You can see this in the sample drawings in *CL samples*.

In the clustering mode `BY_SUBGRAPHS`, each subgraph (cluster) keeps its initial position. The subgraphs can be placed either by a different layout algorithm or interactively.

CL Example
In CSS
Below is a sample CSS specification using the Circular Layout algorithm. Since the Circular Layout places nodes and reshapes the links, it is usually not necessary to specify an additional link layout in CSS. The CSS specification can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see *Graph layout in IBM® ILOG® JViews Diagrammer*).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

GraphLayout {
    graphLayout    : "Circular";
    clusteringMode : "BY_CLUSTER_IDS";
    offset         : "20";
    levelOffset    : "30";
}
```

For a graph to be laid out by the Circular Layout in the clustering mode `BY_CLUSTER_IDS` (see *Clustering mode (CL)*), you need to specify the clustering information (see *Cluster membership and order of the nodes on a cluster (CL)*).

In Java™
Below is a code sample using the `IlvCircularLayout` class. This code sample shows how to perform a Circular Layout on a grapher directly, without using a diagram component or any style sheet:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.circular.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);
IlvCircularLayout layout = new IlvCircularLayout();
layout.attach(grapher);

 ... /* Fill in the grapher with nodes and links here */

// create identifier for cluster 0
IlvClusterNumber clusterId = new IlvClusterNumber(0);

// specify the cluster identifier for cluster 0
// Assume there are three nodes: node1, node2, node3
// the ordering of the nodes: node1 -> node2 -> node3
layout.setClusterId(node1, clusterId, 0); // index 0
layout.setClusterId(node2, clusterId, 1); // index 1
layout.setClusterId(node3, clusterId, 2); // index 2

// create identifier for cluster 1
clusterId = new IlvClusterNumber(1);

// specify the cluster identifier for cluster 1
// Assume there are three nodes: node4, node5, node6
// the ordering of the nodes: node4 -> node5 -> node6
layout.setClusterId(node4, clusterId, 1); // index 1
layout.setClusterId(node5, clusterId, 2); // index 2
layout.setClusterId(node6, clusterId, 0); // index 0

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic features and parameters of the CL

The `IlvCircularLayout` class supports the following parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- *Layout of connected components (CL)*
- *Layout region (CL)*
- *Link clipping (CL)*
- *Link connection box (CL)*
- *Preserve fixed links (CL)*
- *Preserve fixed nodes (CL)*
- *Stop immediately (CL)*

Extra feature for JViews Diagrammer:

- *Save parameters to named properties (CL)*

The following comments describe the particular way in which these parameters are used by this subclass.

Layout of connected components (CL)

The layout algorithm can utilize the generic mechanism to layout connected components. (For more information about this mechanism, see *Layout of connected components*).

Layout region (CL)

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

It is not possible to allow the user to control the size of the layout by specifying a bounding box for the drawing. The layout algorithm chooses the size to have enough space to avoid overlapping nodes and clusters.

The layout region setting (either your own or the default setting) is used only to choose the position of the center of the drawing. This means that only the center of the layout region is taken into consideration. All three ways to specify the layout region are available for this subclass. (See *Layout region*.)

Link clipping (CL)

The layout algorithm can use a link clip interface to clip the end points of a link. (See *Link clipping*.)

This is useful if the nodes have a nonrectangular shape such as a triangle, rhombus, or circle. If no link clip interface is used, the links are normally connected to the bounding boxes of the nodes, not to the border of the node shapes. See *Using a link clipping interface with the Circular Layout* for details of the link clipping mechanism.

Link connection box (CL)

The layout algorithm can use a link connection box interface (see *Link connection box*) in combination with the link clip interface. If no link clip interface is used, the link connection box interface has no effect. For details see *Using a link clipping interface with the Circular Layout*.

Preserve fixed links (CL)

The layout algorithm does not reshape the links that are specified as fixed. (See *Preserve fixed links*.)

Preserve fixed nodes (CL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.)

Save parameters to named properties (CL)

The layout algorithm is able to save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*.)

Stop immediately (CL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the CL

Clustering mode (CL)

The Circular Layout algorithm has two clustering modes.

Example of selecting a clustering mode (CL algorithm)
To select a clustering mode:

In CSS
Add to the `GraphLayout` section:

```
clusteringMode : "BY_SUBGRAPHS";
```

In Java™
Use the method:

```
void setClusteringMode(int mode);
```

The valid values for `mode` are:

- ♦ `IlvCircularLayout.BY_CLUSTER_IDS` (the default): Cluster identifiers need to be explicitly provided for each node (see *Cluster membership and order of the nodes on a cluster (CL)*). A tree-like algorithm places the clusters.
- ♦ `IlvCircularLayout.BY_SUBGRAPHS`: The algorithm handles a nested graph, including intergraph links. It arranges the nodes of each subgraph on a circle, so that the number of link crossings is small. It respects the intergraph links and rotates the cluster so that the number of link crossings is small. It assumes that all nodes are nearly square and that all nodes are in subgraphs, but the subgraph nesting is only 1. Nodes that are inside subgraphs of subgraphs are not handled. Note that in this mode each subgraph keeps its initial position. The subgraphs can be placed either by a different layout algorithm or interactively.

Cluster membership and order of the nodes on a cluster (CL)

This section applies only if the clustering mode is set to `BY_CLUSTER_IDS`.

Before performing the layout, you must specify to which cluster each node of the graph belongs.

Example of specifying node cluster (CL algorithm)
To specify to which cluster each node of the graph belongs:

In CSS
You must declare the clustering information in the node rules, like this:

```
node {
    ...
    ClusterId : "@clusterIds";
    StarCenter : "@starCenter";
}
```

where `clusterIds` refers to a node property whose value in the data model is given in the following syntax: `"<cluster-id>,<index>;<cluster-id>,<index>;..."`. Each pair of `cluster-id` and `index` represents the identifier of the cluster and the ordering index of the node in this cluster. The index can be omitted. If only one pair is specified, this means the node belongs to only one cluster. If several pairs are specified for a given node, this means that this node belongs to more than one cluster.

In the node rule above, `starCenter` is a node property whose value in the data model must be either “true” of “false”. The nodes for which the value is “true” are considered “star centers” by the layout algorithm (see *Star center (CL)*).

In Java
To specify the cluster membership, use a cluster identifier; that is, an instance of a subclass of the class `IlvClusterId` (which is abstract). Two subclasses are provided:

♦ `IlvClusterNumber`, which uses integer numbers as cluster identifiers.

♦ `IlvClusterName`, which uses string names as cluster identifiers.

You can combine these two types of identifiers as any other subclass of `IlvClusterId`. For example, you can write:

```
// create identifier for first cluster (integer)
IlvClusterNumber clusterId1 = new IlvClusterNumber(1);
// create identifier for second cluster (string)
IlvClusterNumber clusterId2 = new IlvClusterName("R&D network");
```

Then, if `node1` to `node3` belong to the first cluster, you can write:

```
layout.setClusterId(node1, clusterId1);
layout.setClusterId(node2, clusterId1);
layout.setClusterId(node3, clusterId1);
```

Assume `layout` is an instance of `IlvCircularLayout`.

If you want the nodes to be drawn in a special order (for example, `node1` -> `node2` -> `node3`), you should also specify an index (an integer value) for each node:

```
layout.setClusterId(node1, clusterId1, 0);
layout.setClusterId(node2, clusterId1, 1);
layout.setClusterId(node3, clusterId1, 2);
```

Two methods allow you to specify the cluster to which a node belongs:

```
void setClusterId(Object node, IlvClusterId clusterId)
```

```
void addClusterId(Object node, IlvClusterId clusterId)
```

If you call the first method, the node belongs only to the cluster whose identifier is `clusterId`. The second method allows you to specify that a node belongs to more than one cluster.

These methods have another version with an additional argument, an integer value representing the index:

```
void setClusterId(Object node, IlvClusterId clusterId, int index)
```

```
void addClusterId(Object node, IlvClusterId clusterId, int index)
```

This value is used to order the nodes on the cluster. If you specify these indices, the algorithm sorts the nodes in ascending order according to the index values.

Note that the values of the index cannot be negative. They do not need to be continuous; only the order of the values is important.

To obtain the current index of a node on a given cluster, use the method:

```
int getIndex(Object node, IlvClusterId clusterId)
```

If no index is specified for the node, the method returns the value `IlvCircularLayout.NO_INDEX`. It is a negative value.

To obtain an enumeration of the cluster identifiers for the clusters to which the node belongs, use the method:

```
Enumeration getClusterIds(Object node)
```

The elements of the enumeration are instances of a subclass of `IlvClusterId`.

To efficiently obtain the number of clusters to which a node belongs, use the method:

```
int getClusterIdsCount(Object node)
```

To remove a node from a cluster with a given identifier, use the method:

```
void removeClusterId(Object node, IlvClusterId clusterId)
```

To remove a node from all the clusters to which it belongs, use the method:

```
void removeAllClusterIds(Object node)
```

Star center (CL)

Example of specifying star center (CL algorithm)

To specify whether a node is the center of a star:

In CSS

Add to the `GraphLayout` section:

```
starCenter : "true";
```

In Java
Use the method:

```
void setStarCenter(Object node, boolean starCenter)
```

To know whether a node is the center of a star, use the method:

```
boolean isStarCenter(Object node)
```

By default, a node is not the center of a star.

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Root clusters (CL)

The algorithm arranges the clusters of each connected component of the graph of clusters around a “root cluster”. By default, the algorithm can choose this cluster. Optionally, you can specify one or more root clusters (one for each connected component).

Example of specifying root clusters (CL algorithm)
To specify one or more root clusters (one for each connected component):

In CSS
It is not possible to specify root clusters via CSS.

In Java
Use the methods:

```
void setRootClusterId(IlvClusterId clusterId)
```

To obtain an enumeration of the identifiers of the clusters that have been specified as root clusters, use the method:

```
Enumeration getRootClusterIds()
```

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Area minimization (CL)

For very large graphs, the radius of the concentric circles on which the clusters are placed can become very large. Therefore, the Circular Layout provides an optional mode that reduces the total area of the layout. To reduce the total area, the clusters are distributed more equally on the circle.

Example of specifying area minimization mode (CL algorithm)
To enable or disable the area minimization mode:

In CSS
Add to the `GraphLayout` section:

```
areaMinimizationEnabled : "true";
```

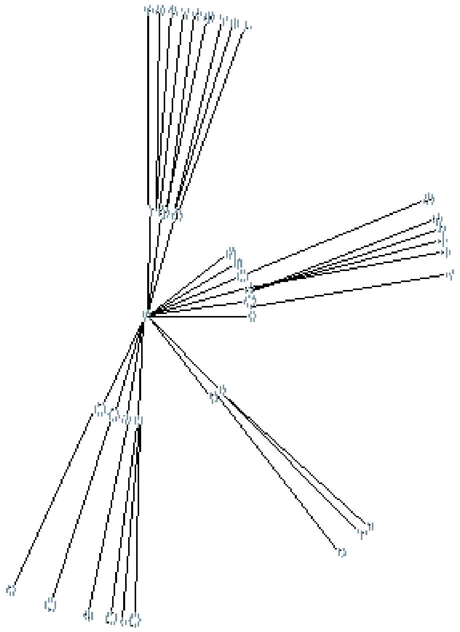
In Java
Use the method:

```
void setAreaMinimizationEnabled(boolean option)
```

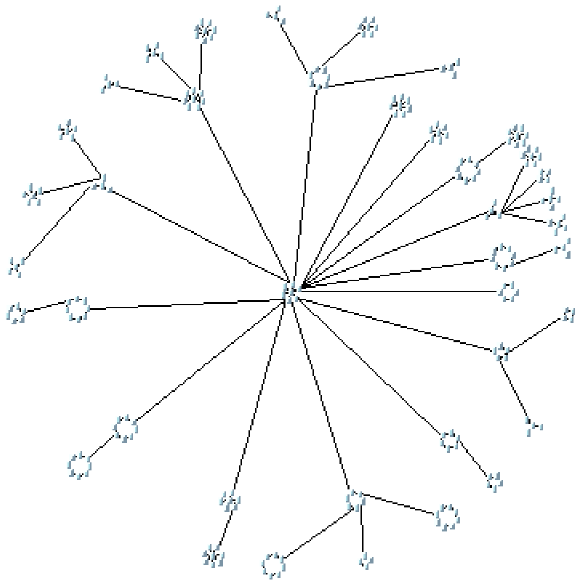
The default value is `false` (area minimization is disabled).

Deciding whether to enable the area minimization mode essentially depends on the size of the network. We recommend the area minimization mode for very large networks.

To get an idea of the difference between these modes, compare the following layouts of the same network:



Area minimization disabled (default)

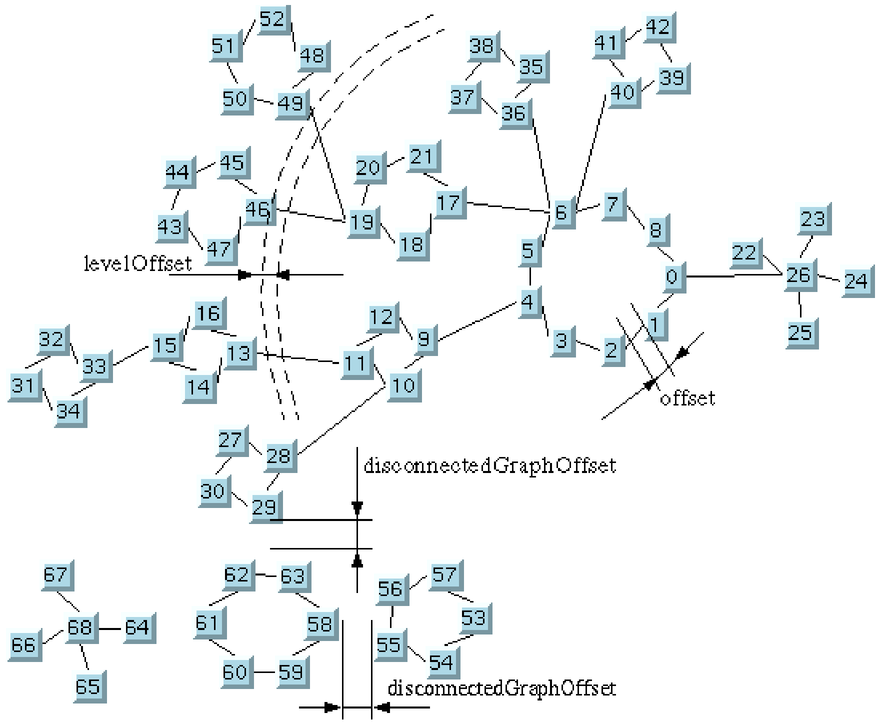


Area minimization enabled

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Dimensional parameters (CL)

Dimensional Parameters for the Circular Layout Algorithm illustrates the dimensional parameters used in the Circular Layout algorithm. These parameters are explained in the sections that follow.



Dimensional Parameters for the Circular Layout Algorithm

Offset (CL)

The layout algorithm tries to preserve a minimum distance between nodes (see *Dimensional Parameters for the Circular Layout Algorithm*).

Example of specifying the offset (CL algorithm)
To specify the offset:

In CSS
Add to the `GraphLayout` section:

```
offset = "20.0s";
```

In Java
Use the method:

```
void setOffset(float offset)
```

Level offset (CL)

If the clustering mode is `BY_SUBGRAPHS`, the level offset parameter controls the minimal offset between nodes that belong to the same cluster.

The following applies if the clustering mode is `BY_CLUSTER_IDS`.

As explained in *The CL algorithm*, interconnected rings and/or clusters are drawn on concentric circles around a root cluster. The radius of each concentric circle is computed to avoid overlapping clusters. In some cases, you may want to increase this radius to obtain a clearer drawing of the network. To meet this purpose, the radius is systematically increased with a “level offset” value (see *Dimensional Parameters for the Circular Layout Algorithm*).

Example of specifying the level offset (CL algorithm)
To specify the level offset:

In CSS
Add to the `GraphLayout` section:

```
levelOffset : "30.0";
```

In Java
Use the method:

```
void setLevelOffset(float offset)
```

The default value is zero.

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Disconnected graph offset (CL)

As explained in *The CL algorithm*, each connected component of the network is laid out separately and the drawing of each component is placed near the others (see *Dimensional Parameters for the Circular Layout Algorithm*).

Example of specifying the offset between each connected component (CL algorithm)
To specify the offset between each connected component:

In CSS
Add to the `GraphLayout` section:

```
disconnectedGraphOffset : "2.5";
```

In Java
Use the method:

```
void setDisconnectedGraphOffset(float offset)
```

This parameter has no effect if the clustering mode is `BY_SUBGRAPHS`.

Get the contents, the position, and the size of the clusters (CL)

At times, you might need to know the position and the size of the circle on which the nodes for each cluster are drawn. This may be the case if you want to perform some reshaping operations on the links. To do this, you can obtain a vector containing all the cluster identifiers after the layout is performed.

Example of obtain a vector containing all the cluster identifiers (CL algorithm)
To obtain a vector containing all the cluster identifiers after the layout is performed:

In CSS
It is not possible to get the contents, position, or size of the clusters via CSS.

In Java
Use the method:

```
Vector getClusterIds()
```

The vector contains instances of a subclass of `IlvClusterId`. By browsing the elements of this `Vector`, you can get the necessary information for each cluster:

```
float getClusterRadius(int clusterIndex)
```

```
IlvPoint getClusterCenter(int clusterIndex)
```

```
Vector getClusterNodes(int clusterIndex)
```

The `getClusterNodes` method returns the nodes that make up the cluster. The argument `clusterIndex` represents the position of the cluster in the `Vector` returned by the method `getClusterIds()`.

Do not use these methods if the clustering mode is `BY_SUBGRAPHS`.

Link style (CL)

When the layout algorithm moves the nodes, straight-line links, such as instances of `IlvLinkImage`, will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any).

Example of specifying automatic removal of all intermediate points of the links (CL algorithm)
To specify that the layout algorithm automatically removes all the intermediate points of the links (if any).:

In CSS
Add to the `GraphLayout` section:

```
linkStyle : "STRAIGHT_LINE_STYLE";
```

In Java

Use the method:

```
void setLinkStyle(int style)
```

The valid values for `style` are:

- ♦ `IlvCircularLayout.NO_RESHAPE_STYLE`

  None of the links is reshaped in any manner.

- ♦ `IlvCircularLayout.STRAIGHT_LINE_STYLE`

  All the intermediate points of the links (if any) are removed. This is the default value.

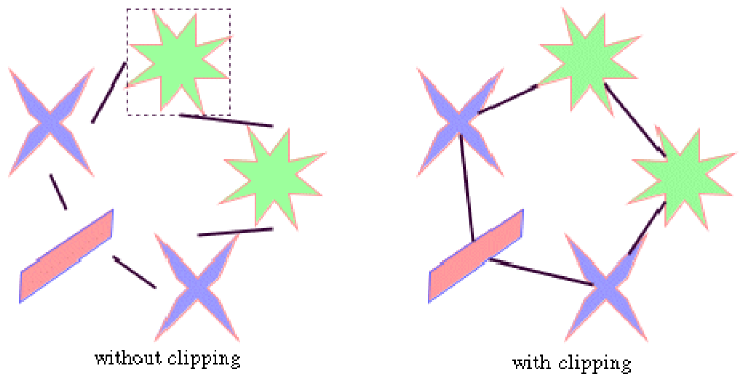
Note: The layout algorithm may raise an `IlvInappropriateLinkException` if layout is performed on an `IlvGrapher`, but inappropriate link classes or link connector classes are used. See *Layout exceptions* for details and solutions to this problem.

Using a link clipping interface with the Circular Layout

By default, the Circular Layout does not place the connection points of links. It relies on the link connectors of the nodes to determine the connection points. If no link connectors are installed at the nodes, the default behavior is to connect to a point at the border of the bounding box of the nodes.

If the node has a nonrectangular shape such as a triangle, rhombus, or circle, you may want the connection points to be placed exactly on the border of the shape. This can be achieved by specifying a link clip interface. The link clip interface allows you to correct the calculated connection point so that it lies on the border of the shape.

The following figure shows an example of link clipping.



Effect of link clipping interface

You can modify the position of the connection points of the links by providing a class that implements the `IlvLinkClipInterface`. An example for the implementation of a link clip interface is in *Link clipping*.

Example of setting a link clip interface (CL algorithm)
To set a link clip interface:

In Java
Use the method:

```
void setLinkClipInterface(IlvLinkClipInterface interface)
```

Note: The link clip interface requires link connectors at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you use `IlvFreeLinkConnector` or `IlvClippingLinkConnector` for link connectors to be used in combination with `IlvGrapher` objects. The clip link connector updates the clipped connection points automatically during interactive node movements.

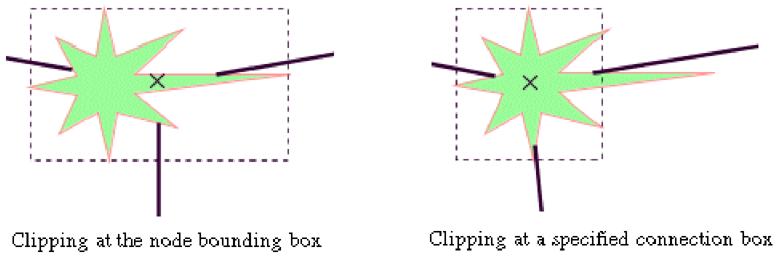
Link connection box (CL)

If a node has an irregular shape, the clipped links sometimes should not point towards the center of the node bounding box, but to a virtual center inside the node. You can achieve this by additionally providing a class that implements the `IlvLinkConnectionBoxInterface`. An example for the implementation of a link connection box interface is in *Link connection box*. To set a link connection box interface in Java, call:

```
void setLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface interface)
```

The link connection box interface is used only when link clipping is enabled by setting a link clip interface. If no link clip interface is specified, the link connection box interface has no effect.

The following figure shows an example of the combined effect.


Clipping at the node bounding box
Clipping at a specified connection box

Combined effect of link clipping interface and link connection box

If the links are clipped at the green irregular star node (previous figure, left), they do not point towards the center of the star, but towards the center of the bounding box of the node. This can be corrected by specifying a link connection box interface that returns a smaller node box than the bounding box (previous figure, right). Alternatively, the problem could be corrected by specifying a link connection box interface that returns the bounding box as the node box but with additional tangential offsets that shift the virtual center of the node.

Grid layout (GL)

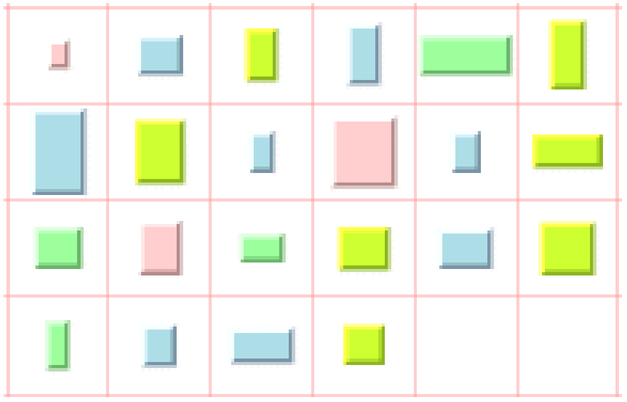
Describes the *Grid Layout* algorithm (class `IlvGridLayout` from the package `ilog.views.graphlayout.grid`).

In this section

General information on the GL

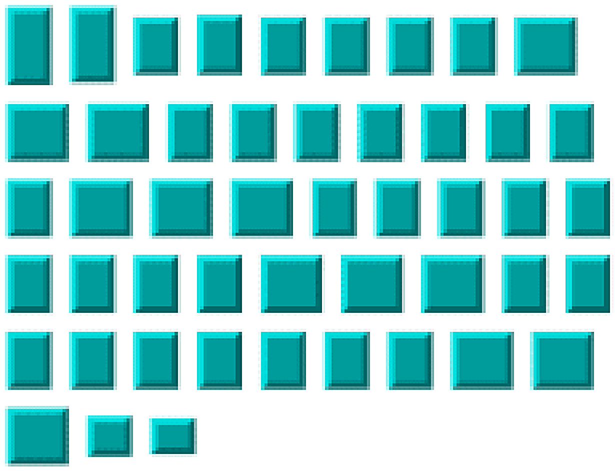
GL sample

The following sample drawings are produced with the Grid Layout (GL).



`TILE_TO_GRID_FIXED_WIDTH` mode with `CENTER` horizontal and vertical alignment

In `TILE_TO_GRID_FIXED_WIDTH` *mode with* `CENTER` *horizontal and vertical alignment,* the red lines are drawn to help identify the grid cells; they are not drawn by the layout algorithm.



`TILE_TO_ROWS` mode with `CENTER` vertical alignment.

What types of graphs suit the GL?

Any graph. However, the links are never taken into consideration. This algorithm is designed for placing nodes independently of their links, if they have any.

Application domains for the GL

Any domain where a collection of isolated nodes needs to be laid out.

Features of the GL

- Arranges a collection of isolated nodes or connected components.
- Takes into account the size of the nodes so that no overlapping occurs.
- Provides several alignment options and dimensional parameters.
- Provides full support for fixed nodes (overlapping of nonfixed nodes with fixed nodes is avoided).
- Provides an incremental mode which helps the retention of a mental map on incremental changes made to a collection of nodes.

The GL algorithm

The Grid Layout (GL) has two main modes: *grid* and *row/column*.

- ♦ In grid mode, the layout arranges the nodes of a graph in the cells of a grid (matrix). If a node is too large to fit in one grid cell (with margins), it occupies multiple cells. The size of the grid cells and the margins are parameters of the algorithm.
- ♦ In row/column mode, the layout arranges the nodes of a graph either by rows or by columns (according to the specified option). The width of the rows is controlled by the width of the layout region parameter. The height of the columns is controlled by the height of the layout region parameter. The horizontal and vertical margins between the nodes are parameters of the algorithm.

GL Example
In CSS
Below is a sample CSS specification using the Grid Layout algorithm. The CSS specification can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see *Graph layout in IBM® ILOG® JViews Diagrammer*).

```
SDM {
    GraphLayout : "true";
    LinkLayout  : "false";
}

GraphLayout {
    graphLayout                 : "Grid";
    layoutMode                  : "TILE_TO_GRID_FIXED_HEIGHT";
    globalHorizontalAlignment   : "LEFT";
    globalVerticalAlignment     : "TOP";
    incrementalMode             : "true";
    horizontalGridOffset        : "50";
    verticalGridOffset          : "70";
}
```

In Java™
The following code sample uses the `IlvGridLayout` class. This code sample shows how to perform a Grid Layout on a grapher directly without using a diagram component or any style sheet:

```
 ...
import ilog.views.*;
import ilog.views.graphic.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.grid.*;
 ...
IlvGrapher grapher = new IlvGrapher();
IlvManagerView view = new IlvManagerView(grapher);

 ... /*  Fill in the grapher with nodes and links here */

IlvGridLayout layout = new IlvGridLayout();
```

```
layout.attach(grapher);

try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Generic features and parameters of the GL

The `IlvGridLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class (see *Base class parameters and features*):

- ♦ *Allowed time (GL)*
- ♦ *Layout region (BL)*
- ♦ *Preserve fixed nodes (BL)*
- ♦ *Stop immediately (BL)*

Extra feature for JViews Diagrammer:

- ♦ *Save parameters to named properties (BL)*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed time (GL)

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed time*.) The result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout region (GL)

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (See *Layout region*.)

The layout region is considered differently depending on the layout mode. For details, see *Layout modes (GL)*.)

Preserve fixed nodes (GL)

The layout algorithm does not move the nodes that are specified as fixed. (See *Preserve fixed nodes*.) Moreover, nonfixed nodes are placed in such a manner that overlaps with fixed nodes are avoided.

Save parameters to named properties (GL)

The layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties* and *Saving layout parameters and preferred layouts*.)

Stop immediately (GL)

The layout algorithm stops after cleanup if the method `stopImmediately()` is called. (For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*.) If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters of the GL

Order parameter (GL)

The order parameter specifies how to arrange the nodes.

Example of specifying node placement iterations and allowed time (GL algorithm)
To specify the ordering option for the nodes:

In CSS
Add to the GraphLayout section:

```
nodeComparator : "DESCENDING_HEIGHT";
```

In Java™
Use the method:

```
void setNodeComparator(Comparator comparator)
```

The valid values for `comparator` are:

♦ AUTOMATIC_ORDERING

The algorithm is free to choose the order in such a way that it tries to reduce the total area occupied by the layout.

♦ NO_ORDERING

No ordering is performed.

♦ `DESCENDING_HEIGHT`

The nodes are ordered in the descending order of their height.

♦ `ASCENDING_HEIGHT`

The nodes are ordered in the ascending order of their height.

♦ `DESCENDING_WIDTH`

The nodes are ordered in the descending order of their width.

♦ `ASCENDING_WIDTH`

The nodes are ordered in the ascending order of their width.

♦ `DESCENDING_AREA`

The nodes are ordered in the descending order of their area.

♦ `ASCENDING_AREA`

The nodes are ordered in the ascending order of their area.

♦ `ASCENDING_INDEX`

The nodes are ordered in the ascending order of their index (see `setIndex(java.lang.Object, int)`).

♦ `DESCENDING_INDEX`

The nodes are ordered in the descending order of their index (see `setIndex(java.lang.Object, int)`).

♦ `null`

The nodes are ordered in an arbitrary way.

♦ Any other implementation of the `java.util.Comparator` interface.

The nodes are ordered according to this custom comparator.

The default is AUTOMATIC_ORDERING.

Note that in incremental mode (see `setIncrementalMode(boolean)`) and with fixed nodes (see `setFixed(java.lang.Object, boolean)`), the order of the nodes is not completely preserved.

Note also that, if the layout mode is `TILE_TO_GRID_FIXED_WIDTH` or `TILE_TO_GRID_FIXED_HEIGHT`, the order options are applied only for nodes whose size (including margins) is smaller than the grid cell size (see `setHorizontalGridOffset(float)` and `setVerticalGridOffset(float)`).

Layout modes (GL)

The Grid Layout algorithm has four layout modes.

Example of selecting a layout mode (GL algorithm)
To To select a layout mode:

In CSS
Add to the `GraphLayout` section:

```
layoutMode : "TILE_TO_GRID_FIXED_HEIGHT";
```

In Java
Use the method:

```
void setLayoutMode(int mode);
```

The valid values for `mode` are:

♦ `IlvGridLayout.TILE_TO_GRID_FIXED_WIDTH` (the default).

The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of columns. This number is equal to the width of the layout region parameter divided by the horizontal grid offset.

♦ `IlvGridLayout.TILE_TO_GRID_FIXED_HEIGHT`

The nodes are placed in the cells of a grid (matrix) that has a fixed maximum number of rows. This number is equal to the height of the layout region parameter divided by the vertical grid offset.

- `IlvGridLayout.TILE_TO_ROWS`

  The nodes are placed in rows. The maximum width of the rows is equal to the width of the layout region parameter. The height of the row is the maximum height of the nodes contained in the row (plus margins).

- `IlvGridLayout.TILE_TO_COLUMNS`

  The nodes are placed in columns. The maximum height of the columns is equal to the height of the layout region parameter. The width of the column is the maximum width of the nodes contained in the column (plus margins).

Alignment parameters (GL)

Global alignment parameters

The alignment options control how a node is placed over its grid cell or over its row or column (depending on the layout mode). The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

Example of setting global alignment (GL algorithm)
To set the global alignment:

In CSS
Add to the `GraphLayout` section:

```
globalHorizontalAlignment : "LEFT";
globalVerticalAlignment : "TOP";
```

In Java
Use the following methods:

```
void setGlobalHorizontalAlignment(int alignment);
```

```
void setGlobalVerticalAlignment(int alignment);
```

The valid values for the alignment parameter are:

- `IlvGridLayout.CENTER` (the default)

  The node is horizontally and/or vertically centered over its grid cell or row or column.

- `IlvGridLayout.TOP`

  The node is vertically aligned on the top of its cell(s) or row. Not used if the layout mode is `TILE_TO_COLUMNS`.

- `IlvGridLayout.BOTTOM`

The node is vertically aligned on the bottom of its grid cell(s) or row. Not used if the layout mode is `TILE_TO_COLUMNS`.

- `IlvGridLayout.LEFT`

  The node is horizontally aligned on the left of its grid cell(s) or column. Not used if the layout mode is `TILE_TO_ROWS`.

- `IlvGridLayout.RIGHT`

  The node is horizontally aligned on the right of its grid cell(s) or column. Not used if the layout mode is `TILE_TO_ROWS`.

- `IlvGridLayout.MIXED`

  Each node can have a different alignment. The alignment of each individual node can be set with the result that different alignments can occur in the same graph.

Alignment of individual nodes

All nodes have the same alignment unless the global alignment is set to `IlvGridLayout.MIXED`. Only when the global alignment is set to mixed can each node have an individual alignment style.

Example of setting alignment of individual nodes (GL algorithm)

To set and retrieve the alignment of an individual node:

In CSS

Write a rule that selects the node, for instance:

```
GraphLayout {
   globalVerticalAlignment : "MIXED";
}
#node1{
  VerticalAlignment : "BOTTOM";
}
```

In Java

In Java

Use the following methods:

```
void setHorizontalAlignment(Object node, int alignment);
```

```
void setVerticalAlignment(Object node, int alignment);
```

```
int getHorizontalAlignment(Object node);
```

```
int getVerticalAlignment(Object node);
```

The valid values for the alignment parameter are:

- `IlvGridLayout.CENTER` (the default)
- `IlvGridLayout.TOP`
- `IlvGridLayout.BOTTOM`
- `IlvGridLayout.LEFT`
- `IlvGridLayout.RIGHT`

Maximum number of nodes per row or column (GL)

By default, in `IlvGridLayout.TILE_TO_ROWS` or `IlvGridLayout.TILE_TO_COLUMNS` mode, the layout places as many nodes on each row or column as possible given the size of the nodes and the dimensional parameters (layout region and margins). If needed, the layout can additionally respect a specified maximum number of nodes per row or column.

Example of specifying the maximum number of nodes per row or column (GL algorithm)
To set the maximum number of nodes per row or column:

In CSS
Add to the `GraphLayout` section:

```
maxNumberOfNodesPerRowOrColumn : "8";
```

In Java
Use the method:

```
void setMaxNumberOfNodesPerRowOrColumn(int nNodes);
```

The default value is `Integer.MAX_VALUE`, that is, the number of nodes placed in each row or column is bounded only by the size of the nodes and the dimensional parameters. The specified value must be at least `1`. The parameter has no effect if the layout mode is `IlvGridLayout.TILE_TO_GRID_FIXED_WIDTH` or `IlvGridLayout.TILE_TO_GRID_FIXED_HEIGHT`.

Incremental mode (GL)

The Grid Layout algorithm normally places all the nodes from scratch. If the graph incrementally changes because you add, remove, or resize nodes, the subsequent layout may differ considerably from the previous layout. To avoid this effect and to help the user to retain a mental map of the graph, the algorithm has an incremental mode. In incremental mode, the layout tries to place the nodes at the same location or in the same order as in the previous layout whenever it is possible.

Example of enabling the incremental mode (GL algorithm)
To enable the incremental mode:

In CSS
Add to the `GraphLayout` section:

```
incrementalMode : "true";
```

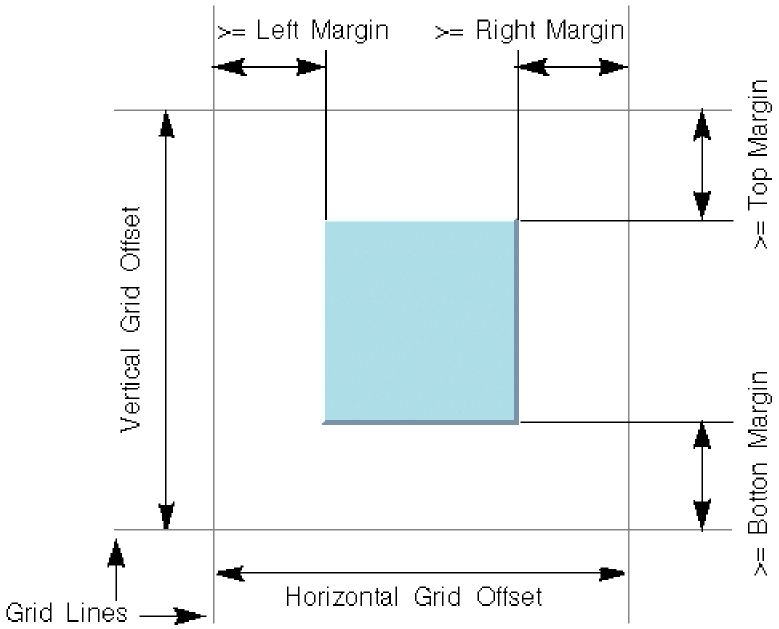
In Java
Call the method `setIncrementalMode(boolean)` as follows:

```
layout.setIncrementalMode(true);
```

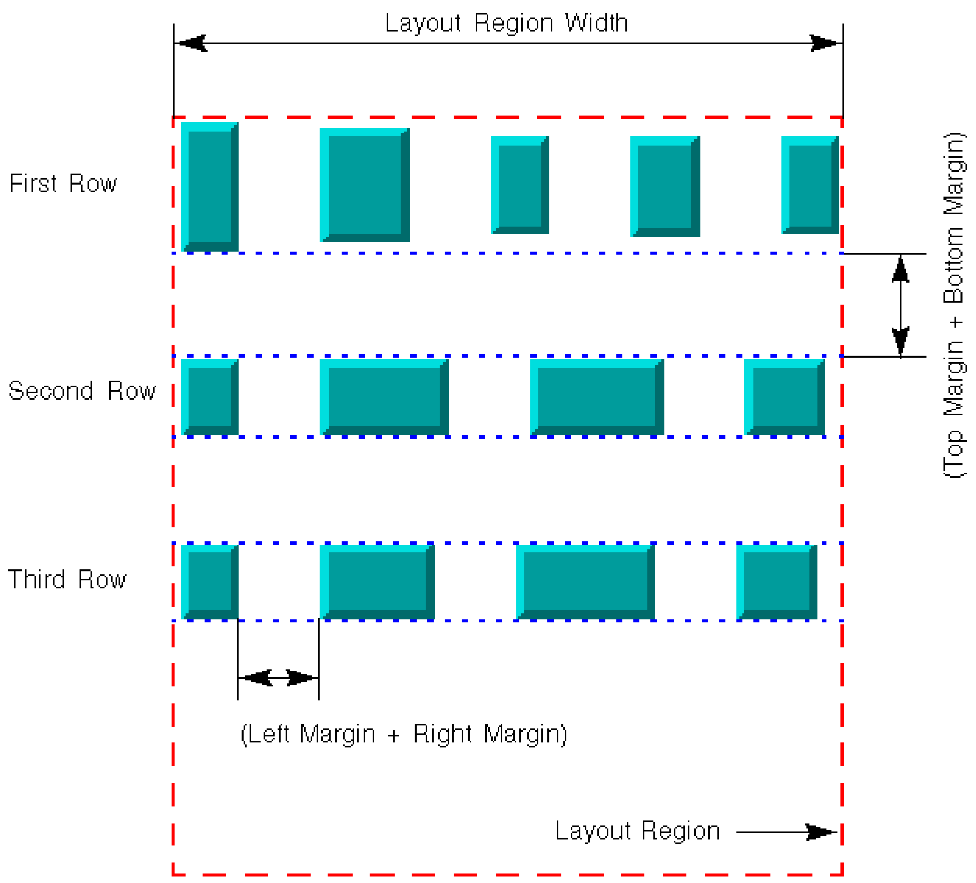
Note: To preserve the stability, the incremental mode may keep some regions free. Therefore, the total area of the layout may be larger than in nonincremental mode, and, in general, the layout may not look as nice as in nonincremental mode.

Dimensional parameters (GL)

Dimensional parameters for the grid mode of the Grid Layout algorithm and *Dimensional parameters for the row/column mode of the Grid Layout algorithm* illustrate the dimensional parameters used in the Grid Layout algorithm. These parameters are explained in the sections that follow.



Dimensional parameters for the grid mode of the Grid Layout algorithm



Dimensional parameters for the row/column mode of the Grid Layout algorithm

Grid offset (GL)

The grid offset parameters control the spacing between grid lines. It is taken into account only by the grid mode (layout modes `TILE_TO_GRID_FIXED_WIDTH` and `TILE_TO_GRID_FIXED_HEIGHT`).

Example of setting grid offset (GL algorithm)
To set the horizontal and vertical grid offset:

In CSS
Add to the `GraphLayout` section:

```
horizontalGridOffset : "50.0";
```

```
verticalGridOffset : "70.0";
```

In Java
Use the methods:

```
void setHorizontalGridOffset(float offset);
```

```
void setVerticalGridOffset(float offset);
```

The grid offset is the critical parameter for the grid mode. If the grid offset is larger than the size of the nodes (plus margins), an empty space is left around the node. If the grid offset is smaller than the size of the nodes (plus margins), the node will need to be placed on more than one grid cell. The best choice for the grid offsets depends on the application. It can be computed according to either the maximum size of the nodes (plus margins) or the medium size, and so on. Of course, if all the nodes have a similar size, the choice is straight-forward.

Margins (GL)

The margins control the space around each node that the layout algorithm keeps empty.

Example of specifying margins (GL algorithm)
To set the margins:

In CSS
Add to the `GraphLayout` section:

```
topMargin    : "6.0";
bottomMargin : "6.0";
leftMargin   : "4.0";
rightMargin  : "4.0";
```

In Java
Use the methods:

```
void setTopMargin(float margin);
```

```
void setBottomMargin(float margin);
```

```
void setLeftMargin(float margin);
```

```
void setRightMargin(float margin);
```

The meaning of the margin parameters is not the same for the grid modes as for the row/column modes.:

- In grid modes, they represent the minimum distance between the node border and the grid line (see *Dimensional parameters for the grid mode of the Grid Layout algorithm*.)
- In row/column modes, they are used to control the vertical distance between the rows or the horizontal distance between the columns and the horizontal or vertical minimal distance between the nodes in the same row or column (see *Dimensional parameters for the row/column mode of the Grid Layout algorithm*).

The default value for all the margin parameters is `5`.

Layout exceptions

Inappropriate-graph exception

Some layout algorithms can only deal with a specific type of graph.If the layout is performed with an inappropriate graph, an exception of type `IlvInappropriateGraphException` is thrown. However, this exception type occurs rather seldom, because most layout algorithms try to work silently in the best possible way with inappropriate graphs. For instance, the Tree Layout will silently handle graphs that are not trees without throwing this exception. The Tree Layout will in this case consider a spanning tree of the input graph for the layout.

The error handling differs depending on whether you use a diagram component with CSS styling, or whether you call layout in Java™ directly.

Example of inappropriate-graph exception

In CSS

The graph layout renderer catches the exception silently and logs it to the logger `ilog.views.sdm.renderer.graphlayout`. See

http://java.sun.com/javase/6/docs/technotes/guides/logging

to learn more about the Java logging facilities available since JDK 1.4. Usually, it is more convenient not to receive this exception. However, if you want to receive it, add to the GraphLayout and LinkLayout sections of your style sheet:

```
graphLayoutExceptionPassedOn: "true";
```

In this case, all graph layout exceptions are converted into a run-time exception and re-thrown by the graph layout renderer.

In Java

You have to catch the exception yourself and handle the error or report the error to the user in an suitable way. Example:

```
try {
 layout.performLayout();
} catch (IlvInappropriateGraphException ex) {
  ... handle the exception here ...
}
```

Inappropriate-link exception

This exception indicates that a particular type of link or link connector cannot be used for the layout algorithm. In general, the following link types can be used safely with all layout algorithms:

♦ `IlvPolylineLinkImage`

♦ `IlvEnhancedPolylineLinkImage`

♦ `IlvSplineLinkImage`

- `IlvSimpleLink`
- `IlvGeneralLink`

The following link connector types can be used with all layout algorithms.

- `IlvFreeLinkConnector`
- `IlvSDMFreeLinkConnector`
- `IlvClippingLinkConnector`.

Link connectors of type `IlvPinLinkConnector` can be used only in the following situations:

How to use IlvPinLinkConnector

When?	With which layout?
Always	Grid Layout Hierarchical Layout Link Layout Random Layout
Only if no link clip interface is provided	Bus Layout Circular Layout Topological Mesh Layout Uniform Length Edges Layout
Never	Tree Layout

See *Link clipping* for details.

Link connectors of other types can sometimes be used with some layouts. However, it is recommended to use only the link connectors listed for Hierarchical Layout, Tree Layout and Link Layout.

The error handling differs depending on whether you use a diagram component with CSS styling, or work directly in Java.

Example of Inappropriate-link exception

In CSS

By default, the graph layout renderer installs appropriate links and link connectors automatically when layout is performed. It internally calls the method `EnsureAppropriateLinks` and replaces the inappropriate links and link connectors by instances of `IlvPolylineLinkImage` and `IlvSDMFreeLinkConnector`. However it is recommended to specify appropriate link classes and link connector classes in CSS right from the beginning, because the link replacement is time-consuming and the result is sometimes confusing.

If you need to disable the automatic handling of inappropriate links and link connectors, add to the GraphLayout and LinkLayout sections of the style sheet:

```
ensureAppropriateLinks: "false";
```

In this case, the graph layout renderer catches the exception silently and logs it to the logger `ilog.views.sdm.renderer.graphlayout`, but it does not replace any links or link connectors. See

http://java.sun.com/javase/6/docs/technotes/guides/logging

to learn more about the Java logging facilities available since JDK 1.4. Usually, it is more convenient not to receive this exception. However, if you still want to receive it, add in the GraphLayout and LinkLayout sections of your style sheet:

```
graphLayoutExceptionPassedOn: "true";
```

In this case, all graph layout exceptions are converted into a run-time exception and re-thrown by the graph layout renderer.

In Java
If you are not sure whether the link types are correct for a given layout, you can call the method

```
IlvGraphLayoutUtil.EnsureAppropriateLinkTypes(
                               IlvGrapherAdapter grapherAdapter,
                               IlvGraphLayout layout,
                               boolean toStraightLine,
                               boolean traverse,
                               boolean interGraphLinks,
                               boolean redraw)
```

This method analyzes the graph and replaces inappropriate links by new instances of `IlvPolylineLinkImage`.

If you are not sure whether the link connectors are correct for a given layout, you can call the method

```
IlvGraphLayoutUtil.EnsureAppropriateLinkConnectors(
                               IlvGrapherAdapter grapherAdapter,
                               IlvGraphLayout layout,
                               boolean moveableConnectionPoints,
                               boolean traverse,
                               boolean redraw)
```

This method analyzes the graph and replaces inappropriate link connectors by new instances of `IlvFreeLinkConnector`.

If you want to do both at the same time, you can call the method

```
IlvGraphLayoutUtil.EnsureAppropriateLinks(IlvGraphLayout layout,
                                                     boolean redraw)
```

This method analyzes the graph and replaces inappropriate links and link connectors of the graph that is attached to the layout.

If the layout fails with an inappropriate-link exception, you can fix the situation quite easily, as demonstrated in the following code:

```
try {
  layout.performLayout();
} catch (IlvInappropriateLinkException ex) {
  IlvGraphLayoutUtil.EnsureAppropriateLinks(ex, redraw);
  // and now, try layout a second time:
  try {
    layout.performLayout();
  } catch (IlvGraphLayoutException ex2) {
  }
}
```

The class `IlvGraphLayoutUtil` provides further variants of the “`Ensure`...” methods. See the *Java API Reference Manual* for more information.

Nested layouts

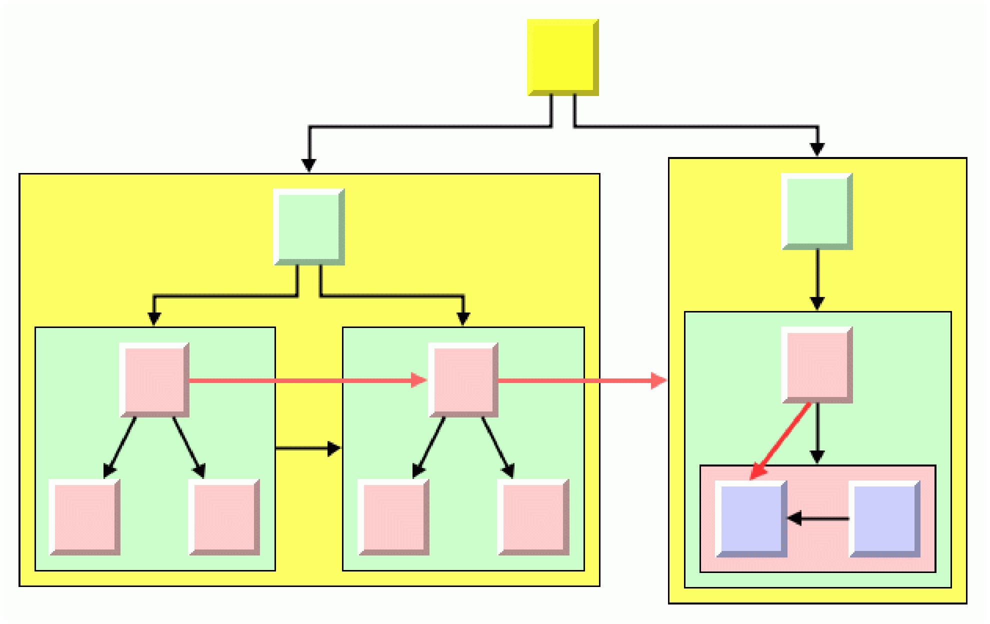
Describes how to perform a layout on a nested graph and explains the utilities that are available for nested graphs.

In this section

Concepts for nested layouts

IBM® ILOG® JViews Diagrammer supports nested graphs, that is, it can render graphs containing nodes that are graphs.

The following figure shows an example of a nested graph.



Example of a Nested Graph

A graph that is a node in another graph is called a subgraph. Links that connect nodes of different subgraphs are called subgraph links. The red links in the figure are intergraph links.

Layout of nested graphs in IBM® ILOG® JViews Diagrammer

Describes how to use styling and SDM renderers with nested graphs to perform graph layout.

In this section

Nested SDM models and nested graphers

IBM® ILOG® JViews Diagrammer uses an SDM data model that specifies the application objects (also called model objects). The data model contains nested graphs if any model objects have a parent-child relationship. The parent-child relationship is expressed in the interface `IlvSDMModel` by the following two methods:

```
Enumeration getChildren(Object parent);
```

```
Object getParent(Object parent);
```

If for any model node, the call `model.getChildren(node)` returns a non-null value, then this model node is a subgraph and all its children must return this node as their parent. In this case, the entire SDM model is called *nested*.

The `SubGraph` renderer translates a nested SDM model into a display of nested graphers. A nested grapher is an instance of `IlvGrapher` that contains other instances of `IlvGrapher` as graphic nodes. A subgrapher can be expanded (its inner node and links are shown) or collapsed (its contents is hidden and the collapsed subgrapher looks like a normal node). The `SubGraph` renderer is responsible only for creating the nested graphers and managing the collapse/expand state, but it does not perform any layout on the nested graphers. For more details on the `SubGraph` renderer, see The SubGraph renderer in *JViews Diagrammer SDK*.

Specification in CSS for nested graphs

In a diagram component, a recursive layout instance is used internally and is called automatically when needed. In IBM® ILOG® JViews Diagrammer, the handling of nested graphs is thus completely automatic.

Layout renderers perform the layout of nested graphs. The node layout renderer performs the arrangement of the nodes, and depending on the layout style, also the reshaping of the normal links (for example, in Hierarchical or Tree Layout). The link layout renderer performs the arrangement of the links, in particular of the intergraph links. The label layout renderer places all labels in the nested graph.

Note: If the nested graph has intergraph links, the link layout renderer must be enabled, otherwise the intergraph links will not be routed at all.

Same layout style everywhere

By default, layout renderers apply the same layout style to all subgraphs. The following example shows a specification in CSS that applies the Tree layout to all subgraphs.

```
SDM {
    GraphLayout: "true";
    LinkLayout: "true";
}

GraphLayout {
    graphLayout: "Tree";
}

LinkLayout {
    layoutMode: "LONG_LINKS";
    interGraphLinksMode: "true";
    combinedInterGraphLinksMode: "false";
}
```

This example specifies that the Tree layout algorithm is applied to all nodes and links. The Link layout algorithm is applied only to the intergraph links, not to the normal links. The `IlvLinkLayout` instance is in mode `LONG_LINKS`, because intergraph links are often very long.

If you change the specification in the `LinkLayout` section to:

```
interGraphLinksMode: "true";
combinedInterGraphLinksMode: "true";
```

the Link layout algorithm is applied to both normal links and intergraph links.

If you change the specification in the `LinkLayout` section to:

```
interGraphLinksMode: "false";
```

the Link layout algorithm is applied only to normal links, not to intergraph links. The intergraph links are in this case not routed at all. Of course, this is useful only if the graph does not contain any intergraph links.

Since the link layout renderer can reuse the Hierarchical layout as the link layout, the specification is slightly more comfortable, as shown in the following example:

```
SDM {
    GraphLayout: "true";
    LinkLayout: "true";
}

GraphLayout {
    graphLayout: "Hierarchical";
}

LinkLayout {
    hierarchical: "true";
    interGraphLinksMode: "true";
}
```

If the hierarchical flag of the link layout renderer is set to `true`, the normal links are routed by the Hierarchical layout. However, the intergraph links are routed by an instance of `IlvLinkLayout`, independent from the combinedInterGraphLinksMode setting.

Individual layout styles per subgraph

It is possible to specify the graph layout style for each subgraph individually. To do so, you need to create a *style rule* that selects the corresponding subgraph, like this:

```
#subgraph15 {
    GraphLayout: "@#sublayout15";
}

Subobject#sublayout15 {
    class: "ilog.views.graphlayout.tree.IlvTreeLayout";
    flowDirection: "Right";
... further parameters of Tree layout ...
}
```

In the SDM model, the subgraph with ID “`subgraph15`” is selected by the first rule. The rule specifies that an object with ID “`sublayout15`” must be created. The name `sublayout15` is arbitrary and has the only purpose of distinguishing it from all other objects. The second rule selects the object “`sublayout15`” that must be created, and specifies that this is a Tree layout. As result, a Tree layout is applied to the subgraph.

If the layout of an individual subgraph uses only default parameters, it can be specified in a shorter way, because no parameters need to be set:

```
#subgraph15 {
    GraphLayout: "Tree";
}
```

Note: Specifying `hierarchical: "true"` for the link layout works only if no individual layout styles are specified per subgraph. It works only if the entire nested graph is laid out by Hierarchical layout.

Accessing sublayouts of subgraphs

Internally, the graph layout renderer uses a different instance of `IlvGraphLayout` for each nested graph. In *Accessing graph layout instances,* we mentioned how to access the layout instance of the node layout renderer:

```
nodeLayoutRenderer.getGraphLayout();
```

This returns the layout instance of the top-level grapher.

The following example shows how to access all graph layout instances of all subgraphs.

```
IlvSDMEngine engine = diagrammer.getEngine();
Enumeration e = engine.getNodeLayoutRenderer().getLayouts(engine, true);
while (e.hasMoreElements()) {
  IlvGraphLayout layout = (IlvGraphLayout)e.nextElement();
  ... do something with this layout instance ...
}
```

You can also access the layout instance of an individual subgrapher, by using the following method of `IlvGraphLayoutRenderer`:

```
getGraphLayout(IlvSDMEngine engine, IlvGrapher grapher);
```

The mechanism for accessing the different link layout instances of a link layout renderer is exactly the same.

The API of the graph layout and link layout renderers is documented in the Java™ *API Reference Manual* of the classes `IlvGraphLayoutRenderer` and `IlvLinkLayoutRenderer`.

Layout of nested graphs in code

Describes how to perform a layout on nested graphs.

In this section

The classes that support nested graphs

The `IlvGrapher` class provided by IBM® ILOG® JViews Framework allows nested graphs to be represented. This facility is useful for applications that are not based on IBM® ILOG® JViews Diagrammer or on styling.

In an application that works directly on an instance of `IlvGrapher`, a recursive layout must be performed explicitly. Additional steps are required to perform a layout on nested graphers.

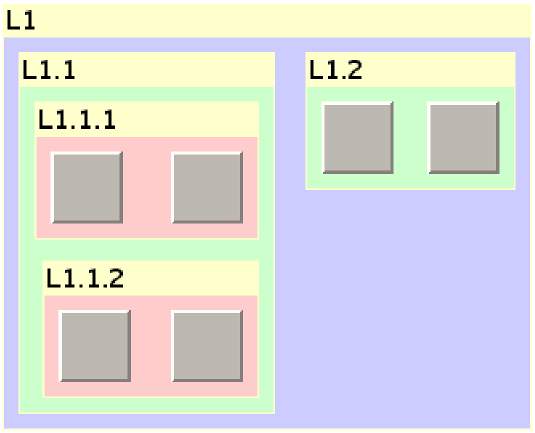
For more information, see Nested graphers.

The mechanism uses the auxiliary classes `IlvRecursiveLayout` and `IlvMultipleLayout` internally. They are explained in detail in *Recursive layout* and *Multiple layout*.

Order of layouts in recursive layouts

Assume grapher `1` contains two subgraphers L`1.1` and `L1.2`, and subgrapher `1.1` contains two subgraphers L`1.1.1` and L`1.1.2`, as shown in the following figure. The recursive layout needs to be applied in reverse order, as follows:

1. Layout on L`1.1.1`
2. Layout on `L1.1.2`
3. Layout on `L1.1`
4. Layout on `L1.2`
5. Layout on `L1`



Nested graph with recursive layouts

This means that the layout is applied to the graph once all the layouts of its subgraphs have been applied first. In our example, all layouts of subgrapher L`1.1` are finished before the layout of grapher L`1` starts. This is the correct order for a recursive layout. This order ensures that the layout of a subgraph does not invalidate the layout of its parent graphs.

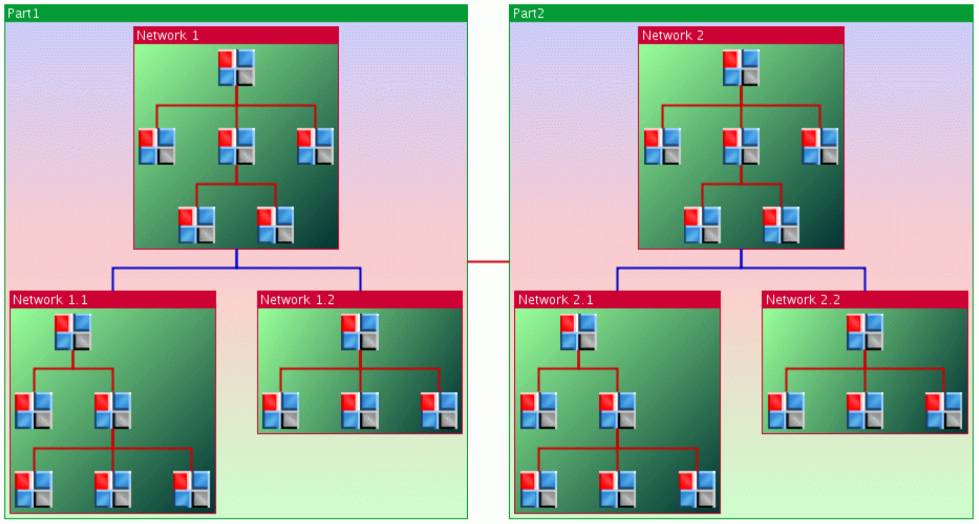
Simple recursion: applying the same layout to all subgraphers

You can apply the same layout where both the following conditions hold:

♦ The same layout algorithm needs to be applied to the topmost graph and all its subgraphs.

♦ The settings of the layout algorithm (that is, the layout parameters) need to be the same for the topmost graph as for all the subgraphs.

The following figure shows an example where a Tree Layout is applied to the topmost graph as well as to all its subgraphs. Moreover, the settings of the Tree Layout algorithm are the same for all the graphs: the application does not need, for instance, one flow direction in the topmost graph and a different one in the subgraphs.



Example of a recursive layout of a nested graph

Obtaining such recursive layouts is very easy. The class `IlvGraphLayout` provides a special version of the `performLayout` method:

```
performLayout(boolean force, boolean redraw, boolean traverse)
```

When the last `boolean` argument is set to `true`, the layout is applied not only to the graph attached to the layout instance, but also, in a recursive way, to its subgraphs.

Internal mechanism

The internal mechanism is based on the principle that a given layout instance is used for only one graph and is not reused for its subgraphs. Therefore, the Tree Layout instance is automatically “cloned” using the `copy()` method of the class `IlvGraphLayout`.

Furthermore, the graph layout is applied to a graph model, and the same principle holds for the graph models (see *Using the Graph Model*): a given graph model instance is used for only one graph and is not reused for subgraphs.

The graph models for the subgraphs are created by calls to the `getGraphModel(java.lang.Object)` method of the class `IlvGraphLayout`, which in turn creates the graph model using the method `createGraphModel(java.lang.Object)` of the class `IlvGraphModel`.

All these operations are done automatically, in a completely transparent way. All you have to do is to call the method `performLayout` with the `traverse` argument set to `true`.

If needed, you can get the layout instances applied on the subgraphs by calling the following method on `IlvGraphLayout`:

```
Enumeration getLayouts(boolean preOrder)
```

This method returns an enumeration of instances of `IlvGraphLayout`. If the `preOrder` flag is `true`, the layout of the parent graph occurs before the layout of its children in the enumeration. If the `preorder` flag is `false`, the layout of the parent graph occurs after the layout of its children. For example, in the graph of *Nesting structure in a graph*, the call `getLayouts(true)` returns the layouts for the subgraphs in this order: L1, L1.1, L1.1.1, L1.1.2, L1.2. The call `getLayouts(false)` returns the layouts for the subgraphs in this order: L1.1.1, L1.1.2, L1.1, L1.2, L1.

Java code sample

The following Java™ code sample illustrates how to apply a single layout algorithm to a nested graph:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links
...
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Attach the topmost grapher to the layout
layout.attach(grapherA);

// Perform the recursive layout
try {
        int code = layout.performLayout(true, true, true);

        System.out.println("Layout completed (code " +
          code + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// Detach the grapher when layout no more needed
layout.detach();
...
```

For experts

In this first variant, the grapher adapter (the graph model of `IlvGrapher`) is handled internally. If a grapher adapter is explicitly allocated, it must be disposed of when no longer necessary. However, all grapher adapters that are created internally are always disposed of automatically. Here is an equivalent variant that shows how to use the grapher adapter:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links
...
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Create a grapher adapter for the topmost grapher
IlvGrapherAdapter adapter = new IlvGrapherAdapter(grapherA);

// Attach the adapter to the layout
layout.attach(adapter);

// Perform the recursive layout
try {
        // perform the layout with argument traverse = true
        int code = layout.performLayout(true, true, true);

        System.out.println("Layout completed (code " +
          code + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// Detach the adapter when layout no more needed
layout.detach();
...
```

Layout parameters

Section *Internal mechanism* explains that, when applying the same layout algorithm in a recursive way, the layout instances for the subgraphs are obtained by “cloning” the layout instance attached to the topmost graph.

The layout parameters of the “clone” are the same as the parameters of the topmost layout, except for the parameters that are specific to a node or a link. Such parameters are not copied when the layouts are cloned and need to be set separately for each layout instance.

For example, if you need to declare a node `node1` contained in the subgraph `grapherB` of the topmost graph `grapherA` as fixed (see*Preserve fixed nodes*), you can use the following code:

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the layout instance
IlvTreeLayout layout = new IlvTreeLayout();

// Attach the topmost grapher to the layout
layout.attach(grapherA);

// Ask the layout algorithm to not move the nodes
// specified as fixed. This settings is automatically
// copied on the sublayouts. Do not specify this global
// settings directly on the sublayout, because it gets automatically
// the same settings as the topmost layout
layout.setPreserveFixedNodes(true);

// Search the layout instance used for grapherB
IlvGraphLayout subLayout = null;
Enumeration layouts = layout.getLayouts(true);
while (layouts.hasMoreElements()) {
       subLayout = (IlvGraphLayout)layouts.nextElement();
       if (subLayout.getGraphModel().getGrapher() == grapherB)
              break;
}

// Specify node1 (contained in grapherB) as fixed
subLayout.setFixed(node1, true);

// Now perform the recursive layout. The node node1 will be considered as fixed
// by the layout applied to grapherB
...
```

Note: You should not try to change any global settings of the layouts applied to the subgraphs (that is, settings that are not specific to a node or a link). These settings are copied anyway from the layout instance of the topmost grapher, so your changes would be erased just before the recursive layout runs.

Advanced recursion: mixing different layouts in a nested graph

The need for mixing layouts arises when at least one of the following conditions is met:

- ♦ The layout algorithm to be applied on subgraphs is not the same as the algorithm needed for the topmost graph.
- ♦ Different layouts need to be applied to different subgraphs.
- ♦ The same layout algorithm needs to be applied to different graphs but with different settings.

In these cases of *advanced recursion*, where you want to apply different layouts to different subgraphs, you need to specify which layout should be used for which subgraph. Furthermore you need to start the layouts in the correct order. This is called *recursive layout*.

The class `IlvRecursiveLayout` is a subclass of `IlvGraphLayout`, but it is not a real layout algorithm. It is rather a facility to apply other layout algorithms recursively on a nested graph.

The class `IlvRecursiveLayout` can also be used to apply the same layout to all subgraphs. In fact, when using the API explained in subsection *Simple recursion: applying the same layout to all subgraphers*, an instance of `IlvRecursiveLayout` is used internally.

The class `IlvRecursiveLayout` can furthermore be used to apply multiple layouts to the same nested graph. This is for instance necessary if for each subgraph, a node layout and a separate link layout must be applied.

Further details and code samples of the class `IlvRecursiveLayout` are explained in the following section *Recursive layout*.

To apply layout algorithms recursively:

1. Allocate and attach an instance of `IlvRecursiveLayout`. Since it is a subclass of `IlvGraphLayout`, you use the same mechanism as for all other graph layout classes:

```
IlvRecursiveLayout recLayout = new IlvRecursiveLayout();
IlvGrapher topLevelGrapher = ...
recLayout.attach(topLevelGrapher);
```

2. Specify which layout style should be used for each subgraph. You must allocate an individual instance of `IlvGraphLayout` for each subgraph.

```
recLayout.setLayout(subgraph1, new IlvTreeLayout());
recLayout.setLayout(subgraph2, new IlvBusLayout());
recLayout.setLayout(subgraph3, new IlvGridLayout());
```

3. Set the layout parameters of these individual layouts of the subgraphs as needed.
4. Apply the recursive layout to the top-level grapher. This automatically applies the sublayouts to the subgraphs as well. Since `IlvRecursiveLayout` is a subclass of `IlvGraphLayout`, you use the same method as for all other graph layout classes

```
try {
        recLayout.performLayout();
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

5. Detach the recursive layout from the top-level grapher when it is no longer needed. This automatically detaches all sublayouts from all subgraphers.

```
recLayout.detach();
```

Recursive layout

Describes the *Recursive Layout* (class `IlvRecursiveLayout` from the package `ilog.views.graphlayout.recursive`).

In this section

Overview of recursive layout

The `IlvRecursiveLayout` class is internally used by the graph layout renderer of a diagram component; in IBM® ILOG® JViews Diagrammer, the rendering mechanism is transparent so that you never need to deal with this class.

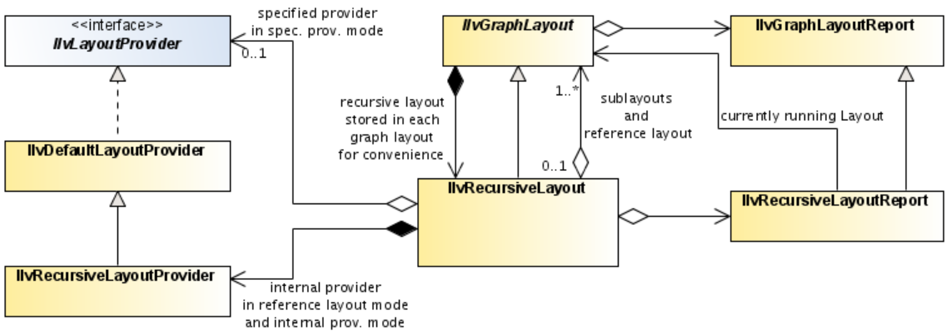
Important: The Recursive Layout can be used only in Java™ code. No CSS syntax is available for this layout.

The Recursive Layout class is not a layout algorithm but rather a facility to apply another layout algorithm recursively on a nested graph. It traverses the nesting structure starting from the graph that is attached to the Recursive Layout itself and recursively applies a layout on all subgraphs. You can tailor which sublayout must be applied to which subgraph.

There are basically two scenarios:

♦ The same layout style must be applied to all subgraphs.

♦ An individual layout style must be applied to each subgraph.



<<interface>>
IlvLayoutProvider
specified provider in spec. prov. mode
0..1
IlvDefaultLayoutProvider
IlvRecursiveLayoutProvider
internal provider in reference layout mode and internal prov. mode
IlvGraphLayout
recursive layout stored in each graph layout for convenience
1..*
sublayouts and reference layout
0..1
IlvRecursiveLayout
IlvGraphLayoutReport
currently running Layout
IlvRecursiveLayoutReport

The class IlvRecursiveLayout which manages sublayouts for nested graphs

Java code sample: same layout style everywhere

This sample assumes that you want to apply a Tree Layout to a nested graph and that each subgraph should be laid out with the same global layout parameters.

The Tree Layout algorithm handles only flat graphs, that is, if applied to an attached graph, it lays out only the nodes and links of the attached graph, but not the nodes and links of the subgraphs that are nested inside the attached graph. Hence the Tree Layout must be encapsulated into a Recursive Layout.

The Recursive Layout traverses the entire nesting hierarchy of the attached graph, while the encapsulated Tree Layout lays out each (flat) subgraph of the nesting hierarchy during the traversal.

```
...
import ilog.views.*;
import ilog.views.graphlayout;
import ilog.views.graphlayout.recursive.*;
import ilog.views.graphlayout.tree.*;

IlvRecursiveLayout layout = new IlvRecursiveLayout(IlvTreeLayout());

IlvGrapher topLevelGrapher = ...
layout.attach(topLevelGrapher);
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
...
```

```
...
import ilog.views.*;
import ilog.views.eclipse.graphlayout.GraphModel;
import ilog.views.eclipse.graphlayout.runtime.*;
import ilog.views.eclipse.graphlayout.runtime.recursive.*;
import ilog.views.eclipse.graphlayout.runtime.tree.*;
...
IlvRecursiveLayout layout = new IlvRecursiveLayout(IlvTreeLayout());

GraphModel graphModel = new GraphModel(myTopLevelGrapherEditPart);
layout.attach(graphModel);
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
```

```
layout.detach();
graphModel.dispose();
...
```

This mode of the Recursive Layout is called *reference layout mode*. In this case, a Tree Layout is performed recursively on the top-level graph and on each subgraph. All layouts are performed with the same global layout parameters.

Note: The term “global layout parameter” applies to the parameters that do not depend on a specific node or link. For example, Tree Layout has a global layout parameter set by `setGlobalLinkStyle`, as well as a layout parameter set by `setLinkStyle (link, style)` which is local to a link.

You can change the global layout parameters by accessing the reference layout of the Recursive Layout:

```
IlvTreeLayout treeLayout = (IlvTreeLayout)layout.getReferenceLayout();
treeLayout.setFlowDirection(IlvDirection.Left);
```

Technically, the reference layout instance is not applied to each subgraph because each subgraph needs an individual layout instance. The reference layout instance is only applied to the top-level graph. Furthermore, a clone of the reference instance is created for each subgraph. This clone remains attached to the subgraph as long as the Recursive Layout is attached to the top-level graph. Before layout is performed, the global layout parameters are copied from the reference layout instance to each cloned layout instance.

Sometimes, you want to specify local layout parameters for individual nodes and links. In this case, you need to access the cloned layout instance that is attached to the subgraph that owns the node or link. For instance, to the link style of an individual link, use:

```
IlvTreeLayout treeLayout =
(IlvTreeLayout)layout.getLayout(link.getGraphicBag());
treeLayout.setLinkStyle(link, IlvTreeLayout.ORTHOGONAL_STYLE);
```

You cannot use the reference layout mode in the following cases:

♦ The layout algorithm to be applied on subgraphs is not the same as the algorithm needed for the topmost graph (the reference layout).

♦ The same layout algorithm, but using different global parameter settings, needs to be applied on different subgraphs.

In these cases, you can use one of the other modes.

Java code sample: mixing different layout styles

The following example shows the second scenario: Each subgraph should be laid out by a different layout style or with individual global layout parameters. In this case, you use the *internal provider mode* of the Recursive Layout.

We assume that you have a graph with three subgraphs. The top-level graph and the first subgraph should be processed with Tree Layout, the second subgraph with Bus Layout, and the third subgraph with Grid Layout. You have to specify which layout should be used for which subgraph, and then you can perform the layout.

```
...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.recursive.*;
import ilog.views.graphlayout.tree.*;
import ilog.views.graphlayout.bus.*;
import ilog.views.graphlayout.grid.*;

IlvRecursiveLayout layout = new IlvRecursiveLayout();
IlvGrapher topLevelGrapher = ...
layout.attach(topLevelGrapher);

// specify the layout of the top level graph
layout.setLayout(null, new IlvTreeLayout());
// specify the layout of subgraphs
layout.setLayout(subgraph1, new IlvTreeLayout());
layout.setLayout(subgraph2, new IlvBusLayout());
layout.setLayout(subgraph3, new IlvGridLayout());

// perform layout
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
...
```

In this scenario, there is no reference layout. All layout parameters of different subgraphs are independent. You need to specify new, independent layout instances for each subgraph; otherwise no layout will be performed for the corresponding subgraph. The layout instances are attached to the subgraph as long as the Recursive Layout is attached to the top-level graph. You can specify in this example different global layout parameters for the Tree Layout of the top-level graph and the Tree Layout of `subgraph1`. You access the layout instance of each individual subgraph to change global layout parameters for this subgraph as well as parameters of nodes and links of the subgraph. For instance if `node1` belongs to `subgraph1` and `node2` belongs to s`ubgraph2`, you can set individual global and local layout parameters in this way:

```
// access the layout of the top level graph
IlvTreeLayout treeLayout1 = (IlvTreeLayout)layout.getLayout(null);
treeLayout1.setFlowDirection(IlvDirection.Bottom);
// access the layouts of the subgraphs
IlvTreeLayout treeLayout2 = (IlvTreeLayout)layout.getLayout(subgraph1);
treeLayout2.setFlowDirection(IlvDirection.Left);
treeLayout2.setAlignment(node1, IlvTreeLayout.TIP_OVER);
IlvBusLayout busLayout = (IlvBusLayout)layout.getLayout(subgraph2);
busLayout.setOrdering(IlvBusLayout.ORDER_BY_HEIGHT);
busLayout.setBus(node2);
IlvGridLayout gridLayout = (IlvGridLayout)layout.getLayout(subgraph3);
gridLayout.setLayoutMode(IlvGridLayout.TILE_TO_COLUMNS);
```

Java code sample: using a specified layout provider

The IBM® ILOG® JViews Diagrammer Graph Layout library provides a flexible mechanism for the choice of the layout instance to be applied to each subgraph in a nested graph: the layout provider. In the previous example, a layout provider was used internally. For simplicity, the details of the mechanism are hidden, and you select the choice of layout by using the method `setLayout` on the Recursive Layout instance. Therefore, this layout mode is called *internal provider mode*.

However, you can also design your own layout provider and use it inside the Recursive Layout. This is the *specified provider mode* of the Recursive Layout.

A layout provider is a class that implements the interface `IlvLayoutProvider`. The interface has a unique method:

```
getGraphLayout(IlvGraphModel graphModel)
```

This method must return the layout instance to be used for the graph model passed as the argument, or `null` if no layout is required for this graph. When performing the Recursive Layout, these methods get the layout instance to be used for each graph from the specified layout provider.

To implement the interface `IlvLayoutProvider`, you must decide how the choice of the layout instance is done. This can be based on some criteria such as the type of graph (eventually known in advance), or a choice already made by the end user and recorded, for example, in a property of the graph. A possible implementation of the `getGraphLayout` method is the following:

```
public IlvGraphLayout getGraphLayout(IlvGraphModel graphModel)
{
  Object prop = graphModel.getProperty("layout type");
  // if none, return null (no layout needed for this graph)
  if (!(prop instanceof String))
    return null;

  IlvGraphLayout layout = null;
  String name = (String)prop;
  if (name.equals("tree"))
    layout = new IlvTreeLayout();
  else if (name.equals("flow"))
    layout = new IlvHierarchicalLayout();
  else
    throw new RuntimeException("unsupported layout choice: " + name);

  layout.attach(graphModel);

  return layout;
}
```

Of course, this is only an example among many possible implementations. The implementation may decide to store the newly allocated layout instance to avoid allocating a new one when the method is again called for the same graph.

If you have implemented a layout provider, you can use it in the Recursive Layout in the following way:

```

..
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.recursive.*;

IlvLayoutProvider layoutProvider = ...
IlvRecursiveLayout layout = new IlvRecursiveLayout(layoutProvider);
IlvGrapher topLevelGrapher = ...
layout.attach(topLevelGrapher);

// Perform the layout
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the Recursive Layout when it is no longer needed
layout.detach();
...
```

Features

- ♦ This subclass of `IlvGraphLayout` is not a usual layout algorithm but rather a facility to manage the layout of a nested grapher.
- ♦ Three layout modes: reference layout mode, internal provider mode, and specified provider mode.
- ♦ Allows you to perform a layout algorithm recursively in a nested grapher.
- ♦ Allows you to perform a recursive layout on a nested grapher while each subgrapher uses an individual layout style.
- ♦ Layout features, speed, and quality depend on the features, speed, and quality of the sublayouts.

Generic features and parameters

Depending on the support of its sublayouts, Recursive Layout may support the following generic parameters defined in the `IlvGraphLayout` class (see *Generic parameters and features*):

♦ *Allowed time*

♦ *Percentage completion calculation*

♦ *Save parameters to named properties*

♦ *Stop immediately*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed time

The Recursive Layout can stop the entire layout of a nested graph after a certain amount of time. If the allowed time setting has elapsed, the Recursive Layout stops; that means it stops the currently running layout of a subgraph and skips the subsequent layouts of subgraphs that have not yet been started. If at the stop time point a sublayout is running on a subgraph that does not support the “allowed time” feature, then this sublayout first runs to completion before the Recursive Layout is stopped. If the Recursive Layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Percentage completion calculation

The Recursive Layout calculates the percentage of completion. This value can be obtained from the layout report during the run of the layout. The value is, however, a very rough estimation. If the layouts on the subgraphs do not support the calculation of the percentage completion, the Recursive Layout can report the percentage based only on the information how many layouts of subgraphs are already finished. For instance, if the entire nesting structure contains five nested graphs, the mechanism reports 20% after the layout of the first subgraph has finished, 40% after the layout of the second subgraph has finished, and so on. If the layouts of the subgraphs support the calculation of the percentage completion, the Recursive Layout calculates a more detailed percentage. In most cases, the calculated percentage is only a very rough estimation that does not always grow linearly over time. (For a detailed description of this feature, see *Percentage of completion calculation* and *Listener layout*)

Save parameters to named properties

The Recursive Layout instance can save its layout parameters into named properties if all its sublayouts support this feature. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Percentage of completion calculation*and *Saving layout parameters and preferred layouts*.)

Stop immediately

The Recursive Layout can be stopped at any time. It stops the currently running layout of a subgraph after cleanup if the method `stopImmediately()` is called and skips the subsequent layouts of subgraphs that have not yet been started. If at the stop time point a sublayout is running on a subgraph that does not support the "stop immediately" feature, then this sublayout first runs to completion before the Recursive Layout is stopped. For a description of this method in the IlvGraphLayout class, see *Stop immediately*. If the layout stops before completion, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Recursive layout modes

Describes the modes available in this layout.

In this section

Overview of recursive layout modes

The Recursive Layout has three different layout modes:

- *Reference layout mode*
- *Internal provider mode*
- *Specified provider mode*

The layout mode is determined by the constructor that you use. The way how to set global layout parameters of the sublayouts that are applied to the subgraphs is slightly different for each layout mode. You can query the current layout mode by using

```
int getLayoutMode()
```

The possible return values are:

- `IlvRecursiveLayout.REFERENCE_LAYOUT_MODE`: The same layout style with the same global layout parameters is applied to all subgraphs of the nested graph.
- `IlvRecursiveLayout.INTERNAL_PROVIDER_MODE`: The layout is applied using an internal recursive layout provider. The layout styles of individual subgraphs can be specified by using the method `setLayout`.
- `IlvRecursiveLayout.SPECIFIED_PROVIDER_MODE`: The layout is applied using an explicitely specified layout provider.

This section is divided as follows:

- *Accessing all sublayouts*
- *Convenience method for setting reference layout mode*

Reference layout mode

Use this mode if you want to apply the same layout style with the same global layout parameters to the entire nested graph. You first need to allocate the reference layout, that is a new instance of any graph layout algorithm (except `IlvRecursiveLayout`) that should be applied to all subgraphs of the nested graph. Then you allocate the Recursive Layout using the constructor with the reference layout as argument

```
IlvRecursiveLayout(IlvGraphLayout referenceLayout)
```

The reference layout is internally only used for the top-level graph of the nested graph. Clones of the reference layout are used for the subgraphs. Hence, all subgraphs are laid out with the same global layout parameters. To change the global layout parameters, you can access the reference layout by

```
IlvGraphLayout getReferenceLayout()
```

Global layout parameters are those parameters that are independent from specific nodes or links. Other layout parameters are local to specific nodes or links. For instance, in `IlvHierarchicalLayout`, the method `setGlobalLinkStyle(style)` is a global layout parameter, while the method `setLinkStyle(link,style)` is a local layout parameter.

If you need to set layout parameters that are local to an individual node or link, you need to access the particular clone of the reference layout that is responsible for the subgraph that owns the node or link. After attaching the Recursive Layout to the top-level grapher or graph model, you can retrieve the layout instance of a specific subgraph by

```
IlvGraphLayout getLayout(Object subgraph)
```

However, in reference layout mode, it makes no sense to modify any global layout parameter on the returned instance. The global layout parameters are always taken from the reference layout only. If you pass `null` as subgraph, you get the layout instance of the top-level graph. This is actually the same layout instance as the reference layout.

The reference layout and its clones used during recursive layout remain attached to the subgraphs (or the graph models of the subgraphs, respectively) as long as the Reference Layout itself is attached. When detaching the Reference Layout, all layouts of subgraphs are automatically detached as well.

Internal provider mode

Use this mode if you want to perform graph layout on a nested graph, but either you need individual global layout parameters for specific subgraphs, or you want to lay out different subgraphs with different styles. In this case, there is no reference layout. You allocate the Recursive Layout using the constructor with no arguments

```
IlvRecursiveLayout()
```

Before you can perform a layout, you need to specify which layout is used for which subgraph. First, you should attach the Recursive Layout to a graph. Then, to specify the layout of the top-level graph, call:

```
recursiveLayout.setLayout(null, sublayout);
```

To specify the layout of a specific subgraph, call

```
recursiveLayout.setLayout(subgraph, sublayout);
```

It is important that you assign a different layout instance for each subgraph. You cannot share the same layout instance among different subgraphs. We recommend, that you allocate a new, fresh layout instance for each subgraph. If you pass `null` as sublayout, then no layout is performed for this particular subgraph.

To set the layout for a subgraph and recursively for all its subgraphs, you can use

```
setLayout(Object subgraph, IlvGraphLayout layout, boolean traverse)
```

and pass the `true` argument for the `traverse` flag. This sets the layouts to a clone of the input layout for each subgraph starting at the input subgraph.

Internally, the Recursive Layout uses a layout provider of type `IlvRecursiveLayoutProvider`. You can access the current layout provider by

```
IlvLayoutProvider getLayoutProvider()
```

However, in internal provider mode, it is mostly not necessary to manipulate the layout provider directly.

Since there is no reference layout, global layout parameters are independent for each subgraph. Global and local layout parameters can be set by accessing the particular layout instance that is assigned to a specific subgraph. After attaching the Recursive Layout to the top-level grapher or graph model, you can retrieve the layout instance of a specific subgraph by

```
IlvGraphLayout getLayout(Object subgraph)
```

If you pass `null` as subgraph, you get the layout instance of the top-level graph.

The layout instances of the subgraphs used during recursive layout remain attached to the subgraphs (or the graph models of the subgraphs, respectively) as long as the Reference Layout itself is attached. When you detach the Reference Layout, all layouts of subgraphs are automatically detached as well.

Specified provider mode

The specified provider mode can be used if you want to perform graph layout on a nested graph, but either you need individual global layout parameters for specific subgraphs, or you want to lay out different subgraphs with different styles. It is your own responsibility to manage the specified layout provider (unlike the case with the internal provider mode), but this is probably only necessary in very advanced applications.

In specified provider mode, there is no reference layout. You allocate the Recursive Layout using the constructor with your layout provider as argument

```
IlvRecursiveLayout(IlvLayoutProvider specifiedProvider)
```

You can access the current layout provider by

```
IlvLayoutProvider getLayoutProvider()
```

You should implement your layout provider in a way so that it delivers a different layout instance for each subgraph. The delivered layout instance must be attached to the graph model of the corresponding subgraph.

Since there is no reference layout, global layout parameters are independent for each subgraph. It is recommended that the implementation of the layout provider takes care of the setting of global and local layout parameters. Theoretically, you can use the method

```
IlvGraphLayout getLayout(Object subgraph)
```

which will return the layout instance that the specified layout provider delivers for the graph model of the input subgraph. If you pass `null` as subgraph, you get the layout instance of the top-level graph. However, the effect of this method depends on the implementation of the layout provider that was passed to the constructor of Recursive Layout.

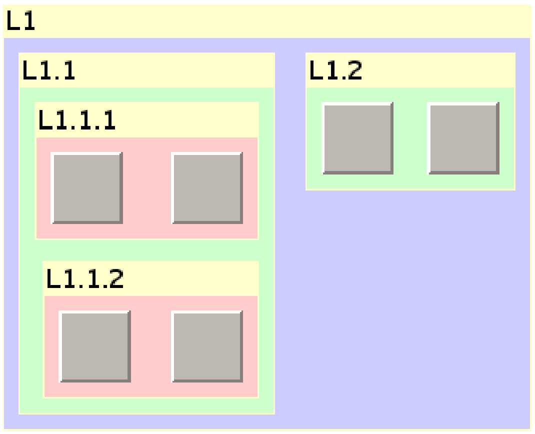
The layout instances of the subgraphs used during recursive layout should be attached by the layout provider. They are usually not automatically detached when the Recursive Layout is detached. Unless you use one of the predefined providers of class `IlvDefaultLayoutProvider` or `IlvRecursiveLayoutProvider`, you should traverse all layouts and detach them explicitly.

Accessing all sublayouts

When the Recursive Layout is attached, you can conveniently access all layouts that will be used during layout. This works in all layout modes:

```
Enumeration getLayouts(boolean preOrder)
```

As explained in *Internal mechanism*, the `getLayouts` method returns an enumeration of instances of `IlvGraphLayout`. If the `preOrder` flag is `true`, the layout of the parent graph occurs before the layout of its children in the enumeration. If the `preorder` flag is `false`, the layout of the parent graph occurs after the layout of its children. For example, in the graph in the following figure, the call `getLayouts(true)` returns the layouts for the subgraphs in this order: L1, L1.1, L1.1.1, L1.1.2, L1.2; the call `getLayouts(false)` returns the layouts for the subgraphs in this order: L1.1.1, L1.1.2, L1.1, L1.2, L1.



Nesting structure in a graph

Note: In specified provider mode, the enumeration returned by `getLayouts` contains the instances that are delivered by the specified provider. If the specified provider returns a different instance in each call of `getGraphLayout(IlvGraphModel)`, then the enumeration does not contain the instances that are later used during layout. Hence it is recommended to use layout providers that store the layout instances internally and return the same instance for the same graph model. The predefines `IlvDefaultLayoutProvider` and `IlvRecursiveLayoutProvider` store the layout instances internally.

Convenience method for setting reference layout mode

The class IlvGraphLayout contains a convenience method. To perform a recursive layout recursively, you can use:

```
int performLayout(boolean force, boolean redraw, boolean traverse)
```

If the traverse flag is true, it traverses the nested graph and performs the layout on each subgraph. In fact, this is just a shortcut for the reference mode of Recursive Layout. The statement

```
flatLayout.performLayout(force, redraw, true);
```

is equivalent to creating a Recursive Layout in reference mode that uses the flatLayout as reference layout:

```
IlvRecursiveLayout recursiveLayout = new IlvRecursiveLayout(flatLayout);
recursiveLayout.performLayout(force, redraw);
```

Specific parameters

Besides some expert parameters, Recursive Layout does not provide any specific layout parameters. You can set specific layout parameters of the sublayouts individually by accessing them via `getLayout(Object)`:

```
IlvGraphLayout sublayout = recursiveLayout.getLayout(subgraph);
sublayout.setParameter(parameter);
```

However, Recursive Layout has some convenient methods that automatically traverse the nested graph recursively and set the corresponding parameter at each sublayout of a subgraph that supports this parameter. This works well particularly in reference layout mode. In internal or specified provider mode, it takes only the current nesting structure into account. If you change the specific layout of a subgraph or the nesting structure (for example, by adding a new subgraph) after using such a convenience method, the new layout of the new subgraph usually has a different setting, so you may need to apply the convenience method again.

The following methods traverse the nested graph recursively and set the corresponding parameter on all sublayouts (see *Generic parameters and features* and *Using advanced features* for details):

- `setCoordinatesMode(int mode)`
- `void setUseDefaultParameters(boolean option)`
- `void setMinBusyTime(long time)`
- `void setInputCheckEnabled(boolean enable)`
- `void propagateLayoutOfConnectedComponentsEnabled(boolean enable)`
- `void propagateLayoutOfConnectedComponents(IlvGraphLayout layout)`
- `void propagateLinkConnectionBoxInterface(IlvLinkConnectionBoxInterface linkConnectionBoxInterface)`
- `propagateLinkClipInterface(IlvLinkClipInterface linkClipInterface)`
- `void checkAppropriateLinks()`
- `void setLinkCheckEnabled(boolean enable)`
- `void setConnectionPointCheckEnabled(boolean enable)`

There is a generic propagation mechanism for setting any parameter which is implemented by reflection. For example, the following call traverses the nested graph recursively, checks for each sublayout using introspection whether a method called `setFlowDirection` exists, and passes the input value `direction` to this method. As a result, all sublayouts that have a flow direction parameter will use the same flow direction, while the layout parameters of those layouts that do not have a flow direction remain unchanged:

```
int code = recursiveLayout.propagateLayoutParameter("flowDirection",
null, direction);
```

The second argument of `propagateLayoutParameter` can be used to select only specific layout classes. The call

```
int code = recursiveLayout.propagateLayoutParameter("flowDirection",
                             IlvHierarchicalLayout.class, direction);
```

propagates the flow direction only to all those sublayouts that are instances of `IlvHierarchicalLayout`. For example if a subgraph uses a Tree Layout, its flow direction remains unchanged in this case, even though `IlvTreeLayout` has a method `setFlowDirection`.

The return code of the propagation method indicates whether setting the parameter has been successful. It is a bitwise-Or combination of the following bit masks:

- `IlvRecursiveLayout.PROPAGATE_PARAMETER_SET` - the parameter was successfully applied at some layout instance of a subgraph.
- `IlvRecursiveLayout.PROPAGATE_PARAMETER_AMBIGUOUS` - the method to set the parameter could not uniquely be determined at some layout instance, because there were many methods with the same name, which creates an unresolvable ambiguity. In this case, an arbitrary method is choosen among the ambiguous methods.
- `IlvRecursiveLayout.PROPAGATE_CLASS_MISMATCH` - the parameter was not applied at some layout instance of a subgraph because the layout instance did not match the specified layout class. This can happen only when a non-null layout class is specified as the second parameter of the method `propagateLayoutParameter`.
- `IlvRecursiveLayout.PROPAGATE_PARAMETER_MISMATCH` - the parameter was not applied at some layout instance of a subgraph because no matching method with appropriate argument types was found via reflection, or because the security manager of the Java™ Virtual Machine did not allow reflection. In Java applets, reflection is often not permitted.
- `IlvRecursiveLayout.PROPAGATE_EXCEPTION` - the method to set the parameter was applied but threw an exception at some layout instance of a subgraph.

For further details about the propagation mechanism, see the class `IlvRecursiveLayoutIlvRecursiveLayout` in the *Java API Reference Manual*.

For experts: mechanisms for advanced users

The mechanisms available for advanced users are:

♦ *Subgraph correction*, to correct the subgraphs during layout

♦ *Listener layout*, to install layout event listeners efficiently

Subgraph correction

In IBM® ILOG® JViews, the position of a subgrapher (instance of `IlvGrapher`) is always calculated from the positions of the contents of the subgrapher. The position of the subgrapher is simply the position of the bounding box around its contents. This mechanism has side effects when performing layout: a subgraph will never appear fixed if its contents is laid out, because when a layout is applied to the contents of the subgraph, the bounding box, hence the position, of the subgraph changes. Depending on the applications, this may be an unwanted effect.

Normally, after the subgraph is laid out, its parent graph gets laid out, which eventually moves the entire subgraph to its final position. Therefore, in most cases, the unwanted position of the subgraph is only temporary and you can ignore the entire problem. However, in a few situations, you need to be aware of the effect, namely:

♦ if the subgraph is specified as fixed (for instance, by a call to the method `setFixed`) and hence should not move;

♦ if the parent graph is never laid out;

♦ if the parent graph is laid out in a layout style with incremental mode on, which analyzes the positions of the nodes (for instance in Tree Layout and Hierarchical Layout).

In all these situations, it is important that the subgraph remains at the old position even after its contents has been laid out. The implementation of `IlvGrapher` does not behave like this.

The solution is simply to move the subgraph back to its old position immediately after the subgraph has been laid out, just before the layout of its parent graph is started. The Recursive Layout allows you to install a subgraph correction interface that contains a `correct` method which is called exactly at this point. You install a subgraph correction interface in the following way:

```
layout.setSubgraphCorrectionInterface(
new IlvSubgraphCorrectionBarycenterFixed());
```

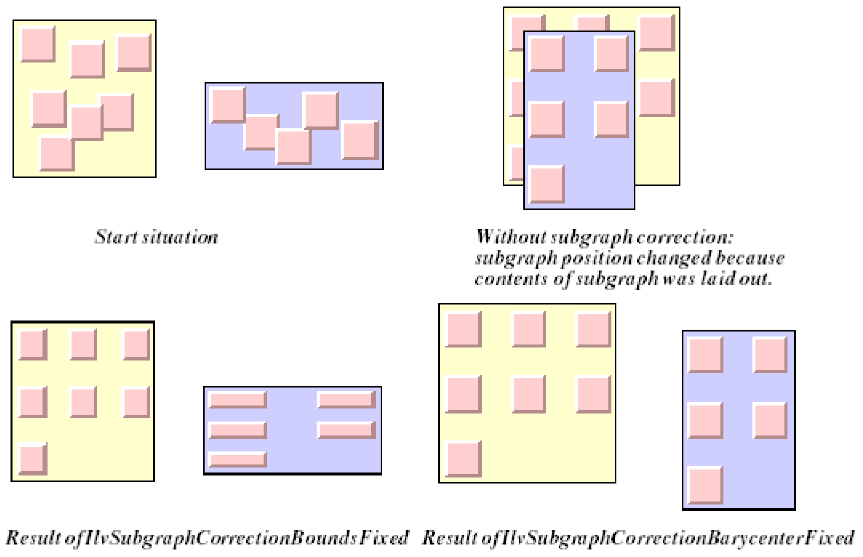
Effect of Subgraph Correction illustrates the effect of subgraph correction.

Two default implementations of `IlvSubgraphCorrectionInterface` are available:

♦ `IlvSubgraphCorrectionBarycenterFixed` corrects the subgraph so that after its contents has been laid out, the center of the subgraph remains the same. However, the size of the subgraph bounding box may change due to the contents layout.

- `IlvSubgraphCorrectionBoundsFixed` corrects the subgraph so that after its contents has been laid out, the bounding box of the subgraph remains the same. However, to achieve this, the zoom level of the subgraph is changed.

These implementations of the subgraph correction interface do not correct the top-level graph, but only the nested subgraphs The instances can be shared between different instances of `IlvRecursiveLayout`.


Start situation
Without subgraph correction: subgraph position changed because contents of subgraph was laid out.
Result of IlvSubgraphCorrectionBoundsFixed
Result of IlvSubgraphCorrectionBarycenterFixed

Effect of Subgraph Correction

Listener layout

Event listener layout is an advanced feature documented in *Using event listeners*. You need to understand that general description and the concept of layout listeners before you read this section. This section describes some specific details of the Recursive Layout related to layout listeners.

The application can listen for layout events sent by the Recursive Layout or by each sublayout individually. For example, a progress bar that displays the progress of the entire nested layout should listen for the layout events fired by the Recursive Layout itself, while an application that wants to detect when a specific sublayout of a subgraph is started or stopped should listen for the layout events sent by that particular sublayout.

To install a layout event listener at the Recursive Layout, call usually:

```
recursiveLayout.addGraphLayoutEventListener(listener);
```

To install a layout listener that receives the layout events of all sublayouts of the Recursive Layout, you can call:

```
recursiveLayout.addSubLayoutEventListener(listener);
```

Note that in this case, the listener is installed at the Recursive Layout instance (not at the sublayout instances) but receives the events from the sublayouts (not from the Recursive Layout). An internal mechanism makes sure that the events are forwarded to the listener.

Alternatively, you could traverse the nesting structure and install the same listener at all subgraph layouts. However, this would have two disadvantages: it requires more memory and you need to reinstall or update the listener whenever you change the layout of a subgraph or the nesting structure by adding or removing subgraphs. When you use `addSubLayoutEventListener`, updating the listener is not necessary in this case.

For experts: more on layout providers

For information on use the Recursive Layout with a specified layout provider, see *Specified provider mode*.

The library provides a default implementation of the interface `IlvLayoutProvider`, named `IlvDefaultLayoutProvider`. In many cases, it is simpler either to use this class as is, or to subclass it, rather than directly implementing the interface.

The class `IlvDefaultLayoutProvider` allows you to set the layout instance to be used for each graph (called the preferred layout) with the method:

```
setPreferredLayout(IlvGraphModel graphModel, IlvGraphLayout layout, boolean
detachPrevious)
```

The layout instance specified as the preferred layout is stored in a property of the graph. The current preferred layout is returned by the method:

```
getPreferredLayout(IlvGraphModel graphModel)
```

The method returns `null` if no layout has been specified for this graph.

When the method `getGraphLayout` is called on the default provider, the previously specified preferred layout is returned, if any. Otherwise, a new layout instance is allocated by a call to the method

```
createGraphLayout(IlvGraphModel graphModel)
```

This newly created layout is recorded as the preferred layout of this graph, which is attached to the layout instance.

When a preferred layout has been specified for a given graph, the default implementation of the method `createGraphLayout` copies the layout instance that is the preferred layout of the nearest parent graph. Therefore, if a preferred layout `L` is specified for a graph `G` and no preferred layout is set for its subgraphs, then the graph `G` and all its subgraphs are laid out using the same layout algorithm `L` (copies of it are used for the subgraphs).

Note: You must call the method `detachLayouts` when you no longer need the layout provider instance; otherwise, the garbage collector may fail to remove some objects.

The settings of the preferred layout made using the class `IlvDefaultLayoutProvider` can be saved in `.ivl` files. For details, see *Saving layout parameters and preferred layouts*.

Java Code Sample:

The following Java™ code sample illustrates the use of the class `IlvDefaultLayoutProvider`.

```
...
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// Fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create a grapher adapter for the topmost graph
IlvGrapherAdapter adapterA = new IlvGrapherAdapter(grapherA);

// Get a grapher adapter for the subgraph
IlvGraphModel adapterB = adapterA.getGraphModel(grapherB);

// Create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// Specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(adapterA, new IlvTreeLayout());
provider.setPreferredLayout(adapterB, new IlvGridLayout());

// Create a recursive layout in specified provider mode
IlvRecursiveLayout layout = new IlvRecursiveLayout(provider);

// Perform the layout
try {
        IlvRecursiveLayoutReport layoutReport =
              (IlvRecursiveLayoutReport)layout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
...
// detach the layouts when the provider is no longer needed
provider.detachLayouts(adapterA, true);
// dispose the topmost adapter when no longer needed
adapterA.dispose();
...
```

Multiple layout

Describes the *Multiple Layout* class (class `IlvMultipleLayout` from the package `ilog.views.graphlayout.multiple`).

In this section

General information

What is multiple layout?

The Multiple Layout class is useful mainly when applying layout in Java™ code. The class is also internally used by the graph layout renderer of a diagram component, but in the diagram component, the rendering mechanism is transparent so that you seldom need to deal with the class `IlvMultipleLayout`.

Important: The Multiple Layout can be used only in Java code. No CSS syntax is available for this layout.

The Multiple Layout class is not a layout algorithm but rather a facility to compose multiple layout algorithms and treat them as one algorithm object. This is necessary in particular when dealing with the recursive layout of nested submanagers (see *Recursive layout* and *Layout of nested graphs in code*) because performing the layouts recursively one after the other has a different effect than combining the layouts into one algorithm object and applying this object all at once. Multiple Layout should also be used to combine a normal layout with a Link Layout that routes intergraph links. This is illustrated in the following sample.

Java code sample

You can, for instance, combine a Tree Layout, a Link Layout, and an Annealing Label Layout into one object of type `IlvGraphLayout` in the following way:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.*;
import ilog.views.graphlayout.multiple.*;
import ilog.views.graphlayout.tree.*;
import ilog.views.graphlayout.link.*;
import ilog.views.graphlayout.labellayout.annealing.*;

IlvTreeLayout treeLayout = new IlvTreeLayout();
IlvLinkLayout linkLayout = new IlvLinkLayout();
IlvAnnealingLabelLayout labelLayout = new IlvAnnealingLabelLayout();
IlvMultipleLayout multipleLayout =
        new IlvMultipleLayout(treeLayout, linkLayout, labelLayout);

IlvGrapher grapher = ...
layout.attach(grapher);

... /* Fill in code to set the layout parameters of treeLayout,
     * linkLayout and labelLayout.
     */
```

```
linkLayout.setInterGraphLinksMode(true);
...
```

By constructing a Multiple Layout instance in this way, the Tree Layout, Link Layout, and Label Layout become *sublayouts* of the Multiple Layout instance. Attaching the Multiple Layout will automatically attach its sublayouts.

The Multiple Layout has two slots for graph layouts and one slot for the label layout. Not all slots need to be used. You can pass `null` as the sublayout for unused slots. If you need more slots, you can compose a Multiple Layout that contains another Multiple Layout as a sublayout.

To perform the composed layout you use one of the following:

♦ *Simple layout*

♦ *Recursive layout*

Simple layout

You can perform the composed layout on a flat grapher (one which contains no submanagers) in the following way:

```
try {
        IlvMultipleLayoutReport layoutReport =
              (IlvMultipleLayoutReport)multipleLayout.performLayout();

        int code = layoutReport.getCode();

        System.out.println("Layout completed (" +
          layoutReport.codeToString(code) + ")");
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

The statement with `multipleLayout.performLayout()` in this case has the same effect as the sequence of the three statements:

```
treeLayout.performLayout();
linkLayout.performLayout();
labelLayout.performLayout();
```

Recursive layout

If you perform the Multiple Layout on a grapher that contains submanagers, there is a difference in the order of the layout (see *Order of layouts in recursive layouts*. You apply a recursive layout on the grapher and its submanagers in the following way:

```
IlvRecursiveLayout recursiveLayout = new IlvRecursiveLayout(multipleLayout);
try {
        IlvGraphLayoutReport layoutReport = recursiveLayout.performLayout();
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

or alternatively, in the following way (both ways are equivalent):

```
try {
        int layoutCode =
                (IlvMultipleLayoutReport)multipleLayout.performLayout(
                                                        true, true, true);
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Assume the attached grapher `A` contains a subgrapher `B`. The combined Multiple Layout applies its sublayouts in reverse order, as follows:

1. Tree Layout on `B`
2. Link Layout on `B`
3. Label Layout on `B`
4. Tree Layout on `A`
5. Link Layout on `A`
6. Label Layout on `A`

This means that all layouts of subgrapher `B` have finished before the layout of grapher `A` starts. This is the correct order for a recursive layout.

If you do not combine the three component layouts into a Multiple Layout, you can only apply them sequentially:

```
treeLayout.performLayout(true, true, true);
linkLayout.performLayout(true, true, true);
labelLayout.performLayout(true, true, true);
```

The effect of these three statements is slightly different than the effect of the Multiple Layout. The layouts are now applied in the following order:

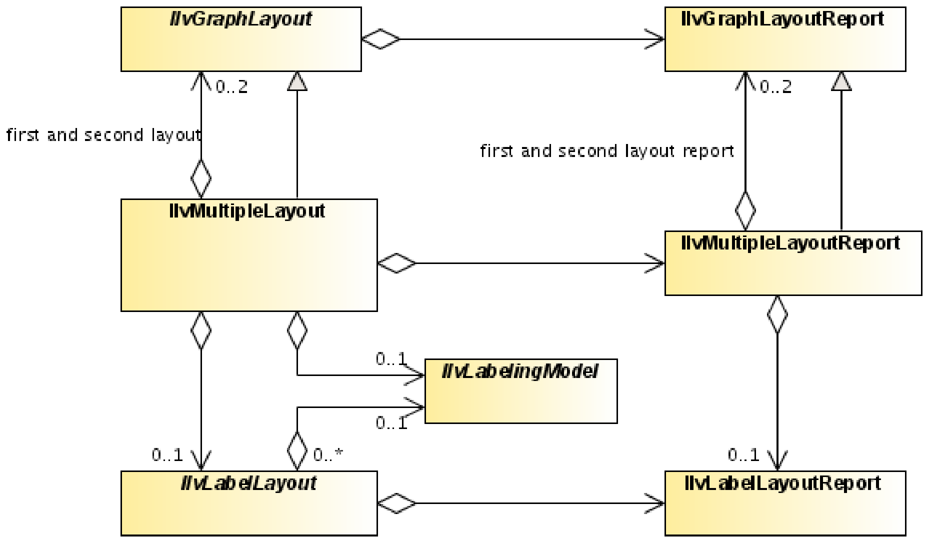
1. Tree Layout on `B`
2. Tree Layout on `A`
3. Link Layout on `B`
4. Link Layout on `A`
5. Label Layout on `B`

6. Label Layout on `A`

This order is not usually suitable for the layout of nested graphers because the Tree Layout of grapher `A` is started too early. The Label Layout on grapher `B` in Step 5 may change the position of grapher `B` within grapher `A`, invalidating the result of the Tree Layout in Step 2. Hence, it is recommended that you combine multiple layout algorithms into one Multiple Layout object and apply this object as a whole to a nested grapher.

Features

- ♦ Allows the composing of two graph layout algorithms and one label layout algorithm into one layout object.
- ♦ Should be used to achieve the correct layout order when dealing with nested graphers.
- ♦ Layout features, speed, and quality depend on the features, speed, and quality of the sublayouts.



The class IlvMultipleLayout can contain two sublayouts and one label layout

Generic features and parameters

Depending on the support of its sublayouts, Multiple Layout may support the following generic parameters and features defined in the `IlvGraphLayout` class (see *Generic parameters and features*):

- ♦ *Allowed time*
- ♦ *Layout of connected components*
- ♦ *Percentage completion calculation*
- ♦ *Stop immediately*

Extra feature for JViews Diagrammer:

- ♦ *Save parameters to named properties*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed time

A Multiple Layout instance supports this feature if all of its sublayouts support the feature. If the allowed time setting has elapsed, the Multiple Layout stops; that means it stops the currently running sublayout and skips the subsequent sublayouts that have not yet been started. If the layout stops early because the allowed time has elapsed, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Layout of connected components

The Multiple Layout instance can use the generic mechanism to lay out connected components if the sublayouts of type `IlvGraphLayout` support this feature. The sublayout of type `IlvLabelLayout` does not need special handling of connected components. For more information about this mechanism, see *Layout of connected components*.

Percentage completion calculation

The Multiple Layout calculates the percentage of completion. This value can be obtained from the layout report during the run of the layout. The value is, however, a very rough estimation. If the sublayouts do not support the calculation of the percentage completion, the Multiple Layout can report the percentage based only on the information that the sublayout has already finished. For instance, if there are three sublayouts, the mechanism reports 33% after the first sublayout has finished, 66%after the second sublayout has finished, and 100% after all three sublayouts have finished. If the sublayouts support the calculation of the percentage completion, the Multiple Layout calculates a more detailed percentage. For a detailed description of this feature, see *Percentage of completion calculation* and *Graph layout event listeners*.

Save parameters to named properties

The Multiple Layout instance can save its layout parameters into named properties if all its sublayouts support this feature. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Save parameters to named properties*and *Saving layout parameters and preferred layouts*.)

Stop immediately

The Multiple Layout instance supports this feature if all its sublayouts support this feature. It stops the currently running sublayout after cleanup if the method `stopImmediately()` is called and skips the subsequent sublayouts that have not yet been started. For a description of this method in the IlvGraphLayout class, see *Stop immediately*. If the layout stops before completion, the result code in the layout report is `IlvGraphLayoutReport.STOPPED_AND_INVALID`.

Specific parameters

Multiple Layout does not provide any specific layout parameters. However, you can set the generic and specific layout parameters of the sublayouts individually. For instance, you can construct a Multiple Layout instance from two graph layouts. Even though the Multiple Layout does not support setting fixed nodes on itself, you can fix nodes for the sublayouts individually by applying `setFixed` to the sublayout instances if the sublayouts support this feature:

```
IlvMultipleLayout multipleLayout =
        new IlvMultipleLayout(layout1, layout2, null);
multipleLayout.attach(grapher);
if (layout1.supportsPreserveFixedNodes()) {
        layout1.setFixed(node1, true);
        ...
}
if (layout2.supportsPreserveFixedNodes()) {
        layout2.setFixed(node2, true);
        ...
}
try {
        // perform a multiple layout: node1 is fixed while layout1 runs
        // and node2 is fixed while layout2 runs
        IlvMultipleLayoutReport layoutReport =
                (IlvMultipleLayoutReport)multipleLayout.performLayout(
                                                        true, true, true);
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Accessing sublayouts

You can obtain the sublayouts of a Multiple Layout instance by the following methods:

- `getFirstGraphLayout()`

  which returns the graph layout that is applied first.

- `getSecondGraphLayout()`

  which returns the graph layout that is applied second.

- `getLabelLayout()`

  which returns the label layout that is applied last.

You can also change the sublayouts. Of course, you should not change the sublayouts while the Multiple Layout instance is attached to a graph. You should detach the graph first.

To set the sublayouts, the following methods are available:

```
void setFirstGraphLayout(IlvGraphLayout layout)
```

```
void setSecondGraphLayout(IlvGraphLayout layout)
```

```
void setLabelLayout(IlvLabelLayout layout)
```

For experts: attaching graph and labeling models

If a graph model is attached to the Multiple Layout instance, the same graph model is automatically attached to the sublayouts of type `IlvGraphModel`. For the sublayout of type `IlvLabelLayout`, a default labeling model is used when possible (`IlvDefaultLabelingModel`, see *Labels and obstacles in Java*). This works if the nodes, links, and labels are stored in an `IlvGrapher`.

If you implement your own labeling model (subclass of `IlvLabelingModel`), you can force the Multiple Layout to use this labeling model instead of the default labeling model. Before you attach the graph model, you call the method `setLabelingModel` in the following way:

```
multipleLayout.setLabelingModel(myLabelingModel);
multipleLayout.attach(myGraphModel);
```

You must specify the labeling model in this way if your nodes, links, and labels are not stored in an `IlvGrapher` but in your own data structures, because the default labeling model is designed to handle only an`IlvGrapher`.

If the Multiple Layout instance is detached from the graph model, all sublayouts are automatically detached as well.

Combining multiple and recursive layout

Often, the Multiple Layout is used inside a Recursive Layout. For convenience, IBM® ILOG® JViews provides a layout algorithm that combines both mechanisms: the Recursive Multiple Layout. This is a Recursive Layout (see *Recursive layout*) that uses an instance of Multiple Layout for each subgraph.

To apply a Tree Layout, a Link Layout, and an Annealing Label Layout recursively on a nested graph, you can use:

```
IlvRecursiveMultipleLayout layout = new IlvRecursiveMultipleLayout(
                                  new IlvTreeLayout(),
                                  new IlvLinkLayout(),
                                           new IlvAnnealingLabelLayout());
```

This is in principle the same as a Recursive Layout that has a Multiple Layout as a reference layout:

```
IlvRecursiveLayout layout = new IlvRecursiveLayout(
                            new IlvMultipleLayout(
                            new IlvTreeLayout(),
                            new IlvLinkLayout(),
                                   new IlvAnnealingLabelLayout()));
```

The Recursive Multiple Layout has a first and second graph layout instance per subgraph, and a label layout instance per subgraph. You access these instances by the following methods:

- `IlvGraphLayout getFirstGraphLayout(Object subgraph)`

  which returns the graph layout that is applied first to the subgraph.

- `IlvGraphLayout getSecondGraphLayout(Object subgraph)`

  which returns the graph layout that is applied secondly to the subgraph.

- `IlvLabelLayout getLabelLayout(Object subgraph)`

  which returns the label layout that is applied last to the subgraph.

If the `subgraph` parameter is `null` in these methods, the layout instances of the top-level graph are returned.

For experts: the reference labeling model

The Recursive Multiple Layout must be used when the label layout should use a specified labeling model that is not the default labeling model (`IlvDefaultLabelingModel`, see *Labels and obstacles in Java*). The Multiple Layout allows you to specify a particular labeling model by using the method `setLabelingModel`, but when you encapsulate the Multiple Layout into a Recursive Layout, this specification would need to be repeated for each layout instance of each subgraph. This would be inconvenient. However, the Recursive Multiple Layout takes care of this mechanism automatically.

If you implement your own labeling model (subclass of `IlvLabelingModel`), you must implement the method:

```
IlvLabelingModel createLabelingModel(Object subgraph)
```

This method should return a new instance of your own labeling model for the input subgraph. The Recursive Multiple Layout uses this method to generate all labeling models for all subgraph from a reference labeling model. Before attaching the Recursive Multiple Layout instance, you can set the reference labeling model in the following way:

```
recursiveMultipleLayout.setReferenceLabelingModel(myLabelingModel);
recursiveMultipleLayout.attach(myGraphModel);
```

The reference labeling model is used for the label layout of the top-level grapher. Clones of the reference labeling model obtained by `createLabelingModel` are used for the label layouts of the subgraphers.

A simple way to perform a label layout recursively is the following:

```
IlvRecursiveMultipleLayout layout =
    new IlvRecursiveMultipleLayout(labelLayout);
layout.setReferenceLabelingModel(myLabelingModel);
layout.attach(topLevelGraph);
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();
        ...
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
layout.detach();
```

If the Recursive Multiple Layout instance is detached from the top level graph model, all sublayouts are automatically detached as well and all labeling models of subgraphs (including the reference labeling model) are disposed of.

Automatic label placement

Describes the label placement algorithms.

In this section

Getting started with labeling

Provides information to get started using the Label Layout framework.

In this section

Introduction to automatic label placement

A label placement algorithm is not a graph layout algorithm in the sense that it does not use the class IlvGraphModel and it is not a subclass of IlvGraphLayout. However, the labeling framework has many similarities to the graph layout framework.

Usually, a *label* is a text or decoration that should be placed close to some graphic object because the label denotes the meaning of the graphic object. The label should not be overlapped by obstacles because this would make it unreadable. If there are many potential positions for one label, the automatic label placement algorithm should find the best position for the label, so that it is close to its graphic object, not too close to unrelated objects, and not overlapped by any obstacle.

Labels occur in many application areas: diagrams, geographic maps, charts, and so on. The Labeling Layout framework can be applied to all these areas, that is, it is not restricted to placing labels in a graph. For example, if you want to label cities in a geographic map, you can use the Annealing Label Layout algorithm with appropriate point label descriptors. For simplicity, this topic shows how to place labels at links in a graph.

The label layout framework distinguishes between:

♦ Graphic objects called *labels* that must be placed

♦ Graphic objects called *obstacles* that must not move but occupy space that is not available for the labels

♦ Graphic objects that are *ignored*, that is, they are not moved nor considered as occupying space; every graphic object that is neither a label nor an obstacle is ignored

A label should not overlap any obstacle or other label. In fact, this strict condition is not feasible if there is not enough free space for the labels; in this situation the label layout algorithm tries to reduce the amount of overlaps.

In IBM® ILOG® JViews, graphic objects (subclasses of IlvGraphic) are contained in a manager (IlvManager), and graphs are managed by graphers (IlvGrapher, which is a subclass of IlvManager). Because the Label Layout algorithms can be applied not only to graphs but also to any graphic objects, the algorithms work on IlvManager. Consequently, a label layout algorithm is not a subclass of IlvGraphLayout and *does not use* an IlvGraphModel. IBM® ILOG® JViews Diagrammer provides a label layout framework that is (despite many similarities) completely independent from the graph layout framework.

IBM® ILOG® JViews Diagrammer provides a Swing component that encapsulates a manager and the view that displays the manager. It uses a model-view architecture, that is, application objects must be provided as an SDM model, and a style sheet (CSS file) describes how the corresponding graphic objects are added to the manager and are displayed in the view.

You can use the Label Layout algorithm in the following ways:

♦ **In a diagram component:** You specify the label layout in CSS format. The diagram component loads this specification and automatically applies the label layout when necessary. The diagram component recognizes only labels of nodes and links of the types IlvGeneralNode, IlvGeneralLink, IlvSDMCompositeNode, and IlvSDMCompositeLink.

♦ **In an application that uses IBM® ILOG® JViews managers:** Instead of using style files, you access the API of the label layout directly. This is suitable for applications that do not need the model-view architecture. It is necessary for applications that requires the automatic placement of labels that are not part of `IlvGeneralNode`, `IlvGeneralLink`, `IlvSDMCompositeNode`, and `IlvSDMCompositeLink`.

If you have implemented your own data structures or use third-party data structures that represent labels, you can provide an adapter between your data structures and the labeling model. This complex application of the label layout package is explained in *Defining your own labeling model*.

Getting started with Label Layout in a diagram component

For information on how to load a style file into the diagram component and how to access the node layout, link layout, and label layout renderers, see *Basic concepts*.

To support labels in CSS:

1. Write a style rule specifying that links have labels.

   The following CSS specification indicates that each link has one label with the text coming from the “*name*” property of the model links of the SDM data model:

   ```
   link {
       class: "ilog.views.sdm.graphic.IlvGeneralLink";
       label: "@name";
   }
   ```

2. Specify, for example, that each label should be placed at the left side of the corresponding link:

   ```
   link:labelLayout {
       allowedSide: "Left";
       sideAssociation: "GLOBAL";
   }
   ```

3. To enable label layout, specify the label layout renderer:

   ```
   SMD {
       LabelLayout: "true";
   }
   ```

4. Set global label layout parameters in the `Label Layout` section. For example, the following specification defines the minimum offsets between pairs of labels (5) and between a label and an obstacle (10):

   ```
   LabelLayout {
       labelOffset: "5";
       obstacleOffset: "10";
   }
   ```

When a style file with these CSS specifications is loaded into the diagram component, the Annealing Label layout is automatically performed. The label layout renderer always uses the Annealing Label layout, therefore it is not necessary to specify *which* label layout must be performed.

The complete CSS specification and a simple application that loads the specification and performs a label layout are available as a sample at: **<installdir>/jviews-diagrammer 86/codefragments/labellayout**

Getting started with Label Layout in Java

To obtain a label layout, apply the Annealing Label Layout algorithm to a manager directly.

To apply the layout to a manager:

1. Create a manager object (`IlvManager`) and fill it with obstacles and labels. For instance, if you want to place labels at the links of a graph, create a grapher and fill it with nodes and links (the obstacles) and additionally with the labels that should be placed at the links. In this case, the labels should be added using `addObject(ilog.views.IlvGraphic, boolean)` instead of `addNode(ilog.views.IlvGraphic, boolean)` because the labels should not be nodes at the same time.

   Unlike with diagram component and style sheets, you are not restricted to using `IlvGeneralLink`, or `IlvSDMCompositeLink`. You can place labels at any link.

2. Create an instance of the Annealing Label Layout algorithm.

3. Declare a handle for the corresponding layout report. The layout report is an object in which the layout algorithm stores information about its behavior. For details, see *The label layout report*.

4. Attach the manager to the layout instance.

   Here, it is assumed that the labels are subclasses of `IlvLabel`, `IlvZoomableLabel` or `IlvText`. If they are not, the label model can be extended as illustrated in *Labels and obstacles in Java*.

5. For each label, set a label descriptor.

   The label descriptor describes the conditions for the placement of the label. For instance, if a label should be placed close to the source or destination node of a link, use an `IlvAnnealingPolylineLabelDescriptor` with the corresponding options. For details, see *Label descriptors*.

6. Modify the default settings for the layout parameters, if needed.

7. Call the `performLayout` method.

8. Read and display information from the layout report.

9. When the layout instance is no longer needed, detach the manager from the layout instance.

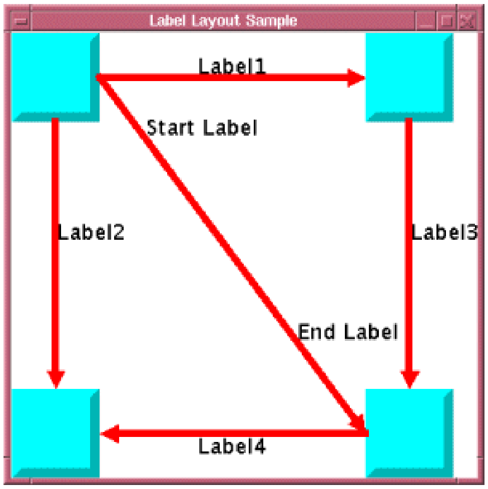
A complete example of these steps can be found at the location
<installdir>/jviews-diagrammer 86/codefragments/labellayout

It contains the following Java code:

```
// the JViews Graphic Framework
import ilog.views.*;
import ilog.views.graphic.*;

// the JViews Label Layout Framework
import ilog.views.graphlayout.labellayout.*;
```

```
// the JViews Annealing Label Layout
import ilog.views.graphlayout.labellayout.annealing.*;

// the Java AWT package
import java.awt.*;

// the Swing package
import javax.swing.*;


/**
 * A very simple example for the use of a Layout Algorithm.
 */
public class LayoutSample2
{
  public static final void main(String[] arg)
  {
    SwingUtilities.invokeLater(new Runnable() {
      public void run() {
       // Create the manager instance (subclass of IlvManager). Since we want

        // to place link labels, we use a grapher in this example.
        IlvGrapher grapher = new IlvGrapher();

        // Create the manager view instance
        IlvManagerView view = new IlvManagerView(grapher);
        view.setBackground(Color.white);

        // An Swing Frame to display
        JFrame frame = new JFrame("Label Layout Sample");

        // Put the manager view inside an AWT Frame and show it
        frame.getContentPane().add(view);
        frame.setSize(400, 400);
        frame.setVisible(true);

        // Fill the grapher with nodes and links
        IlvReliefRectangle node1 = new IlvReliefRectangle(
                                   new IlvRect(0f, 0f, 50f, 50f));
        IlvReliefRectangle node2 = new IlvReliefRectangle(
                                   new IlvRect(200f, 0f, 50f, 50f));
        IlvReliefRectangle node3 = new IlvReliefRectangle(
                                   new IlvRect(0f, 200f, 50f, 50f));
        IlvReliefRectangle node4 = new IlvReliefRectangle(
                                   new IlvRect(200f, 200f, 50f, 50f));
        grapher.addNode(node1, false);
        grapher.addNode(node2, false);
        grapher.addNode(node3, false);
        grapher.addNode(node4, false);

        // set some nice colors
        setNodeColors(node1, node2, node3, node4);

        IlvLinkImage link1 = new IlvLinkImage(node1, node2, true);
        IlvLinkImage link2 = new IlvLinkImage(node1, node3, true);
        IlvLinkImage link3 = new IlvLinkImage(node2, node4, true);
        IlvLinkImage link4 = new IlvLinkImage(node4, node3, true);
        IlvLinkImage link5 = new IlvLinkImage(node1, node4, true);

        grapher.addLink(link1, false);
        grapher.addLink(link2, false);
        grapher.addLink(link3, false);
        grapher.addLink(link4, false);
        grapher.addLink(link5, false);

        // set some nice colors
        setLinkColors(link1, link2, link3, link4, link5);

        // Add labels. Labels are neither "nodes" nor "links", hence add them

        // as objects. Since we perform layout later, the initial position
        // doesn't play a role.

        IlvPoint p = new IlvPoint(0f,0f);
        IlvZoomableLabel label1 = new IlvZoomableLabel(p, "Label1");
        IlvZoomableLabel label2 = new IlvZoomableLabel(p, "Label2");
        IlvZoomableLabel label3 = new IlvZoomableLabel(p, "Label3");
        IlvZoomableLabel label4 = new IlvZoomableLabel(p, "Label4");
        IlvZoomableLabel label5 = new IlvZoomableLabel(p, "Start Label");
        IlvZoomableLabel label6 = new IlvZoomableLabel(p, "End Label");

        grapher.addObject(label1, false);
        grapher.addObject(label2, false);
        grapher.addObject(label3, false);
        grapher.addObject(label4, false);
        grapher.addObject(label5, false);
        grapher.addObject(label6, false);


        // Declare a handle for the layout instance
        IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();

        // Declare a handle for the layout report
        IlvLabelLayoutReport layoutReport = null;

        // Attach the manager to the layout instance
        layout.attach(grapher);

        // For each label, set a label descriptor that specifies: label1
          //  belongs to link1, label2 belongs to link2, and so on. link5 has

        // 2 labels.

        layout.setLabelDescriptor(
            label1,
            new IlvAnnealingPolylineLabelDescriptor(
                label1, link1, IlvAnnealingPolylineLabelDescriptor.CENTER));
        layout.setLabelDescriptor(
            label2,
            new IlvAnnealingPolylineLabelDescriptor(
                label2, link2, IlvAnnealingPolylineLabelDescriptor.CENTER));
        layout.setLabelDescriptor(
            label3,
            new IlvAnnealingPolylineLabelDescriptor(
                label3, link3, IlvAnnealingPolylineLabelDescriptor.CENTER));
        layout.setLabelDescriptor(
            label4,
            new IlvAnnealingPolylineLabelDescriptor(
                label4, link4, IlvAnnealingPolylineLabelDescriptor.CENTER));
        layout.setLabelDescriptor(
            label5,
            new IlvAnnealingPolylineLabelDescriptor(
                label5, link5, IlvAnnealingPolylineLabelDescriptor.START));
        layout.setLabelDescriptor(
            label6,
            new IlvAnnealingPolylineLabelDescriptor(
                label6, link5, IlvAnnealingPolylineLabelDescriptor.END));

        // Modify the layout parameters, if needed
        layout.setLabelOffset(10);
        layout.setObstacleOffset(5);

        // Perform the layout and get the layout report
        layoutReport = layout.performLayout();

        // Print information from the layout report (optional)
        System.out.println("layout done in " +
                       layoutReport.getLayoutTime() +
                       " millisec., code = " +
                       layoutReport.getCode());

        // Fit the graph in the window
        view.fitTransformerToContent();
        // Redraw the grapher
        grapher.reDraw();

        // Detach the grapher from the layout instance
        layout.detach();
      }
    });
  }
```

The following figure shows the graph produced by the sample Java™ application.



Output from Sample Java Application

Specifying labels and obstacles

Explains what labels and obstacles are.

In this section

Labels and obstacles in a diagram component

IBM® ILOG® JViews Diagrammer uses an SDM data model that distinguishes between nodes and links. Labels can belong to nodes or to links. The label layout considers the nodes and links themselves as obstacles. However, only nodes of type `IlvGeneralNode` and `IlvSDMCompositeNode`, and only links of type `IlvGeneralLink` and `IlvSDMCompositeLink`. can have labels.

Labels at nodes

Each instance of `IlvGeneralNode` has at most one label. This label can be placed by a hard-coded mechanism of `IlvGeneralNode` which does not take into account the free space, or it can be placed by the label layout renderer. The parameter `labelPosition` of the class `IlvGeneralNode` enables you to indicate how the label is placed. If the label position is set to:

- ♦ “`Center`”: the label is placed by the hard-coded mechanism at the center of the node. It is not taken into account by the label layout.
- ♦ “`Left`”, “`Right`”, “`Top`”, “`Bottom`”: the label is placed by the label layout renderer if the label layout renderer is enabled, otherwise it is placed by the `IlvGeneralNode` hard-coded mechanism.

The label can be implemented internally by the class `IlvZoomableLabel` or `IlvText` and can be specified as zoomable or scaled relative to the node size. The following is a sample specification of a node with a label:

```
node {
    class : "ilog.views.sdm.graphic.IlvGeneralNode";
    label : "@name";
    labelScaleFactor: "0.5";
    labelFont: "PLAIN-BOLD-12";
    labelAntialiasing: "true";
    labelZoomable:     "true";
    useIlvText:        "true";

    labelPosition : "Left";
    labelSpacing : "2.0";
}
```

This node uses the class `IlvGeneralNode` with a label of type `IlvText`. The label is zoomable, that is, it scales according to the zoom factor. The label text is taken from the SDM model property `name`. The `labelPosition` parameter specifies that the label must be placed to the left of the node, and the `labelSpacing` parameter indicates that the maximal distance of the label from the node is 2 units.

In order to specify further layout parameters of the label, use the pseudo class `labelLayout`:

```
node:labelLayout {
    ... any option of the IlvAnnealingPointLabelDescriptor ...
}
```

In particular, you can specify whether the layout should treat or ignore the label:

```
node:labelLayout {
    treatAs: "IGNORED";
}
```

The `treatAs` parameter can take the following values:

- `LABEL` - the label is positioned by the label layout.
- `OBSTACLE` - the label is not moved by the label layout, but it occupies space that cannot be used by other labels
- `IGNORED` - the label is ignored even though the node that owns the label is not ignored.

Labels at links

Each instance of `IlvGeneralLink` has at most one label. This label can be placed by a hard-coded mechanism of `IlvGeneralLink` which does not take into account the free space, or it can be placed by the label layout renderer. If the label layout renderer is enabled, the label is always placed by the label layout.

The general link can have an arbitrary set of decorations. The label is usually the first of these decorations—that is, the decoration with index 0. Any decoration that implements the interface `IlvLabelInterface` can be used as a label. If no such decoration is specified, it is automatically generated if a label is required.

The following is a sample specification of a link with a label:

```
link {
    class : "ilog.views.sdm.graphic.IlvGeneralLink";
    label : "@name";
}
```

The label text is taken from the SDM model property `name`. Since no decoration is specified, an instance of `IlvZoomableLabel` is automatically generated as the decoration with index 0. If you want to use a different class for the label, or if you want the label to have a specific graphic property such as a background color, you must specify the decoration explicitly:

```
link {
    class : "ilog.views.sdm.graphic.IlvGeneralLink";
    decorations[0]: "@+deco";
    label : "@name";
    decorationPositions[0]: "0.8";
}
Subobject#deco {
    class: "ilog.views.sdm.graphic.IlvGraphicFactories$ZoomableLabel";
    IlvRect:        "0,0,5,5";
    backgroundOn:   "true";
    background:     "#BABABA";
}
```

The “`Subobject#deco`” rule specifies the graphic properties of the label. This label has a special background color. The declaration “`decorationPositions[0]`” specifies where the label attaches the link. If the label layout renderer is disabled, the label is placed in a hard-coded way at this position. If the label layout renderer is enabled, the label is placed approximately at this position, but also tries to avoid overlaps with other labels or obstacles.

General links may have many decorations, but the label layout recognizes only the first decoration of type `IlvLabelInterface` as a label, recognizes the general link itself as an obstacle, and ignores all other decorations of the general link.

In order to specify specific label parameters for the label layout, use the pseudo class “`labelLayout`”:

```
link:labelLayout {
    ... any option of the IlvAnnealingPolylineLabelDescriptor ...
}
```

In particular, you can specify whether the layout should treat or ignore the label:

```
link:labelLayout {
    treatAs: "IGNORED";
}
```

The `treatAs` parameter can take the values `LABEL`, `OBSTACLE`, or `IGNORED`, as explained in *Labels at nodes*.

Labels at composite nodes

If you need nodes and links that can have multiple labels, you must use the classes `IlvSDMCompositeNode` and `IlvSDMCompositeLink`. Composite nodes and links can have any number of decorations (children). The label layout recognizes all children of type `IlvLabel`, `IlvZoomableLabel` or `IlvText` as labels, and all other children as obstacles.

The following example shows how to specify labels at composite nodes:

```
node {
    class: "ilog.views.sdm.graphic.IlvSDMCompositeNode";
    children[0]:    "@+base";
    children[1]:    "@+label1";
    children[2]:    "@+label2";

    constraints[1]:@+attachmentLabel;
    constraints[2]:@+attachmentLabel;
}

#base {
   class:"ilog.views.graphic.IlvRectangle(definitionRect)";
   definitionRect:"@|rect(0,0,60,60)";
}

Subobject#label1 {
   class:"ilog.views.graphic.IlvText";
   label:@name;

}

Subobject#label2 {
   class:"ilog.views.graphic.IlvText";
   label:"hello";

}

Subobject#attachmentLabel {
   class:"ilog.views.graphic.composite.layout.IlvAttachmentConstraint";
   hotSpot:TopCenter;
   anchor:BottomCenter;
   offset: "0,3";
}
```

The composite node has three children: The first child (index 0, rule `#base`) is the basic shape of the node. The other two children (index 1 and 2, rules `Subobject#label1` and `Subobject#label2`) are the labels of the node. The first node label `label1` uses the SDM model property `name` as text while `label2` uses the constant string “`Hello`”. Composite nodes and links require children to be attached by attachment constraints. See Building composite nodes in CSS in *JViews Diagrammer SDK*.

The label layout treats the basic shape as an obstacle, and children with index 1 and 2 as labels. In order to specify label parameters for the label layout, use the pseudo classes “`labelLayout`” and “`labelLayout_`*i*” where *i* is the child index of the corresponding label. This declaration applies to *all* labels of the node:

```
node:labelLayout {
    ... any option of the IlvAnnealingPointLabelDescriptor ...
}
```

This declaration applies to the label with child index 2 of node.

```
node:labelLayout_2 {
    ... any option of the IlvAnnealingPointLabelDescriptor ...
}
```

In particular, you can specify whether layout should treat or ignore the label:

```
node:labelLayout_2 {
    treatAs: "IGNORED";
}
```

The `treatAs` parameter can take the values `LABEL`, `OBSTACLE` and, or `IGNORED`, as explained in section *Labels at nodes*.

Labels at composite links

Setting labels on composite links is very similar to setting labels on composite nodes (see *Labels at composite nodes*, except that the values of the `hotSpot` and `anchor` parameters of the `Subobject#attachmentLabel` rule are different, as shown in the following example.

```
link {
      class: "ilog.views.sdm.graphic.IlvSDMCompositeLink";
      children[0]:    "@+label1";
      children[1]:    "@+label2";
      constraints[0]: "@+attachmentLabel";
      constraints[1]: "@+attachmentLabel";
}
#base {
   class:"ilog.views.graphic.IlvRectangle(definitionRect)";
   definitionRect:"@|rect(0,0,60,60)";

}

Subobject#label1 {
   class:"ilog.views.graphic.IlvText";
   label:@name;

}

Subobject#label2 {
   class:"ilog.views.graphic.IlvText";
   label:"hello";

}

Subobject#attachmentLabel {
   class:"ilog.views.graphic.composite.layout.IlvAttachmentConstraint";
      hotSpot: "FromLink";
      anchor:  "Center";
       offset:  "0,0";
}
```

Ignoring nodes or links

The following CSS sample shows how to ignore a node and all its labels. This means that the label layout algorithm does not treat the node as an obstacle and does not try to place its labels. The label layout algorithm behaves as if the object did not exist:

```
#node23 {
    LabelLayoutIgnored: "true";
}
```

Similarly, you can specify `LabelLayoutIgnored` for link in order to ignore the link and all its labels.

Note: The CSS syntax is case-sensitive: `LabelLayoutIgnored` starts with an uppercase letter, while most other label layout declarations start with a lowercase letter.

Labels and obstacles in Java

The diagram component works with a predefined, complex labeling model that recognizes labels at instances of `IlvGeneralNode`, `IlvSDMCompositeNode`, `IlvGeneralLink` and `IlvSDMCompositeLink`. If you use other classes or if you want to place labels at other objects, you cannot use the diagram component and you have to code the labeling model.

The `IlvLabelingModel` class defines a suitable, generic API to position labels automatically through an IBM® ILOG® JViews Diagrammer label layout algorithm. Its purpose is similar to `IlvGraphModel` for the graph layout framework.

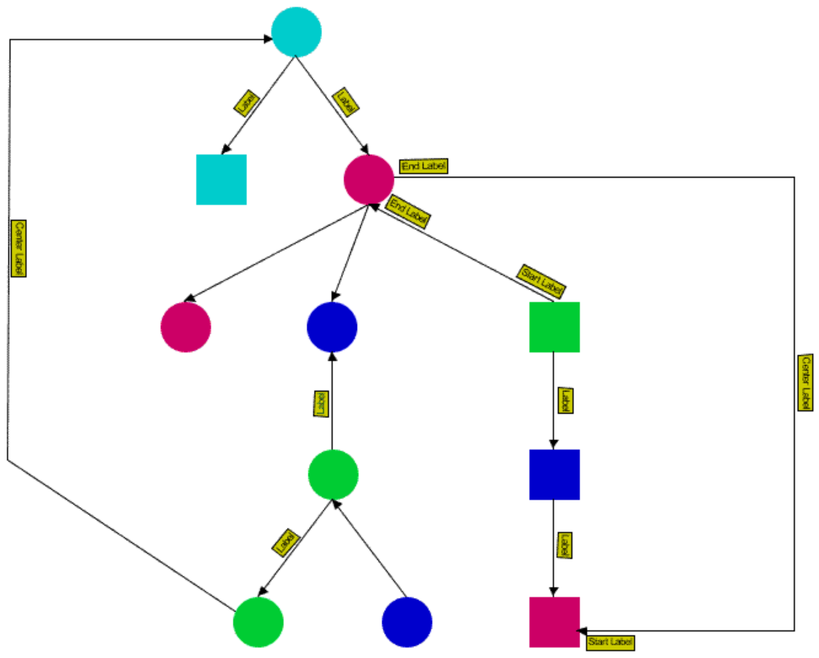
The class `IlvLabelingModel` is an abstract base class that allows you to implement an adapter to your own data structures for the labeling algorithm (see *Defining your own labeling model*). To get the labeling model of an attached layout, use the method:

```
layout.getLabelingModel();
```

If labels and obstacles are contained in an `IlvManager` instance, you do not need to implement an adapter. You can use the default labeling model (the class `IlvDefaultLabelingModel`).

The default labeling model is suitable when the labels are upright rectangular objects and the overlap calculation considers their bounding boxes.

It is also suitable for rectangular labels that have a rotation that depends on the position of a label. For example, the label on a polyline link may be rotated to align the label with the gradient of the line segment of a link. The overlap calculation considers the rotated rectangle of the label in this case.



Rotation of rectangular labels (rotation dependent on the position of the labels)

The default labeling model may not be suitable when the labels have a nonrectangular shape that leaves large parts of the bounding box of the label empty. The overlap detection will wrongly assume that the empty space inside the bounding box causes overlaps

Note: The default labeling model is not suitable for `IlvGeneralNode`, `IlvGeneralLink`, `IlvSDMCompositeNode`, `IlvSDMCompositeLink`, because these classes implement labels that are not directly contained in a manager. Therefore the diagram component uses a specialized labeling model.

The default labeling model considers subclasses of `IlvLabel`, `IlvZoomableLabel` and `IlvText` as labels and all other objects as obstacles. This is the most common case. However, you can redefine this meaning:

♦ If you want a graphic to be handled as a label even though it is not a subclass of `IlvLabel` or `IlvZoomableLabel`, call:

```
defaultLabelingModel.setLabel(graphic, true);
```

- If you do not want to handle an instance of `IlvLabel`, `IlvZoomableLabel` or `IlvText`as a label but as an immovable obstacle, call:

  ```
  defaultLabelingModel.setObstacle(graphic, true);
  ```

- If you want a graphic to be ignored by the labeling model, call:

  ```
  defaultLabelingModel.setLabel(graphic, false);
  defaultLabelingModel.setObstacle(graphic, false);
  ```

  This is useful particularly if you want to create graphics that act as background images and labels should be placed on top of the background images. If the background images were obstacles, the label layout algorithm would try to avoid the area covered by the background images.

The overlap calculation of the default labeling model takes the bounding box of the obstacles into account. This is a good approximation for the most objects. For subclasses of `IlvLine`, `IlvPolyline`, and `IlvLinkImage`, it takes the precise polyline shape into account. However, the default labeling model does not work well with spline obstacles.

Using the label layout API

Describes how to perform a label layout.

In this section

Overview

You will see how to perform a label layout in Java™ when working directly on an `IlvManager` instance. If you are using IBM® ILOG® JViews Diagrammer with style sheets, see *Controlling layout renderers by style sheets*.

Note: Before reading this information, you should be familiar with the `IlvGraphLayout` class (see *Using the graph layout API*). Many of the concepts for the labeling layout mechanism are similar and not all details are repeated in this topic.

The label layout base class and its subclasses

The `IlvLabelLayout` class is the base class for all label layout algorithms. This class is an abstract class and cannot be used directly.

Subclasses of IlvLabelLayout

There are currently two subclasses:

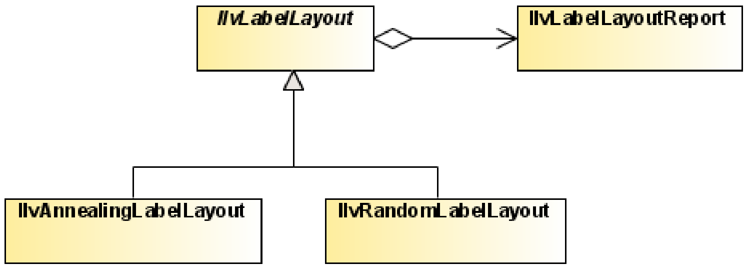
- ♦ `IlvRandomLabelLayout` which randomizes the label positions for demonstration purpose.
- ♦ `IlvAnnealingLabelLayout` which carries out real label arrangement.

You can also create your own subclasses to implement other label layout algorithms.

The label layout renderer always uses the `IlvAnnealingLabelLayout`. You can use `IlvRandomLayout` or your own label layout algorithm when you write Java™ code.

It is not possible to use `IlvRandomLayout` or your own label layout algorithm in a diagram component based on CSS styling.

Despite the fact that only subclasses of `IlvLabelLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by its subclasses. And, of course, you will need to understand it if you subclass it yourself.



The class IlvLabelLayout and its subclasses and relationship to layout reports

Instantiating and attaching a subclass of IlvLabelLayout

The class `IlvLabelLayout` is an abstract class. It has no constructors. You will instantiate a subclass as shown in the following example:

```
IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();
```

In order to place labels, a manager needs to be attached to the layout instance. The following method, defined on the class `IlvLabelLayout`, allows you to specify the manager you want to lay out:

```
void attach(IlvManager manager)
```

For example:

```
...
IlvManager manager = new IlvManager();
/* Add obstacles and labels to the manager here */
layout.attach(manager);
```

The `attach` method does nothing if the specified manager is already attached. If a different manager is attached, this method first detaches this old manager, then attaches the new one. You can obtain the attached manager using the method `getManager`. If the manager is attached in this way, a default labeling model is created internally. The labeling model can be obtained by:

```
IlvLabelingModel labelingModel = layout.getLabelingModel();
```

Warning: It is not allowed to attach any such model created internally to any other layout instance.

After layout, when you no longer need the layout instance, you should call the method

```
void detach()
```

If the `detach` method is not called, some objects may not be garbage-collected. This method also performs cleaning operations on the manager (properties that may have been added by the layout algorithm on objects of the manager.

Performing a layout

The `performLayout` methods start the layout algorithm using the currently attached manager and the current settings for the layout parameters. The method returns a report object that contains information about the behavior of the label layout algorithm.

```
IlvLabelLayoutReport performLayout()
IlvLabelLayoutReport performLayout(boolean force, boolean redraw)
```

The first method simply calls the second one with the `force` argument set to `false` and the `redraw` argument set to`true`.

- ♦ Because the `force` argument is set to `false` (by default), the layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the manager or any of the parameters have changed since the last time the layout was successfully performed. A "change" can be any of the following:

  - Obstacles or labels were added or removed.
  - Obstacles or labels were moved or reshaped.
  - The value of a layout parameter was modified.
  - The transformer changed while nonzoomable obstacles or labels were used.

  Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `performLayout` returns without performing the layout. If the argument `force` is passed as `true`, the verification is skipped, and layout is performed even if no changes occurred.

- ♦ The `redraw` argument determines whether the manager needs to be redrawn. This mechanism works exactly in the same way as in graph layout. For details, see *Redrawing the grapher after layout*.

  The protected abstract method `layout(boolean redraw)` is then called. This means that control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the labels to new positions.

Performing a recursive layout on nested subgraphs

The examples and explanations above assume that you work with a flat manager. IBM® ILOG® JViews Diagrammer allows you to nest a grapher as a node into another grapher. You can create a hierarchy of nested graphers (see the following figure); see also Nested graphers in *Advanced Features of JViews Framework*).

You can apply a recursive graph layout to the nested grapher hierarchy by calling:

```
graphLayout.performLayout(true, true, true);
```

However, it usually makes no sense to apply a label layout alone to nested graphers. When labels are placed in a subgrapher, this will likely change the bounds of the subgrapher; hence the node positions in its parent grapher will no longer be up-to-date and a new graph layout will be necessary.

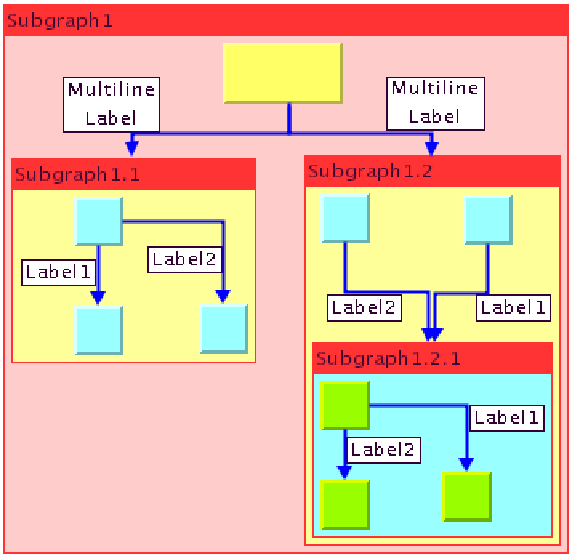
It makes sense to apply a label layout in combination with another graph layout to nested graphers.

1. First, the graph layout is applied to arrange the nodes and links nicely.
2. Then the label layout is applied to position the labels according to the node and link positions.
3. When this is finished for all subgraphers, then it can be done for the parent grapher.

To perform a graph layout and a label layout together, you can use the Multiple Layout class. This is a subclass of `IlvGraphLayout` that allows combining graph layouts with a label layout. The following sample shows how to apply a Tree Layout and an Annealing Label Layout in combination on a subgrapher.

```
IlvTreeLayout treeLayout = new IlvTreeLayout();
IlvAnnealingLabelLayout labelLayout = new IlvAnnealingLabelLayout();
IlvGraphLayout multipleLayout = new IlvMultipleLayout(treeLayout,
                                                      null,
                                                      labelLayout);
// Now set the parameters for tree layout and label layout ...
// Finally, perform a recursive layout that handles tree layout and label
// layout together
try {
    multipleLayout.performLayout(true, true, true);
} catch (IlvGraphLayoutException e) {
    ...
}
```

Thus, the label layout does not provide a separate mechanism for a recursive layout on submanagers. By incorporating the label layout into a multiple graph layout, you can use all the graph layout facilities that are available for nested graphs (see also *Nested layouts*).



Nested subgraphers with labels

The label layout report

The label layout report contains information about the particular behavior of a label layout algorithm. After the layout is completed, this information is available for reading from the label layout report. The information can also be obtained during layout by using a layout listener, as described in *Layout events and listeners*.

The layout report is created automatically at the start of layout via the method `createLayoutReport` and is available as long as the manager is attached to the layout instance.

To read a layout report, all you need to do is store the layout report instance returned by the `performLayout` method and read the information, as shown in the following example:

```
...
IlvLabelLayoutReport layoutReport = labelLayout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
     System.out.println("Label layout done.");
else
     System.out.println("Label layout not done, code = " +
                                          layoutReport.getCode());
```

The class `IlvLabelLayoutReport` stores the following information, which is very similar to the information stored in an `IlvGraphLayoutReport` (see *Information stored in a layout report* for details):

Code

This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

- ♦ `IlvLabelLayoutReport.LAYOUT_DONE`
- ♦ `IlvLabelLayoutReport.STOPPED_AND_VALID`
- ♦ `IlvLabelLayoutReport.STOPPED_AND_INVALID`
- ♦ `IlvLabelLayoutReport.NOT_NEEDED`
- ♦ `IlvLabelLayoutReport.NO_LABELS`
- ♦ `IlvLabelLayoutReport.EXCEPTION_DURING_LAYOUT`

To read the code, use the method:

```
int getCode()
```

Layout time

This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

```
long getLayoutTime()
```

Percentage of completion

This field contains an estimate of the percentage of the layout that has been completed. To access the percentage, use the method:

```
int getPercentageComplete()
```

Layout events and listeners

The label layout framework provides the same event mechanism as the graph layout framework. Various events may occur.

Label layout events

The class `GraphLayoutEvent` corresponds to the class `GraphLayoutEvent` (see *Graph layout event listeners*). You can install a listener for these events at the layout instance by using the method:

```
labelLayout.addLabelLayoutEventListener(listener);
```

The listener must implement the `LabelLayoutEventListener` interface and receives events while the layout is running. A typical example is to check how much of the layout has already completed:

```
class MyLabelLayoutListener
  implements LabelLayoutEventListener
{
  public void layoutStepPerformed(LabelLayoutEvent event)
  {
    IlvLabelLayoutReport layoutReport = event.getLayoutReport();
    System.out.println("percentage of completion: " +
                       layoutReport.getPercentageComplete());
  }
}
```

Label layout parameter events

The class `LabelLayoutParameterEvent` corresponds to the class `GraphLayoutParameterEvent` (see *Parameter event listeners*). You can install a listener to these events at the layout instance by

```
labelLayout.addLabelLayoutParameterEventListener(listener);
```

The listener must implement the `LabelLayoutParameterEventListener` interface and receives events when layout parameters change. It also receives a special event at the end of a successful layout. For example:

```
class MyLabelLayoutParameterListener
  implements LabelLayoutParameterEventListener
{
  public void parametersUpToDate(LabelLayoutParameterEvent event)
  {
    if (!event.isParametersUpToDate())
      System.out.println("Any label layout parameter has changed.”);
```

```
   }
}
```

Layout parameters and features in IlvLabelLayout

The class `IlvLabelLayout` defines a number of generic features and parameters. These are a subset of the mechanism, methods, and parameters that are available for the `IlvGraphModel` class. Therefore, they are only listed here; for detailed explanations, refer to the appropriate subsection in *Generic parameters and features* which describes the corresponding features for the `IlvGraphLayout` class.

In CSS, the main difference is that you add the specification to the `LabelLayout` section (not to the `GraphLayout` section).

Although the `IlvLabelLayout` class defines the generic parameters, it does not control how they are used by its subclasses. Each label layout algorithm (that is, each subclass of `IlvLabelLayout`) supports a subset of the generic features and determines the way in which it uses the generic parameters. When you create your own label layout algorithm by subclassing `IlvLabelLayout`, you decide whether to use the features and the way in which you are going to use them.

The `IlvLabelLayout` class defines the following generic features:

♦ *Allowed time*

♦ *Percentage of completion calculation*

♦ *Random generator seed value*

♦ *Stop immediately*

♦ *Use default parameters*

To specify that the label layout is allowed to run for 60 seconds:

In CSS
Add to the `LabelLayout` section:

```
allowedTime: "60000";
```

In Java™
Call:

```
labelLayout.setAllowedTime(60000);
```

For more details of all generic features, see *Generic parameters and features*.

Annealing label layout

Describes the *Annealing Label Layout* algorithm (class `IlvAnnealingLabelLayout` from the package `ilog.views.graphlayout.labellayout.annealing`).

In this section

Rotated labels
Describes the rotated labels used by the Annealing Label Layout.

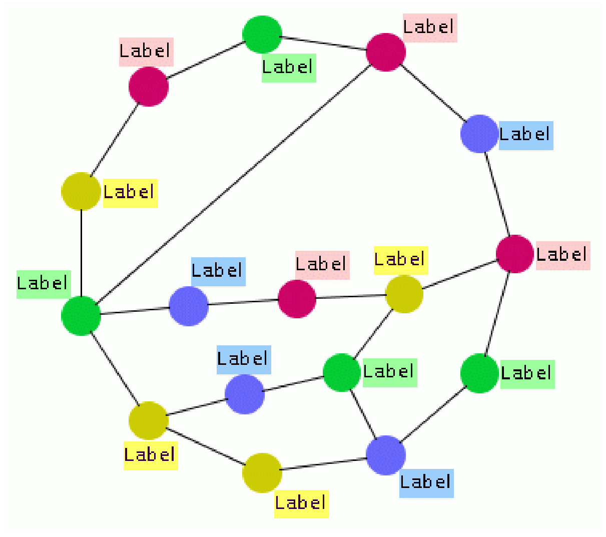
Specific global parameters
Describes the global parameters used by the Annealing Label Layout.

For experts: implementing your own label descriptors
Describes how to create a label descriptor for the Annealing Label Layout.

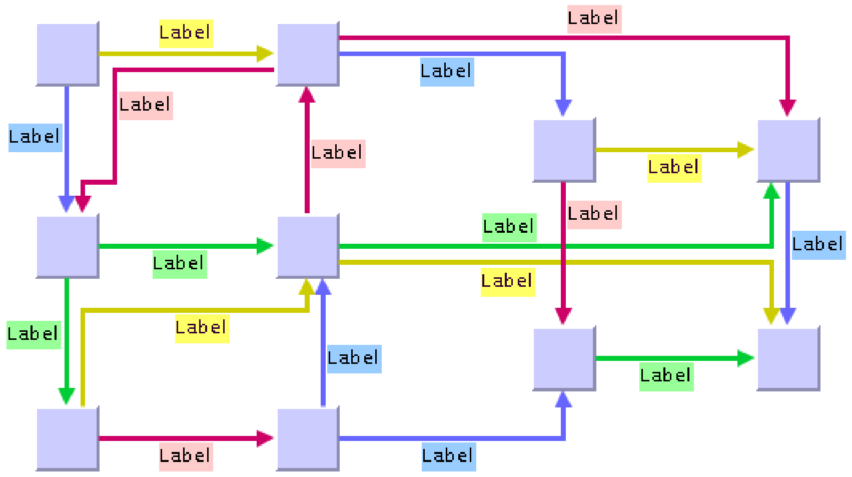
General information

The Annealing Label layout is used by the label layout renderer of a diagram component or can be used in Java™ code.

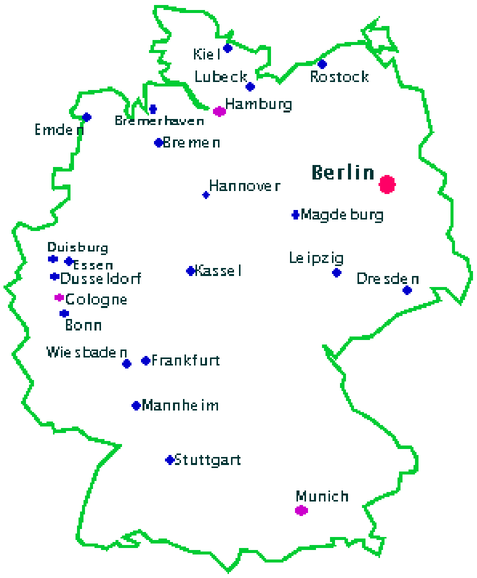
The following sample drawings are produced with the Annealing Label Layout.



Label placement at nodes with the Annealing Label Layout



Label placement at links with the Annealing Label Layout



Label placement at cities in a map of Germany with Annealing Label Layout

Features

- Places only labels. Does not move any obstacles around.
- Quality-controlled, randomized iterative heuristic.
- Can place labels at points, rectangles, ellipses, and polylines when used inside Java™ code.
- Can be used to place labels at any nodes and links when used inside Java code.
- Can be used to place labels at cities on a geographic map when used inside Java code.
- Can place multiple labels at the same object (point, node, link, city, and so on).
- Can handle upright rectangular labels and rectangular labels that have a rotation that depends on their position
- Tries to avoid overlaps among labels, and between labels and obstacles, by using the available free space.
- Provides several anchor and preference options.
- Easily extensible by subclassing label descriptors.
- Efficient, scalable algorithm. Produces nice label placements even with a large number of labels.

Extra feature for JViews Diagrammer:

- Can place labels at `IlvGeneralNode`, `IlvGeneralLink`, `IlvSDMCompositeNode`, and `IlvSDMCompositeLink` when used inside a diagram component with CSS styling.

Limitations

- The Annealing Label Layout algorithm, as a randomized iterative heuristic, does not guarantee that labels are placed without overlaps whenever possible. However, it produces a high quality layout with a high probability of minimum overlap. The more iterations, the higher the probability of high quality.
- The algorithm is not able to create free space for labels by moving obstacles around. It is recommended that you perform a graph layout algorithm with large spacing parameters to create the necessary free space before placing the labels.
- While the algorithm is able to place labels at straight and polyline graphics, it is not able to place labels precisely at smooth curves such as spline graphics (`IlvSpline`) or spline links (`IlvSplineLinkImage`, `IlvOneSplineLinkImage`, `IlvDoubleSplineLinkImage`). It is also not able to place the label correctly on `IlvGeneralLink`, if the `curved` option of `IlvGeneralLink` is used.

The algorithm

The algorithm uses *simulated annealing*. This is a general, randomized optimization technique that simulates a thermodynamic particle system. Each label is moved to a new random position within the limits given by its label descriptor. The quality of the new position is calculated and compared to the quality of the old position. If the quality has not improved, the label is moved back to the old position. The amount of movement is controlled by a conceptual temperature: when the system is hot, the labels can move long distances, producing potentially large global quality improvements. When the system cools down, the move distances become smaller and hence focus on local fine-tuning of the position.

Each label has its own label descriptor. The label descriptor describes the path on which the label can move. If a label must be placed at a city in a geographic map, then its label descriptor makes sure that the label is always placed close to the graphics that represent the city. If a label must be placed at a specific point, the `IlvAnnealingPointLabelDescriptor` can be used and describes an approximately elliptical path around the point. If a label must be placed at a polyline, the `IlvAnnealingPolylineLabelDescriptor` can be used and describes a boundary path at both sides of the polyline.

Annealing Label Layout algorithm Example
In CSS
The sample below shows how to use CSS syntax to declare labels at nodes of type `IlvGeneralNode`. The specification can be loaded as a style file into an application that uses the `IlvDiagrammer` class (see *Graph layout in IBM® ILOG® JViews Diagrammer*).

```
SDM {
    LabelLayout: "true";
}

LabelLayout {
    enabled: "true";
    labelOffset: "5";
    obstacleOffset: "10";
}

node {
    class: "ilog.views.sdm.graphic.IlvGeneralNode";
    label: "@name";
    labelPosition: "Left";
    labelSpacing: "2.0";
}

node:labelLayout {
    maxDistFromPath: "5";
}
```

In Java™
The following code sample uses the `IlvAnnealingLabelLayout` class. This code sample shows how to perform an Annealing Label Layout on a manager directly.:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.labellayout.*;
import ilog.views.graphlayout.labellayout.annealing.*;
 ...
IlvManager manager = new IlvManager();
IlvManagerView view = new IlvManagerView(manager);

 ... /* Fill in here code that fills the manager with labels and obstacles */


IlvAnnealingLabelLayout layout = new IlvAnnealingLabelLayout();
layout.attach(manager);

/* Assume: label1 should be placed at the right side of rectangular node1 */
layout.setLabelDescriptor(
          label1,
          new IlvAnnealingPointLabelDescriptor(
                label1, node1, IlvAnnealingPointLabelDescriptor.RECTANGULAR,
                IlvDirection.Right));

... /* Fill in here code that sets descriptors for all other labels */

IlvLabelLayoutReport layoutReport = layout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
     System.out.println("Layout done.");
else
     System.out.println("Layout not done, code = " +
                                   layoutReport.getCode());
```

Generic features and parameters

The `IlvAnnealingLabelLayout` class supports generic parameters defined in the `IlvLabelLayout` class. The following sections describe the particular way in which these parameters are used by the subclass `IlvAnnealingLabelLayout`.

- *Allowed time*
- *Percentage of completion calculation*
- *Random generator seed value*
- *Save parameters to named properties*
- *Stop immediately*
- *Use default parameters*

Allowed time

The label layout algorithm stops if the allowed time setting has elapsed. This feature works similarly to the same feature in `IlvGraphLayout`; see *Allowed time*. If the layout stops early because the allowed time has elapsed, the result code in the layout report is:

- `IlvLabelLayoutReport.STOPPED_AND_VALID` if the labels were moved to some better (but not yet optimal) positions.
- `IlvLabelLayoutReport.STOPPED_AND_INVALID` if the time elapsed even before that.

Percentage of completion calculation

The label layout algorithm calculates the estimated percentage of completion. This value can be obtained from the label layout report during the run of the layout. (For a detailed description of this feature, see *Percentage of completion calculation* and *Layout events and listeners*.)

Random generator seed value

The Annealing Label Layout is a randomized heuristic. It uses a random number generator to control the movements. For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph. You can specify the particular value to be used as a seed value.

Example of specifying seed value
To specify the seed value `10`:

In CSS
Add to the `LabelLayout` section:

```
useSeedValueForRandomGenerator: "true";
seedValueForRandomGenerator: "10";
```

In Java™

Call:

```
layout.setUseSeedValueForRandomGenerator(true);
layout.setSeedValueForRandomGenerator(10);
```

Save parameters to named properties

The label layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Using named properties to save layout parameters*.)

Stop immediately

The label layout algorithm stops after cleanup if the method `stopImmediately` is called. This method works for the `IlvLabelLayout` class similarly to the corresponding method in the `IlvGraphLayout` class. For a description of this method in the `IlvGraphLayout` class, see *Stop immediately*. If the layout stops early in this way, the result code in the layout report is:

♦ `IlvLabelLayoutReport.STOPPED_AND_VALID` if the labels were moved to some better (but not yet optimal) positions.

♦ `IlvLabelLayoutReport.STOPPED_AND_INVALID` if the layout stopped even before that.

Use default parameters

After modifying any label layout parameter, you may want the layout algorithm to use the default values. You select the default values for all global parameters by:

```
layout.setUseDefaultParameters(true);
```

IBM® ILOG® JViews Diagrammer keeps the previous settings when selecting the default values mode. You can switch back to your own settings by:

```
layout.setUseDefaultParameters(false);
```

This setting affects only the global layout parameters. The label descriptors have no default values, so parameters of the label descriptors do not change depending on this flag.

Label descriptors

To define where a label must be placed, you must specify a label descriptor for each label. The algorithm places only those labels that have a label descriptor.

A label descriptor describes the locations that are allowed for the label. For instance, if you place a city name label in a geographic map, you want the label to be positioned close to the graphic objects that represent the city. Positions far away from the city are not reasonable for the label.

Example of label descriptors
To specify these parameters:

In CSS
Label descriptors are automatically created for all labels. You only need to specify the details of the label descriptors by using the pseudo classes `labelLayout` and `labelLayout_`*i* where *i* is the child index of the label in `IlvSDMCompositeNode` or `IlvSDMCompositeNode`. For instance, the descriptor of a label at an `IlvGeneralNode` object is specified in the following way:

```
node:labelLayout {
    ... any Bean property of the point label descriptor ...
}
```

The descriptor of a label which is the 3rd child of an `IlvSDMCompositeLink` is specified in the following way:

```
link:labelLayout_3 {
    ... any Bean property of the polyline label descriptor ...
}
```

In Java™
You need to allocate a new label descriptor for each label. There are two ways to specify the label descriptors:

♦ To set the descriptor for one label, call:

  ```
  layout.setLabelDescriptor(label, descriptor);
  ```

  You can retrieve the current label descriptor using the method:

  ```
  layout.getLabelDescriptor(label);
  ```

♦ You can instead specify a label descriptor provider. `IlvAnnealingLabelDescriptorProvider` is an interface that delivers label descriptors for labels on the fly. The provider has the advantage that you don't need to run in advance through all labels to set the label descriptor. Instead, the layout detects the labels and asks the provider to deliver the label descriptor if none was set explicitly. The provider can then allocate the label descriptor on the fly, or can deliver a preallocated descriptor that was stored somewhere else outside layout. Using the label descriptor provider is in particular useful if the number of labels and obstacles change frequently, because you don't need to keep track of the labels that have already a label descriptor, and if you want to have a central place that manages all label descriptors.

The reference manual of the class `IlvAnnealingLabelDescriptorProvider` contains further information on how to implement this interface. In order to set the provider, call the following method:

```
layout.setLabelDescriptorProvider(provider);
```

Subclasses of label descriptors

There are two predefined subclasses of label descriptors:

♦ *Point label descriptor*

♦ *Polyline label descriptor*

In a diagram component with CSS styling, the point label descriptor is used for all labels at nodes, and the polyline label descriptor is used for all labels at links.

If you program the label layout directly in Java code, depending on the parameters passed during the construction, these subclasses allow you to place a label:

♦ Close to a given point.

♦ Close to a specific rectangular or elliptic obstacle (such as a node).

♦ Along an imaginary polyline.

♦ Close to a polyline obstacle (for example, `IlvLine`, `IlvPolyline`).

♦ Close to a link.

You can also implement your own label descriptors by subclassing `IlvAnnealingLabelDescriptor`. This is explained in the section *For experts: implementing your own label descriptors*.

Point label descriptor

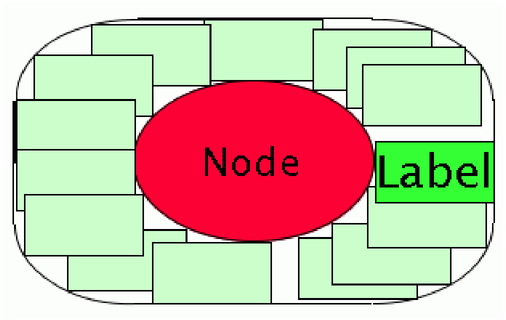
The `IlvAnnealingPointLabelDescriptor` can be used to place a label at a specific obstacle or point. This is known as the *point labeling problem*.

Positioning at an obstacle

The following example shows how to position a label at a specific obstacle in Java™ .

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPointLabelDescriptor(label, node,
          IlvAnnealingPointLabelDescriptor.ELLIPTIC, IlvDirection.Right));
```

This specification can be used if the label must be placed at a node that has an elliptical or circular shape. The label is placed in the free area around the node so that the border of the label just touches the border of the node (see the following figure). The preferred position is the right side of the node, but this preferred position is used only if it does not create overlaps. If the node is moved or reshaped, the next call of label layout will update the position of the label automatically so that it follows the node.



Potential label positions around a node

The example uses the following constructor:

```
IlvAnnealingPointLabelDescriptor(Object label,
                                 Object relatedObstacle,
                                 int shape,
                                 int preferredDirection)
```

This constructor takes the following parameters:

- The `relatedObstacle` parameter is the obstacle that gets the label. The label is placed outside but close to this obstacle. The related obstacle can be a node of a graph, a city of a geographic map, a station of a railroad, and so on, whatever needs to have a label. The shape of the related obstacle should be either an ellipse, a circle, or a rectangle.
- The `shape` argument can take the following values:
  - `IlvAnnealingPointLabelDescriptor.ELLIPTIC` for ellipses or circles,

- `IlvAnnealingPointLabelDescriptor.RECTANGULAR` for rectangles.

  If the real shape of the related obstacle is neither of these, pass the shape that is the best approximation. For instance, if the obstacle is an `IlvRoundRectangle`, it can be considered as a rectangular shape and the `RECTANGULAR` option is then the best approximation.

♦ The `preferredDirection` parameter is a suggestion of where the label layout algorithm should preferably place the label. If the area at the preferred position is occupied, the label will be placed elsewhere. Options for the preferred position are:

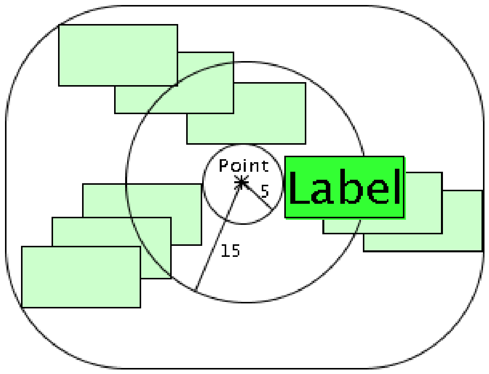
- `IlvDirection.Left`
- `IlvDirection.Right`
- `IlvDirection.Top`
- `IlvDirection.Bottom`

Positioning at a point

The following example shows how to position a label at a specific point in Java:

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPointLabelDescriptor(label, null, new IlvPoint(25, 75),
          5f, 15f, IlvDirection.Right));
```

Use this specification if the label must be placed close to specific coordinates, like (in this example 25, 75) regardless of any obstacle. The label must be at least 5 coordinate units and at most 15 coordinate units away from the point (see the following figure). The preferred position is at the right side of the point.



Potential label positions between 5 and 15 units away from a point

The example uses the following constructor:

```
IlvAnnealingPointLabelDescriptor(Object label,
                                  Object relatedObstacle,
                                  IlvPoint referencePoint,
                                  float minDist,
                                  float maxDist,
                                  int preferredDirection)
```

This `IlvAnnealingPointLabelDescriptor(java.lang.Object, java.lang.Object, ilog.views.IlvPoint, float, float, int)` constructor takes the following parameters:

♦ relatedObstacle and referencePoint: The label is placed close to the reference point. It does not take the actual position of the related obstacle into account. If the related obstacle is moved, the label does not follow the obstacle on the next call of layout, but stays at the reference point.

  If a related obstacle is given, the label is not pushed away from the related obstacle. Rather, it is pushed away from all other obstacles to avoid overlaps. You can set the `relatedObstacle` parameter to `null` if the label is independent of all obstacles.

♦ The parameters `minDist` and `maxDist` are the minimal and maximal distances from the reference point, measured from the border of the label. If you set the minimal and maximal distance to 0, the label will just touch the reference point. To keep the circular area around the reference point free, set the minimal distance accordingly. Most of the time you probably want to keep the label close to the reference point; hence, set the minimal and maximal distances to the same value.

♦ The preferredDirection parameter indicates whether the label should be placed to the left, right, top, or bottom of the reference point. This is a suggestion for the labeling algorithm, as described for *Positioning at an obstacle*.

Positioning on multiple criteria

The most powerful constructor combines all the possibilities described in *Positioning at an obstacle* and *Positioning at a point*:

```
IlvAnnealingPointLabelDescriptor(Object label,
                                  Object relatedObstacle,
                                  IlvPoint referencePoint,
                                  int shape,
                                  float halfWidth,
                                  float halfHeight,
                                  float maxDistFromPath,
                                  float preferredDistFromPath,
                                  int preferredDirection)
```

This `IlvAnnealingPointLabelDescriptor(java.lang.Object, java.lang.Object, ilog.views.IlvPoint, int, float, float, float, float, int)` constructor takes the following parameters:

♦ relatedObstacle and referencePoint: If a related obstacle is given and the reference point is `null`, the label is placed close to the related obstacle. If a reference point is not null, the label is placed close to the reference point independently of the related obstacle position.

♦ `shape` : the shape of the free area around the point can be rectangular or elliptic.

- ♦ `halfWidth` and `halfHeight`: The parameter `halfWidth` is the minimal distance of the label to the reference point in the horizontal direction. The parameter `halfHeight` is the minimal distance of the label to the reference point in the vertical direction. If the reference point is `null`, the parameters `halfWidth` and `halfHeight` are calculated from the bounding box of the related obstacle.
- ♦ The parameter `maxDistFromPath` specifies the maximal additional distance allowed for the label (shown in *Potential Label Positions With Rectangular Shape at a Point*).
- ♦ The parameter `preferredDistFromPath` specifies the preferred additional distance for the label. Its value should be between 0 and `maxDistFromPath`.
- ♦ The preferredDirection parameter indicates whether the label should be placed to the left, right, top, or bottom of the reference point or related obstacle. This is a suggestion for the labeling algorithm, as described in *Positioning at an obstacle*.

Starting from an empty descriptor (point)

An alternative way to create a point label descriptor is to start from the empty descriptor:

```
descriptor = new IlvAnnealingPointLabelDescriptor();
```

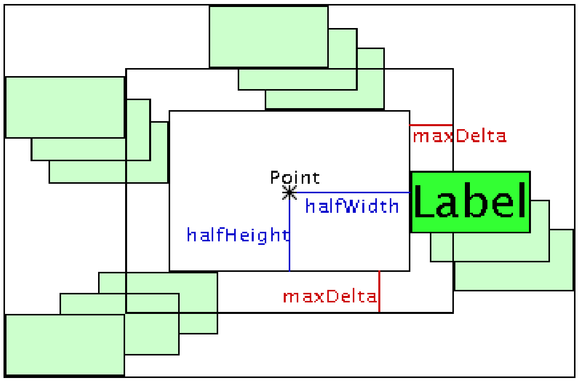
Before using the empty descriptor, you must fill it with information on how the label should be placed. As a minimum, you need to specify a related obstacle or a reference point. For example:

```
descriptor.setRelatedObstacle(obstacle);
descriptor.setShape(IlvAnnealingPointLabelDescriptor.ELLIPTIC);
descriptor.setPreferredDirection(IlvDirection.Left);
```

At the end of all changes, the descriptor must be passed to the layout instance:

```
layout.setLabelDescriptor(label, descriptor);
```

Once the descriptor is passed to the layout instance, it should normally not be changed. If you need to change it later, you must pass it to the layout instance again.



Potential Label Positions With Rectangular Shape at a Point

Example of specifying point label descriptor

Example of an `IlvGeneralNode` specification with label:

In CSS

You can specify all multiple criteria of the point labeling descriptor except the related obstacle. In CSS, the related obstacle is always the node that owns the label. The reference point and the shape (`ELLIPTIC` or `RECTANGULAR`) are automatically derived from the node that owns the label, but can be overridden in the CSS specification.

Example of an `IlvGeneralNode` specification with label:

```
node {
    class: "ilog.views.sdm.graphic.IlvGeneralNode";
     label: "@name";
    shapeType: "Rectangle"; // shape of path for label descriptor
    shapeWidth: "30";       // twice the halfWidth
    shapeHeight: "30";      // twice the HalfHeight
    labelPosition: "Left";  // preferredDirection
    labelSpacing: "2.0";    // preferredDistFromPath
}
```

In this example, `shapeType`, `shapeWidth` and `shapeHeight` specify the basic shape of the node and are taken into account by the label layout, the `labelPosition` specifies the preferred direction of the label for the label layout, and the `labelSpacing` specifies the preferred distance of the label from the path around the border of the basic shape of the node.

These parameters can be overridden by specifying the details of the point label descriptor in the following way:

```
node:labelLayout {
    shape: "ELLIPTIC";
    halfWidth: "16";
    halfHeight: "16";
    maxDistFromPath: "5";
    preferredDistFromPath: "3";
    preferredDirection: "Top";
}
```

If the node is an instance of `IlvSDMCompositeNode` and its fourth child is a label, the details of the label descriptor for that label can be specified in the following way:

```

node:labelLayout_4 {
    shape: "ELLIPTIC";
    halfWidth: "16";
    halfHeight: "16";
    maxDistFromPath: "5";
    preferredDistFromPath: "3";
    preferredDirection: "Top";
}
```

The meaning of these parameters is explained in *Positioning on multiple criteria*.

Polyline label descriptor

If you want to place labels at straight lines, polylines, or links, you should use the class `IlvAnnealingPolylineLabelDescriptor`. The allowed area for labels at a polyline is different from the rectangular or elliptic area considered for placing labels at a reference point (see *Positioning at a point*). A polyline has two sides where the label can be placed along a path. This is known as the *polyline labeling problem*.

This section first explains how to code a polyline label descriptor in Java™ , then gives a CSS sample.

Note: The polyline label descriptor is not suitable for placing labels at splines or spline links. Because splines have a complex geometric shape, the automatic placement of labels at splines is currently not supported.

Simple positioning at a link

This specification can be used if the label must be placed at a straight or polyline link. Here is an example:

```
 layout.setLabelDescriptor(label,new IlvAnnealingPolylineLabelDescriptor(label,

link,IlvAnnealingPolylineLabelDescriptor.CENTER));
```

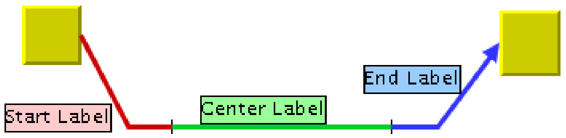
The label is placed near the center of the link such that one border of the label just touches the link. If the link is moved or reshaped, the next label layout call will update the position of the label automatically to follow the link.

The example uses the following `IlvAnnealingPolylineLabelDescriptor(java.lang.Object, ilog.views.IlvLinkImage, int)` constructor:

```
IlvAnnealingPolylineLabelDescriptor (Object label,
                                     IlvLinkImage link,
                                     int anchor)
```

The options for the `anchor` parameter are:

- `IlvAnnealingPolylineLabelDescriptor.CENTER`: places the label near the center of the link (that is, in the middle third of the link path).
- `IlvAnnealingPolylineLabelDescriptor.START`: places the label near the source node of the link (that is, in the first third of the link path).
- `IlvAnnealingPolylineLabelDescriptor.END`: places the label near the target node of the link (that is, in the last third of the link path).
- `IlvAnnealingPolylineLabelDescriptor.FREE`: places the label anywhere on the link.



Anchors for Label Positions at a Link

Simple positioning at a polyline obstacle

Use this specification if the label must be placed at a polyline obstacle (an instance of `IlvLine` or `IlvPolyline` in the default labeling model). The polyline obstacle does not need to be a link of a grapher.

Here is an example:

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPolylineLabelDescriptor(label, polyline,
          IlvAnnealingPolylineLabelDescriptor.FREE,
          IlvDirection.Left, IlvDirection.TopLeft,
          IlvAnnealingPolylineLabelDescriptor.GLOBAL));
```

The label is placed anywhere at the left or top side of the polyline obstacle, with preference given to the left side.

The example uses the following constructor:

```
IlvAnnealingPolylineLabelDescriptor
                                 (Object label,
                                  Object relatedPolylineObstacle,
                                  int anchor,
                                  int preferredSide,
                                  int allowedSide,
                                  int sideAssociation)
```

Even though a polyline does not have a source node or a target node, the `anchor` parameter can be used in the same way as for links (`CENTER`, `START`, `END`, and `FREE`). The first control point of the polyline is the start point. Labels with anchor `START` are placed closer to the first control point, and labels with anchor `END` are placed closer to the last control point of the polyline.

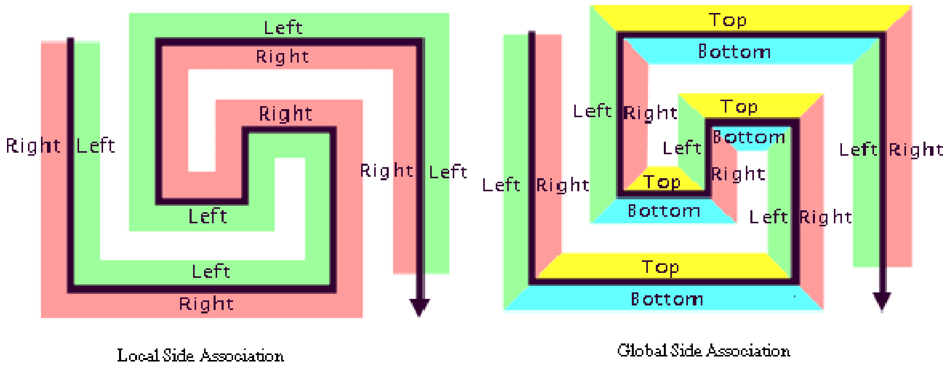
The value of the `preferredSide` parameter is a suggestion of where the label layout algorithm should preferably place the label. If the area at the preferred side is occupied, the label is placed elsewhere.

In contrast, the `allowedSide` parameter is a strict constraint: it is always obeyed, even if the entire area at the allowed side is occupied and the label must overlap the obstacles in that area.

Side association

The orientation of the preferred and allowed sides depend on the sideAssociation parameter. This parameter can take the following values (see the following figure):

- `IlvAnnealingPolylineLabelDescriptor.LOCAL`
- `IlvAnnealingPolylineLabelDescriptor.GLOBAL`



Side Associations

Local side association

If the side association is local, each polyline has two sides: left and right. The sides can be determined from the flow direction of the polyline from start point to end point. Consider yourself standing on the polyline looking in the direction where the polyline continues towards the end point, and then determine which is the left and which is the right side. Hence, the meaning of left and right in local side association is relative to the polyline. The options for the preferredSide and allowedSide parameters are in this case:

- `IlvDirection.Left`
- `IlvDirection.Right`

You can also specify the value 0 for the allowed side, which indicates that you do not want to restrict the side: both sides are allowed.

Global side association

If the side association is global, the side specification is independent of the flow direction of the polyline and more like a compass direction: north is top, south is bottom, west is left, and east is right. Here more options are possible: in addition to the basic top, bottom, left, right, all meaningful combinations of these are allowed. You specify the sides in the following way:

- `IlvDirection.Left`
- `IlvDirection.Right`
- `IlvDirection.Top`

- `IlvDirection.Bottom`

You can also use combinations of these, such as:

- `IlvDirection.Left | IlvDirection.Right` (left or right but not top or bottom).
- `IlvDirection.Left | IlvDirection.Top` (which is the same as `IlvDirection.TopLeft`, meaning the left or the top side).

You can specify the value 0 for the allowed side if all sides should be allowed.

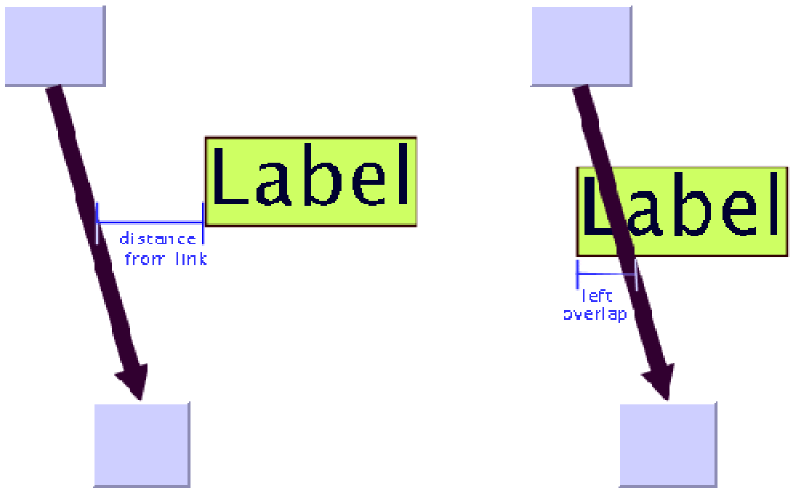
Full positioning at a link

The most powerful constructor of a descriptor for a label that should be placed at a link is the following `IlvAnnealingPolylineLabelDescriptor` :

```
IlvAnnealingPolylineLabelDescriptor
                                  (Object label,
                                   Object link,
                                   Object source,
                                   Object target,
                                   int anchor,
                                   float maxDistFromLink,
                                   float preferredDistFromLink,
                                   int preferredSide,
                                   int allowedSide,
                                   int sideAssociation,
                                   float topOverlap,
                                   float bottomOverlap,
                                   float leftOverlap,
                                   float rightOverlap)
```

It combines all the possibilities described in *Simple positioning at a link*: in addition to the anchor, the side association, and the allowed and preferred sides, you can specify overlap options.

You must pass the link as well as the source node and the target node of the link. The label does not need to touch the link. If you want to allow the label to be placed at a short distance from the link, then specify the maximal distance by `maxDistFromLink` and the preferred distance by `preferredDistFromLink`. Conversely, you may want to allow the link to partially overlap the label. You specify this by setting `topOverlap`, `bottomOverlap`, `leftOverlap`, or `rightOverlap` to a value larger than 0. This is illustrated in the following figure.



Distance and Overlap at Link

Full positioning at a polyline obstacle

If the label should be placed at a polyline that is not a link, then the most powerful `IlvAnnealingPolylineLabelDescriptor` constructor is the following:

```
IlvAnnealingPolylineLabelDescriptor
                                (Object label,
                                 Object relatedObstacle,
                                 IlvPoint[] referencePoints,
                                 float lineWidth,
                                 float minPercentageFromStart,
                                 float maxPercentageFromStart,
                                 float prefPercentageFromStart,
                                 float maxDistFromPath,
                                 float preferredDistFromPath,
                                 int preferredSide,
                                 int allowedSide,
                                 int sideAssociation,
                                 float topOverlap,
                                 float bottomOverlap,
                                 float leftOverlap,
                                 float rightOverlap)
```

It combines all previously mentioned possibilities. If the label should be placed at a polyline obstacle, then pass this object as the related obstacle. If the label should be placed at an imaginary polyline, then pass the polyline with the `points` parameter and the width of the polyline with the `lineWidth` parameter. Instead of an anchor, you can pass the area where the label is placed by the minimal, maximal, and preferred percentage values relative to the polyline length. The minimal and maximal percentages are strictly obeyed, while the preferred percentage is only a recommendation for the layout.

- For instance, if you want to specify that the label can be placed anywhere but you prefer the center of the polyline, specify 0 and 100 for the minimal and maximal percentages and 50 for the preferred percentage. If there is not enough free space at the center, the label will be placed elsewhere.
- But if you want to specify that the label must be placed close to the center even if there is not enough space, then specify, for instance, 40 and 60 for the minimal and maximal percentages instead.

Starting from an empty descriptor (polyline)

An alternative way to create a polyline label descriptor is to start with the empty descriptor:

```
descriptor = new IlvAnnealingPolylineLabelDescriptor();
```

Before using the empty descriptor, you must fill it with information on how the label should be placed. As a minimum, you need to specify a related obstacle or reference points and line width. For instance:

```
descriptor.setRelatedObstacle(polyline);
descriptor.setMinPercentageFromStart(10f);
descriptor.setMaxPercentageFromStart(90f);
descriptor.setPreferredPercentageFromStart(50f);
```

At the end of all changes, the descriptor must be passed to the layout instance:

```
layout.setLabelDescriptor(label, descriptor);
```

Once the descriptor is passed to the layout instance, it should normally not be changed. If you need to change it later, you must pass it to the layout instance again.

Example of specifying polyline label descriptor

Example of an `IlvGeneralLink` instance with a label:

In CSS

You can specify all multiple criteria of the polyline labeling descriptor except the related obstacles. In CSS, the related obstacle is always the link that owns the label. The related source and target object of the link label descriptor are calculated from the source and target of the link. The reference points and line width are automatically derived from the link that owns the label, but can be overridden in the CSS specification.

```
link {
    class : "ilog.views.sdm.graphic.IlvGeneralLink";
    lineWidth : "2";
    label : "@name";
    decorationPositions[0]: "0.8";  // prefPercentageFromStart
}
```

In this example, the `lineWidth` value of the link automatically specifies the line width for the label descriptor, and the `decorationPositions[0]` affects where the label is anchored at the link. These parameters can be overridden by specifying the details of the polyline label descriptor in the following way:

```
link:labelLayout {
    lineWidth: "3";
    minPercentageFromStart: "30";
    maxPercentageFromStart: "70";
    preferredPercentageFromStart: "50";
    maxDistFromPath: "4";
    preferredDistFromPath: "2";
    preferredSide: "Left";
    allowedSide: "Left";
    sideAssociation: "GLOBAL";
    topOverlap: "1";
    bottomOverlap: "1";
    leftOverlap: "1";
    rightOverlap: "1";
}
```

If the link is an instance of `IlvSDMCompositeLink` and its fourth child is a label, the details of the descriptor for this label can be specified in the following way:

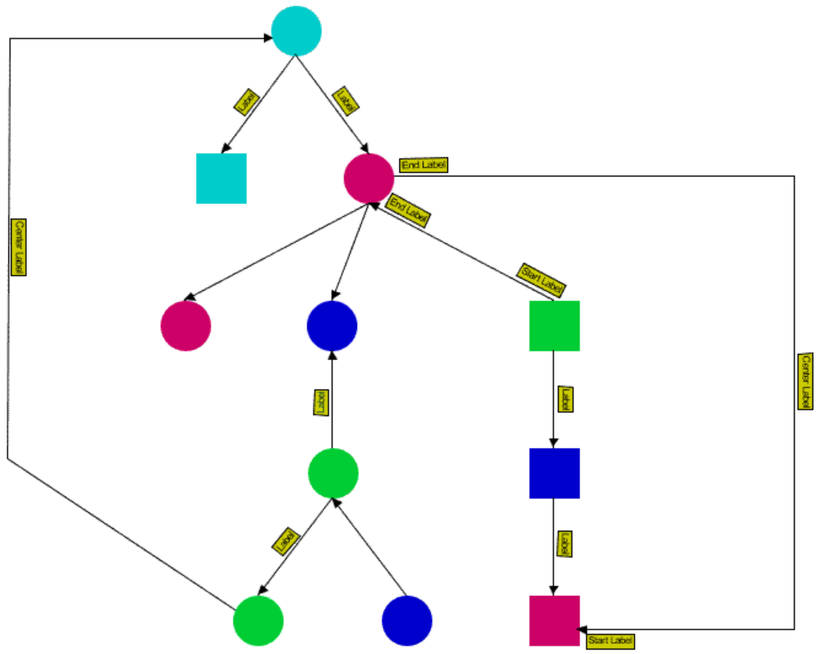
```
link:labelLayout_4 {
    lineWidth: "3";
    minPercentageFromStart: "30";
    maxPercentageFromStart: "70";
    preferredPercentageFromStart: "50";
    maxDistFromPath: "4";
    preferredDistFromPath: "2";
    preferredSide: "Left";
    allowedSide: "Left";
    sideAssociation: "GLOBAL"”;
    topOverlap: "1";
    bottomOverlap: "1";
    leftOverlap: "1";
    rightOverlap: "1";
}
```

The meaning of these parameters is already explained in *Full positioning at a link* and *Full positioning at a polyline obstacle*.

Rotated labels

Both the point label descriptor and the polyline label descriptor can be used if the rectangular label has a rotation that depends on its position.

These descriptors work only if the labeling model attached to the Annealing Label Layout implements the interface `IlvLabelingModelWithRotation`. (The labeling model `IlvDefaultLabelingModel` implements this interface.)



Rectangular label rotation dependent on position

In Java™ code, the label descriptor must describe how the rotation of a label depends on its position. For example, a label associated with a line can always have the rotation of the line. You must create a label descriptor whose method `getRotation` returns the corresponding rotation. See *Label descriptor that returns the corresponding rotation*.

Label descriptor that returns the corresponding rotation

```
layout.setLabelDescriptor(label,
    new IlvAnnealingPolylineLabelDescriptor(label, line,
            IlvAnnealingPolylineLabelDescriptor.FREE,
            IlvDirection.Left, IlvDirection.TopLeft,
            IlvAnnealingPolylineLabelDescriptor.GLOBAL) {

        public double getRotation(IlvLabelingModel model,
                IlvRect labelRect) {
            IlvLine line = (IlvLine)getRelatedObstacle();
            IlvPoint p1 = line.getFrom();
            IlvPoint p2 = line.getTo();
            double angle = Math.atan2(p2.y-p1.y, p2.x-p1.x);
            return Math.toDegrees(angle);
        }
    });
```

If the method `getRotation` is not overridden, it will query the labeling model for the rotation. Therefore, as an alternative to overriding the method `getRotation` at the label descriptor, it is also possible to override the method `getRotation(java.lang.Object, ilog.views.IlvRect)` of the default labeling model (more specifically of the `IlvLabelingModelWithRotation` interface).

In CSS, it is not necessary to specify the rotation of the label. Rotated labels are only supported for `IlvGeneralLink`. CSS automatically uses polyline label descriptors that return the appropriate rotation for labels of `IlvGeneralLink`.

Specific global parameters

The following global parameters are specific to the `IlvAnnealingLabelLayout` class:

- *Label offset*
- *Obstacle offset*
- *Label movement policy*
- *Automatic update*
- *Expert parameters*

Label offset

The label offset controls the desired minimal distance between two neighbored labels (see *Label and Obstacle Offsets*, left). To avoid labels being placed too close to each other, you can increase the label offset. This conceptually pushes the labels farther apart. However, depending on the available space, the minimal distance between labels cannot always be maintained.

Example of specifying label offset
To set the label offset:

In CSS
Add to the `LabelLayout` section:

```
labelOffset: "25";
```

In Java™
Call:

```
layout.setLabelOffset(25f);
```

Obstacle offset

The obstacle offset controls the desired minimal distance between a label and an unrelated obstacle. The obstacle offset is usually more important than the label offset because, if the obstacle offset is too small, the label may be placed so close to an unrelated obstacle that it incorrectly appears to be assigned to that obstacle (see *Label and Obstacle Offsets*, right: does, for example, the green label belong to the upper yellow node or to the green node?). Increasing the obstacle offset conceptually pushes the label away from the obstacle. However, depending on the available space, the minimal distance between label and obstacle cannot always be maintained.

The obstacle offset should not be set to an unreasonably large value (such as `Float.MAX_VALUE`) because this can cause computational problems inside quadtree operations.

Example of specifying node placement iterations and allowed time (GL algorithm)
To set the obstacle offset:

In CSS
Add to the `LabelLayout` section:

```
obstacleOffset: "25";
```

In Java
Call:

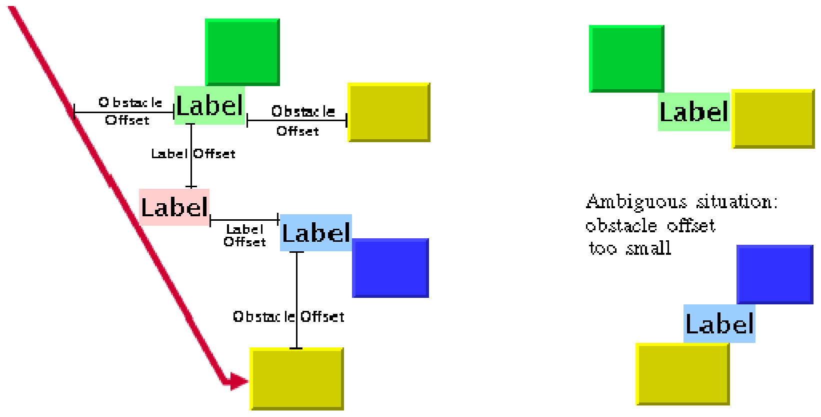
```
layout.setObstacleOffset(25f);
```

The specified obstacle offset works globally for all labels.

In Java, it is also possible to specify a smaller obstacle offset for specific label/obstacle pairs. You need to install an obstacle offset interface that returns the obstacle offset for a given pair:

```
layout.setObstacleOffsetInterface(new IlvObstacleOffsetInterface() {
    public float getOffset(IlvLabelingModel m, Object label, Object obstacle)

    {
        if (label instanceof IlvZoomableLabel &&
            obstacle instanceof IlvIcon)
            return 3f;
        else
            // use the global obstacle offset
            return Float.MAX_VALUE;
    }
});
```

The effective offset is the lower of the values returned by the obstacle offset interface and the globally specified offset respectively. Hence, the obstacle offset interface in the previous example means that `IlvZoomableLabel` labels can be placed up to 3 units near `IlvIcon` obstacles, while they are placed away from all other obstacles by at least the amount specified by the call`layout.setObstacleOffset`.



Label and Obstacle Offsets

Label movement policy

The label movement policy is an easy way to define which labels should be moved by the label layout algorithm.

Example of specifying label movement policy
In CSS
It is not possible to specify the label movement policy in CSS, but you can integrate a label movement policy into the label layout renderer by subclassing the renderer. An example showing how to subclass renderers is provided in *Writing a new layout renderer to clip links*.

In Java
The following code installs a label movement policy such that the layout moves only labels with a height greater than 100:

```
layout.setLabelMovementPolicy(new IlvLabelMovementPolicy() {
    public boolean allowMove(IlvLabelingModel labelingModel, Object label)
    {
        return (labelingModel.boundingBox(label).height > 100);
    }
});
```

Labels with smaller heights are not moved. However, they are also not completely ignored, because the layout tries to position the movable labels so that they do not overlap the immovable labels, and the label offset is respected between movable and immovable labels.

A more general useful example is a movement policy that prohibits moving labels that initially do not overlap anything. This predefined movement policy is available through the class `IlvOverlappingLabelMovementPolicy`. You can use this class in applications that have their own label positioning mechanism and use the Annealing Label layout only as a postprocessing step to improve the positions of overlapping labels. To install this policy, call:

```
layout.setLabelMovementPolicy(new IlvOverlappingLabelMovementPolicy());
```

Automatic update

After layout, the labels are placed close to the related obstacle according to the label descriptor. For instance, a link label is placed close to its link. However, if you move the link interactively, the label normally stays at the old position, which may be far away from the link after the movement. The label loses the connection to the link, and a new layout is necessary.

Because it is too time-consuming to redo the layout after each single interactive move, the Annealing Layout algorithm has a feature that automatically updates the labels on geometric changes, that is, the label follows the link when the link moves.

Example of specifying automatic update
To enable this feature:

In CSS
Add to the LabelLayout section:

```
autoUpdate: "true";
```

In Java
Call:

```
layout.setAutoUpdate(true);
```

If automatic update is enabled, the algorithm does not perform a full layout of all labels during each interactive change. It repositions only the label whose related obstacle has moved in one step. Thus it may produce more overlaps than a full layout. The automatic update mechanism is much faster, however, and hence better suitable for interactions.

Note: The automatic update feature works only if the labeling model provides `LabelingModelEvent` objects on each obstacle movement. The `IlvDefaultLabelingModel` provides these events. If you implement your own labeling model, you must provide these events in order to use the automatic update feature.

Expert parameters

A few parameters are available for an advanced use of the Annealing Label Layout.

Quadtree

To speed up the layout, the Annealing Label Layout algorithm uses a quadtree data structure. The quadtree enables a very efficient check for overlaps. The layout algorithm automatically detects from the graph model whether the quadtree can be used. You can switch it off explicitly by calling:

```
layout.setUseQuadtree(false);
```

Normally, it is not useful to switch the quadtree off because it slows down label positioning. However, if you implement your own labeling model, you may want to use this flag to verify that the labeling model is correct.

Simulated annealing

Simulated annealing is an iterative mechanism. In each iteration step, all labels are tested for better positions. Usually, the algorithm is capable of detecting automatically when to stop. The algorithm stops if:

♦ The maximal number of iterations is reached.

♦ After several iterations, no better position was found for any label.

♦ After several iterations, the quality did not improve by a given percentage.

In a few cases, it may be necessary to limit the number of iterations, which can be done by calling:

```
layout.setAllowedNumberOfIterations(100);
```

As a general hint, to obtain a reasonable layout, the allowed number of iterations should not be considerably lower than the number of labels.

Simulated Annealing stops if, after several iterations, no better position was found for any label. Because the search is randomized, this does not necessarily mean that the best position was already found; however, it indicates that finding the best position would require too much layout time. The number of ineffective iterations before stopping can be changed by calling:

```
layout.setMaxNumberOfFailIterations(maxNumber);
```

The default value is 20. If you set it to a higher value, the layout slows down but may find better positions for the labels. If you set it to a lower value, the layout stops sooner, but the label positions may be far from optimal.

In some cases, the algorithm improves the quality in each step, but the amount of improvement gets smaller in each step. In this situation, the previous fail-iteration criteria does not work well because there is an improvement in each step, but the amount of the improvement is so negligibly small that we want to stop. Therefore, it is also possible to require that the quality must improve in each step by a minimum percentage.

For example, to specify that the algorithm must improve over ten rounds by at least 2%, call:

```
layout.setNumberIterationsForMinImprovement(10);
layout.setMinImprovementPercentageToContinue(2);
```

By default, the layout stops if the quality did not improve by 0.5% over five iterations. If you set the required improvement percentage higher or the number of iterations lower, the layout stops sooner, but the label positions may be far from optimal. If you set the required percentage to 0%, this stop criterion is disabled and will no longer have any effect.

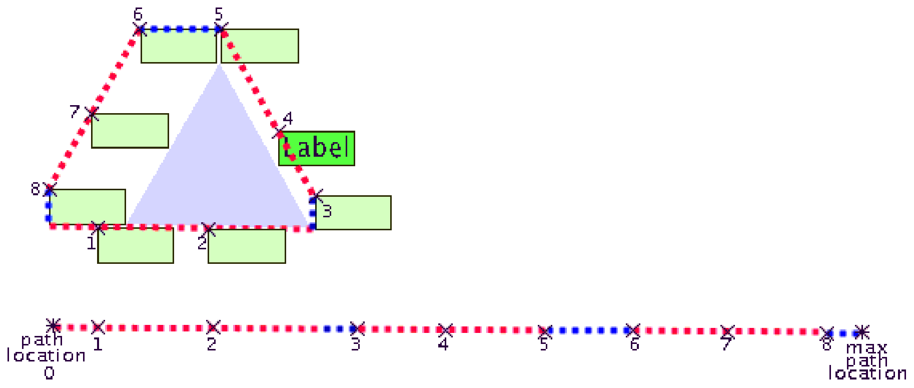
For experts: implementing your own label descriptors

The Annealing Label Layout is extensible. The point label descriptor and the polyline label descriptor are designed to cover the majority of the cases. In rare situations, you may want to implement your own label descriptor by subclassing the base class `IlvAnnealingLabelDescriptor`. This section describes the necessary steps.

A label descriptor basically specifies the path where the top-left corner of a label can be placed. For simplification, it considers the path rolled out such that the path has only one dimension. If the path is known, the precise label position can be specified by just one value: the path location. The Annealing Label Layout proposes different path locations during the layout; however, it does not know what the path looks like. The task of the label descriptor is to translate the path location into concrete (x, y) coordinates of the label.

As an example, we want to create a label descriptor that can place labels precisely at a triangular obstacle. We could use the point label descriptor as an approximation, but it does not place the labels precisely at a triangular shape.

In the following figure, the upper diagram shows the path around the triangle (the dashed red and blue line). Below, you can see the same path rolled out in one dimension. The Annealing Label Layout may ask the label descriptor to place the label at position 1 to 8. For the Annealing Label Layout, these positions are just numbers between 0 and `maxPathLocation`. The task of the label descriptor is to translate these numbers into the correct positions as shown in the upper part of the figure.



Path locations at a triangle label descriptor

The base class `IlvAnnealingLabelDescriptor`. has two protected data members:

♦ `actPathLocation` is the current path location of the label.

♦ `maxPathLocation` is the maximal value of the path location.

To create a new label descriptor, you need to implement a method that initializes the path constructor at the beginning of layout. You should calculate the maximal path location `maxPathLocation` and initialize the `actPathLocation` here. The method is called only once during layout:

```
void initialize(IlvLabelingModel labelingModel)
```

In the previous figure, the maximal path location for an equilateral triangle is:

```
3 * sidelength + 2 * labelwidth + 2  * labelheight
```

At each iteration step, the layout calls the method `setPosition` and provides an actual value for the path location. The method `setPosition` should store the value into `actPathLocation` and translate the path location into appropriate (x, y) coordinates. Then it should call the predefined method `updatePosition(x, y)` with these coordinates:

```
public void setPosition(double pathLocation, float distFromPath)
{
    float x, y;
    // make sure the position is between 0 and max
    while (pathLocation > maxPathLocation)
        pathLocation -= maxPathLocation;
    while (pathLocation < 0)
        pathLocation += maxPathLocation;
    // store the actual position
    actPathLocation = pathLocation;
    // translate the path location into (x, y)
    if (pathLocation < labelwidth + sidelength) {
        x = (float)pathLocation;
        y = triangleBottom;
    } else if (pathLocation < labelwidth + labelheight + sidelength) {
        x = labelwidth + sidelength;
        y = triangleBottom - (float)pathLocation + labelwidth + sidelength;
    } else if ... (other cases) ...
        ...
    // finally, update the internal data structures
    updatePosition(x, y);
}
```

The label may have a preferred position at the triangle. The Annealing Layout checks a location close to the preferred position from time to time. You should implement the following method to return the preferred path location:

```
double getPreferredPathLocation()
```

Furthermore, you should implement a strategy on how to come close to the preferred location. Towards the end of layout, the algorithm calls the method:

```
setTowardsPreferredPosition(pathLocation, dist, i, maxI)
```

to perform a sequence of steps that shift the label from the current position closer to the preferred position.

with `i` from `1` to `maxI`. Implement the method so that at each step you calculate a path location closer to the preferred location. When `i` is `maxI`, it should be exactly at the preferred location. You can call setPosition to move the label to the preferred (x, y) position. For instance:

```
public void setTowardsPreferredPosition(
                double pathLocation, float dist, int i, int maxI)
{
    double offset = pathLocation - getPreferredPathLocation();
    double newLocation = pathLocation - i * offset / maxI;
    setPosition(newLocation, dist);
}
```

These methods take the distance parameter in addition to the path location. This is the distance from the path. If the label must always be on the path, you can assume this distance is 0. Set it to a different value only if your label descriptor allows the label to have a variable offset from the path.

Random Label Layout

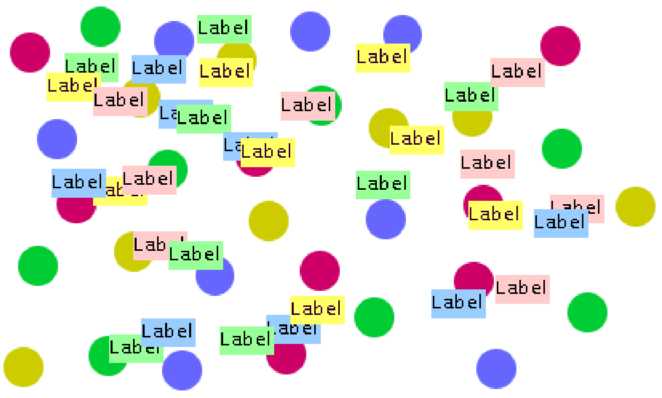
Describes the *Random Label Layout* algorithm (class `IlvRandomLabelLayout` from the package `ilog.views.graphlayout.labellayout.random`).

In this section

Sample

The Random Label layout can only be used in Java™ . It is not available in the label layout renderer of a diagram component, hence there is no way to specify this layout in CSS. It exists only for demonstration purposes.

The following sample drawing was produced with the Random Label Layout.



Label placement produced with the Random Label Layout

Although each label belongs to a circle of the same color, the random label placement does not show this. Instead it places the labels randomly, creating many overlaps. The Random Label Layout can be used to shuffle the labels arbitrarily in a given area.

Features

- Mainly for demonstration purposes.
- Random placement of the labels inside a given region.

The algorithm

The Random Label Layout algorithm is not really a useful layout algorithm. It simply places the labels at randomly-computed positions inside a user-defined region. Nevertheless, the Random Label Layout algorithm may be useful for demonstration purposes.

Code sample

Below is a code sample using the `IlvRandomLabelLayout` class:

```
 ...
import ilog.views.*;
import ilog.views.graphlayout.labellayout.*;
import ilog.views.graphlayout.labellayout.random.*;
 ...
IlvManager manager = new IlvManager();
IlvManagerView view = new IlvManagerView(manager);

 ... /*  Fill in here code that fills the manager with labels and obstacles
*/

IlvRandomLabelLayout layout = new IlvRandomLabelLayout();
layout.attach(manager);
layout.setLayoutRegion(new IlvRect(0, 0, 200, 200));
IlvLabelLayoutReport layoutReport = layout.performLayout();
if (layoutReport.getCode() == IlvLabelLayoutReport.LAYOUT_DONE)
     System.out.println("Layout done.");
else
     System.out.println("Layout not done, code = " +
                                   layoutReport.getCode());
```

Generic features and parameters

The IlvRandomLabelLayout class supports the following generic parameters defined in the IlvLabelLayout class:

♦ *Allowed time*

♦ *Percentage of completion calculation*

♦ *Random generator seed value*

♦ *Save parameters to named properties*

♦ *Stop immediately*

♦ *Use default parameters*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed time

The label layout algorithm stops if the allowed time setting has elapsed. This feature works similarly as in IlvGraphLayout; see *Allowed time*. If the layout stops early because the allowed time has elapsed, the result code in the layout report is IlvLabelLayoutReport.STOPPED_AND_INVALID.

Percentage of completion calculation

The label layout algorithm calculates the estimated percentage of completion. This value can be obtained from the label layout report during the run of the layout. (For a detailed description of this feature, see *Percentage of completion calculation* and *Layout events and listeners*.)

Random generator seed value

The Random Label Layout uses a random number generator to compute the coordinates. For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

You can specify a particular value to be used as a seed value. For example, to specify the seed value 10, call:

```
layout.setUseSeedValueForRandomGenerator(true);
layout.setSeedValueForRandomGenerator(10);
```

Save parameters to named properties

The label layout algorithm can save its layout parameters into named properties. This can be used to save layout parameters to `.ivl` files. (For a detailed description of this feature, see *Using named properties to save layout parameters*.)

Stop immediately

The label layout algorithm stops after cleanup if the method `stopImmediately` is called. (This method works for the `IlvLabelLayout` class similarly to the corresponding method in the `IlvGraphLayout` class. For a description of this method in the IlvGraphLayout class, see *Stop immediately*. If the layout stops early in this way, the result code in the layout report is `IlvLabelLayoutReport.STOPPED_AND_INVALID`.

Use default parameters

After modifying any label layout parameter, you may want the layout algorithm to use the default values. You select the default values for all parameters by:

```
layout.setUseDefaultParameters(true);
```

IBM® ILOG® JViews Diagrammer keeps the previous settings when selecting the default values mode. You can switch between back to your own settings by:

```
layout.setUseDefaultParameters(false);
```

Specific parameters

The following parameter is specific to the `IlvRandomLabelLayout` class:

Layout region

The Random Label Layout algorithm places the labels randomly in a specified region. You can set this region by:

```
setLayoutRegion(IlvRect region)
```

You can obtain the current region by:

```
getLayoutRegion()
```

If no layout region is specified, the default region (0, 0, 100, 100) is used.

Using advanced features

Describes advanced features for using IBM® ILOG® JViews Diagrammer Label Layout.

In this section

General information

All the advanced features are available only for programming in Java™ , they cannot specified by CSS. The label layout renderer uses these advanced features internally, thus if you are using a diagram component with CSS styling, you do not need to learn about these advanced features. However, if you program label layout in Java, it gives you a powerful way to adapt or extend the label layout.

Filtering manager layers

Graphic objects within an `IlvManagerLayer` instance can be managed by layers. IBM® ILOG® JViews Diagrammer allows you to specify that only the labels and obstacles belonging to certain layers must be taken into account when performing the layout. Use the following methods of the `IlvDefaultLabelingModel` class:

```
void addLayer(IlvManagerLayer layer)
```

```
boolean removeLayer(IlvManagerLayer layer)
```

```
boolean removeAllLayers()
```

To get an enumeration of the manager layers to be taken into account during the layout, use the method:

```
Enumeration getLayers()
```

To determine whether a manager layer belongs to the layers to be taken into account during layout, use the method:

```
boolean isLayerAdded(IlvManagerLayer layer)
```

If no layers have been specified or all the specified layers have been removed, all layers in the manager are used. In this case, the `getLayers` method returns `null` and `isLayerAdded` returns `false` for any layer.

Transformers for label layout

The most natural transformer value that could be chosen is the “identity” transformer.

An identity transformer has no translation, zoom, or rotation factors. In terms of IBM® ILOG® JViews, this would mean that the geometry of the `IlvManager` would be considered in the manager coordinates, not in the manager view coordinates (transformed coordinates). However, the special case of *nonzoomable* graphic objects must be taken into account. For this case, the idea of simply using the geometry of the grapher in manager coordinates is not pertinent.

Label layout algorithms have to deal with the geometry of labels and obstacles. In a manager, labels and obstacles can be any graphic objects, that is, any subclass of `IlvGraphic`. Their position and size are given by their `boundingBox(IlvTransformer t)` method and usually depend on the transformer used for their display. Therefore, when you need to perform layout on a `IlvManagerLayer` object, you need to consider the geometry of the manager for a given value of the transformer.

Nonzoomable graphic objects as labels and obstacles

A graphic object is said to be zoomable if its bounding box follows the zoom level. Otherwise, the object is nonzoomable. For instance, `IlvLabel` objects are nonzoomable while `IlvZoomableLabel` objects are zoomable. To determine whether a graphic object is zoomable, use its boolean zoomable() method or check its documentation.

If all the graphic objects of an `IlvManager` instance are zoomable, a layout obtained on the basis of the graph geometry in manager coordinates will appear the same for any value of the transformer used for the display. Simply speaking, the drawing of the manager will just be zoomed, or translated.

When at least one nonzoomable graphic object is used as a label or obstacle, the geometry in manager coordinates can no longer be used. When drawn with different transformer values (for instance, at different zoom levels), the same `IlvManager` instance may look very different. In this case, you cannot use multiple manager views because only one of them can look correct. All other views will look wrong.

Reference transformer for labeling

The reference transformer is the transformer that is currently being used for..........

How a reference transformer is used

The mechanism by which the reference view and reference transformer are used is similar to that of the `IlvGrapherAdapter` class in graph layout. For details, see *Reference transformer for grapher* and *Reference views*.

Note: The reference transformer is used only if the coordinates mode is `IlvLabelLayout.VIEW_COORDINATES` or `IlvLabelLayout.INVERSE_VIEW_COORDINATES`. If you change the reference transformer, the layout is no longer up-to-date. A subsequent layout is necessary. Hence, changing the zoom level of the reference view also renders the layout out-of-date.

Reference views

To specify the reference view, use the method:

```
void setReferenceView(IlvManagerView view)
```

If no reference view is explicitly specified, the first manager view is used.

The default labeling model needs to know the transformer of the reference view or an explicitly specified reference transformer.

Specifying a reference transformer

To specify the reference transformer explicitly use the method:

```
void setReferenceTransformer(IlvTransformer transformer)
```

Specifying the mode for labeling coordinates

The default labeling model provides several coordinates mode values. The coordinates mode can be specified on the default labeling model and on the layout instance. The coordinates mode of the layout instance is used during layout, while the coordinates mode specified for the default labeling model is used on operations of the labeling model when layout is not currently performed.

To specify the coordinates mode, use the following method, available in the classes `IlvDefaultLabelingModel` and `IlvLabelLayout`:

```
void setCoordinatesMode(int mode)
```

Valid options of the coordinates mode are:

- `IlvLabelLayout.MANAGER_COORDINATES` - The geometry of obstacles and labels is computed using the coordinate space of the manager without applying any transformation. This mode is suitable if the manager does not contain any nonzoomable labels.
- `IlvLabelLayout.VIEW_COORDINATES` - The geometry of obstacles and labels is computed in the coordinate space of the reference manager view. This mode is suitable if the manager contains nonzoomable objects. The layout will be correct with respect to the reference view but not correct with respect to any other view. Dimensional layout parameters (such as the label offset) are specified in the coordinate space of the reference view.
- `IlvLabelLayout.INVERSE_VIEW_COORDINATES` - This is the default. The geometry of the graph is computed using the coordinate space of the reference manager view and then applying the inverse transformation. This simulates the manager coordinate space. This mode is also suitable if the manager contains nonzoomable objects. The layout will be correct only with respect to the reference view. Dimensional layout parameters (such as the label offset) are specified in the coordinate space of the manager.

To make sure that manager coordinates are used during layout, call:

```
layout.setCoordinatesMode(IlvLabelLayout.MANAGER_COORDINATES);
```

This does not change the coordinates mode of the labeling model until layout is started. Most of the time, however, it is recommended that you use the same coordinates mode for the default labeling model and the layout instance, so you call it twice:

```
defaultLabelingModel.setCoordinatesMode(IlvLabelLayout.MANAGER_COORDINATES);
layout.setCoordinatesMode(IlvLabelLayout.MANAGER_COORDINATES);
```

Using named properties to save layout parameters

IBM® ILOG® JViews Diagrammer Graph Layout offers the facility to convert graph layout parameters into named properties of `IlvGrapher`. The same facility is available to convert label layout parameters into named properties of the `IlvManager`. Named properties can be stored in `.ivl` files.

The following method indicates whether a label layout class supports this mechanism:

`supportsSaveParametersToNamedProperties()`

It returns `true` if the layout class can transfer the parameter settings to named properties.

Saving layout parameters to .ivl files

The following example shows how to save an `IlvManager` instance, including all label layout parameter settings, to an `.ivl` file.

```
IlvDefaultLabelingModel labelingModel =
   (IlvDefaultLabelingModel)labelLayout.getLabelingModel();
// transfer the layout parameters to named properties
if (labelLayout.supportsSaveParametersToNamedProperties())
    labelingModel.saveParametersToNamedProperties(labelLayout, false);
// save the attached manager with the named layout properties to file
labelingModel.getManager().write("abcd.ivl");
// remove the named layout properties because they are no longer needed
labelingModel.removeParametersFromNamedProperties();
```

The mechanism is the same as in the graph layout module. See *Saving layout parameters to .ivl files*.

Loading layout parameters from .ivl files

The following example shows how to load and recover the parameters of the label layout when the layout settings are stored in an `.ivl` file:

```
// Read the IVL file. This reads all named properties as well.
manager.read("abcd.ivl");
IlvDefaultLabelingModel labelingModel =
   (IlvDefaultLabelingModel)labelLayout.getLabelingModel();
// Transfer the parameter settings from the named properties to the layout.
// At this time point, the manager must be attached to the label layout
labelingModel.loadParametersFromNamedProperties(labelLayout);
// just to be sure that no named layout properties remain in the memory
labelingModel.removeParametersFromNamedProperties();
```

The mechanism is the same as in the graph layout Module. See *Loading layout parameters from .ivl files*.

Defining your own labeling model

Describes how to develop a custom label layout algorithm if you need one.

In this section

The need for a custom label layout algorithm

Note: Before reading this section, you should be familiar with the IlvLabelingModel class (see *Labels and obstacles in Java*).

It is sometimes necessary to add label layout features to an existing application.

If the application uses a diagram component with styling, using the Annealing Label Layout is a straightforward process. You use the internal labeling model of the label layout renderer, but you do not need to worry about the details.

If your application is not based on styling, but already uses the class `IlvManagerLayer` to manipulate and display labels and obstacles, you can use and adapt the default labeling model (`IlvDefaultLabelingModel`).

Even if the application uses data structures that are independent of the IBM® ILOG® JViews data structures, it is possible to apply a supplied label layout algorithm.

If you need to define your own labeling model, create a subclass of `IlvLabelingModel`. If the layout is to support rotated labels, your subclass must additionally implement the interface `IlvLabelingModelWithRotation`.

See also

Labels and obstacles in Java

The IlvLabelingModel Class

The methods defined in the `IlvLabelingModel` class can be divided into several categories: those that provide information on the structure of the labels and obstacles, on their geometry, on their overlap penalty, and notification of changes in the manager.

They are described in the following sections:

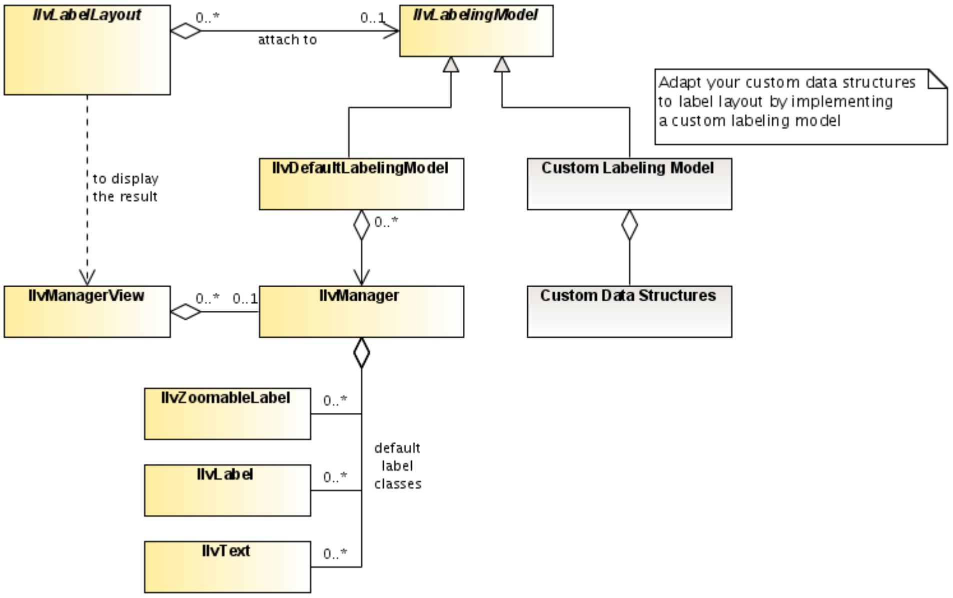
♦ *Information on the structure of labels and obstacles*

♦ *Information on the geometry of labels and obstacles*

♦ *Overlap calculation*

♦ *Notification of changes*

♦ *Storing and retrieving object properties*



The Class IlvLabelingModel in the JViews Graphic Framework

Information on the structure of labels and obstacles

The following methods of the `IlvLabelingModel` class allow the layout algorithms to retrieve information on the labels:

`isLabel(java.lang.Object)`	Decides whether an object is a label.
`getLabels()`	Enumerates all existing labels.
`getLabelsCount()`	Returns the number of labels that exist.

The following methods allow the layout algorithms to retrieve information on the obstacles in a similar way:

```
boolean isObstacle(Object obj)
```

```
boolean isPolylineObstacle(Object obj)
```

```
Enumeration getObstacles()
```

```
int getObstaclesCount()
```

For optimization purposes, the labeling model distinguishes between normal obstacles and polyline obstacles. While normal obstacles cover the major part of their bounding box, polyline obstacles have a line width and range over intermediate points; thus they cover only a small part of their bounding box.

Since a polyline obstacle is an obstacle, both `isObstacle` and `isPolylineObstacle` return `true` for a polyline obstacle.

Information on the geometry of labels and obstacles

For labels and obstacles, the label layout can retrieve the bounding box with the method:

```
IlvRect boundingBox(Object labelOrObstacle)
```

For the special polyline obstacles, the label layout can retrieve the precise shape of the polyline with the methods:

```
float getPolylineWidth(Object polylineObstacle)
```

```
IlvPoint[] getPolylinePoints(Object polylineObstacle)
```

The following method moves a label to the new position.

```
void moveLabel(Object label, float x, float y, boolean redraw)
```

Overlap calculation

A good label layout avoids overlaps. Thus, the calculation of overlap values is an important step of the algorithm. The speed of the layout algorithm depends crucially on the speed of the overlap calculation. The labeling model provides the following methods for overlap calculations:

`getLabelOverlap(java.lang.Object, ilog.views.IlvRect, java.lang.Object, ilog.views.IlvRect, float)`	Calculates the overlap between two labels.
`getObstacleOverlap(java.lang.Object, ilog.views.IlvRect, java.lang.Object, ilog.views.IlvRect, float)`	Calculates the overlap between a label and a normal obstacle.
`getPolylineObstacleOverlap(java.lang.Object, ilog.views.IlvRect, java.lang.Object, ilog.views.IlvPoint[], float, float)`	Calculates the overlap between a label and a polyline obstacle.

These methods return a positive penalty value that indicates how much the objects overlap.

♦ The returned value is 0 if the objects do not overlap.

♦ A smaller overlap value designates less overlap than a larger overlap value. The actual number is arbitrary and depends on the implementer of the labeling model. For example, if all objects are rectangles, then it could be the size of the overlapping area of the rectangles.

Typically the overlap value is calculated before the label is moved. It is calculated for a speculative label position, not for the real label position. Hence, the speculative bounding box of the label is passed as an argument. Similarly, the bounding box (or polyline shape) of the obstacle is passed as an argument as well. The meaning of the returned value is the overlap penalty if the label were placed at its passed bounding box, and the obstacle were placed at its passed bounding box.

If the overlap methods return a positive nonzero penalty only when the speculative bounding boxes overlap, then the following method can return `true`:

```
boolean isBoundingBoxDependent()
```

This method exists mainly for quadtree optimization purposes.

Notification of changes

The following methods of the `IlvLabelingModel` class allow a layout algorithm to be notified of changes in the data structures:

```
void addLabelingModelListener(LabelingModelListener listener)
void removeLabelingModelListener(LabelingModelListener listener)
```

```
void fireLabelingModelEvent(Object obstacleOrLabel, int eventType, boolean
adjusting)
```

```
void fireLabelingModelEvent(LabelingModelEvent event)
```

A “change” can be a structural change (that is, a label or obstacle was added or removed) or a geometrical change (that is, a label or obstacle was moved or reshaped). The event type is typically a bitwise-Or of the bit masks defined in the class `LabelingModelEvent`. For instance, when a label was removed, an event with type `(STRUCTURE_CHANGED | LABEL_REMOVED)` is fired and the removed label is stored in the event. The labeling model event listener mechanism provides a means to keep the layout algorithms informed of these changes. When the layout algorithm is restarted on the same graph, it is able to detect whether the data structures have changed since the last time the layout was successfully performed. If necessary the layout can be performed again. If there was no change, the layout algorithm can avoid unnecessary work by not performing the layout.

The labeling model event listener is defined by the `LabelingModelListener` interface. To receive the events (that is, instances of the `LabelingModelEvent` class), a class must implement the `LabelingModelListener` interface and must register itself using the `addLabelingModelListener(ilog.views.graphlayout.labellayout.LabelingModelListener)` method of the `IlvLabelingModel` class.

Note: The label layout algorithms register themselves as listeners to the labeling model (via the functionality in `IlvLabelLayout`). Therefore, there is usually no need to manipulate these listeners directly.

Storing and retrieving object properties

The following methods of the `IlvLabelingModel` class allow a layout algorithm to store data objects for each label or obstacle:

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

```
Object getProperty(Object labelOrObstacle, String key)
```

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

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

The layout algorithm may need to associate a set of properties with the labels or obstacles, or global properties. Properties are a set of `key-value` pairs, where the `key` is a `String` object and the `value` can be any kind of information value.

The IlvLabelingModelWithRotation Interface

The methods of the `IlvLabelingModel` class are designed to support upright rectangular labels. If rotated rectangular labels occur in your application, the labeling model must additionally implement the interface `IlvLabelingModelWithRotation`.

In `IlvLabelingModelWithRotation`, all labels are considered rectangles that can be rotated around the center of the label. Hence, the method `boundingBox(java.lang.Object)` of the labeling model returns the bounding box of the unrotated label, not the bounding box of the rotated label.

The `IlvLabelingModelWithRotation` interface assumes that the rotation of the label depends on the position of the label. For example, if the label is placed close to link segments, the label should be rotated according to the link segments.

Rotations

You can retrieve the rotation of the label with the method:

```
double getRotation(Object label, IlvRect positionRectangle)
```

You can set the rotation of the label with the method:

```
void setRotation(Object label, double angle)
```

The angles are in degrees.

The method `setRotation(Object label, double angle)` is called by the Annealing Label Layout at the end of layout to inform the label about the final rotation calculated by the layout. If the label is attached to a link and always follows the rotation of the link automatically, it will not be necessary to inform the label about the final rotation. Therefore, `setRotation(`Object label, double angle`)` can be empty, but `getRotation(`Object label, IlvRect positionRectangle`)` must return the rotation of the link segment when the label is placed at `positionRectangle`.

Overlap calculations

The `IlvLabelingModelWithRotation` interface offers methods for calculating the overlaps of rotated labels. These methods have as additional parameter an angle for the label parameters:

Methods for Calculating Overlaps of Rotated Labels

```
double getLabelOverlap(Object label1, IlvRect rect1,
    double angle1, Object label2, IlvRect rect2,
    double angle2, float minDist);

double getObstacleOverlap(
    Object label, IlvRect labelRect, double angle,
    Object obstacle, IlvRect obstacleBBox, float minDist);
```

```
double getPolylineObstacleOverlap(
    Object label, IlvRect labelRect, double angle,
    Object polylineObstacle, IlvPoint[] pts,
    float lineWidth, float minDist);
```

For each label parameter, an unrotated rectangle (`labelRect`) is passed, which defines the speculative position of the label and a rotation angle.

The meaning of the returned value is the overlap penalty if the label were placed at `labelRect` and rotated by the angle and if the obstacle were placed at `obstacleBBox`. The penalty `0` means that the objects do not overlap.

Subclassing the default labeling model

The default labeling model `IlvDefaultLabelingModel` is a subclass of `IlvLabelingModel`. It has certain properties that may not be suitable for your application:

- Only objects of type `IlvLabel` and `IlvZoomableLabel` are considered labels.
- Only objects of type `IlvLine`, `IlvPolyline`, and `IlvLinkImage` are considered polyline obstacles.
- All other objects are considered rectangular obstacles.

It is easy to change these properties by subclassing `IlvDefaultLabelingModel`. For instance, if you use only labels of class `MySpecialLabel`, you can override the method `isDefaultLabelClass`:

```
public boolean isDefaultLabelClass(Object obj)
{
    return (obj instanceof MySpecialLabel);
}
```

If you want some objects to be completely ignored, make sure they are not considered as labels or obstacles. To avoid considering objects of class `IgnorableGraphic` as obstacles, you can override the method `isObstacle`:

```
public boolean isDefaultObstacleClass(Object obj)
{
    return super.isDefaultObstacleClass(obj) &&
           !(obj instanceof IgnorableGraphic);
}
```

Note: Instead of overriding `isDefaultLabelClass` and `isDefaultObstacleClass`, you can also specify which objects are labels and obstacles by using the methods `setLabel` and `setObstacle`, as illustrated in *Labels and obstacles in Java*.

Some obstacles do not have a rectangular shape. For simplicity and speed, the overlap value is based on the bounding box of normal obstacles. Hence the default labeling model may compute overlaps in situations where in fact there are no overlaps, because the bounding box is usually a little larger than the area that is really covered by the obstacle. You can correct this by overriding the method `getOverlapValue` by implementing a more precise (but also more complex) overlap test:

```
public double getObstacleOverlap(
    Object label, IlvRect labelBBox,
    Object obstacle, IlvRect obstacleBBox,
    float minDist)
  {
    ... complex calulation of the overlap considering the precise shape of
    ... the obstacle. If the label is closer than minDist to the obstacle,
    ... it should be considered as overlap
    ...
    if (hasOverlap)
        return a value proportional to the overlap;
    else
        return 0.0;
   }
```

Since the default labeling model `IlvDefaultLabelingModel` implements the `IlvLabelingModelWithRotation` interface, you can similarly override all overlap methods that have a rotation angle parameter for the labels. See *The IlvLabelingModelWithRotation Interface*.

Creating a new labeling model

The default labeling model is suitable only if the underlying data structure is an `IlvManager`. The case may arise when an application uses its own classes and when, for some reason, you do not want to replace these classes with IBM® ILOG® JViews classes such as `IlvManager` and `IlvLabel`. Here, you cannot use the default labeling model. To enable the label layout algorithms to work with these data structures, you must write a custom labeling model (that is, a subclass of `IlvLabelingModel`).

The custom labeling model must implement all the *abstract* methods of the `IlvLabelingModel` class. The nonabstract methods of this class have a default implementation that is functional. However, they may not be optimal because they do not take advantage of the characteristics of the underlying graph implementation. In this case, they can be overridden as well. The efficiency of the layout algorithm depends directly on the efficiency of the implementation of the labeling model and the underlying data structure.

The following is the minimal set of methods that must be implemented:

```
abstract Enumeration getLabels()
```

```
abstract boolean isLabel(Object obj)
```

```
abstract void moveLabel(Object label, float x, float y, boolean redraw)
```

```
abstract double getLabelOverlap(Object label1, IlvRect bbox1, Object label2,
IlvRect bbox2, float minDist)
```

```
abstract Enumeration getObstacles()
```

```
abstract boolean isObstacle(Object obj)
```

```
abstract double getObstacleOverlap(Object label, IlvRect labelBBox, Object
obstacle, IlvRect obstacleBBox, float minDist)
```

```
abstract IlvRect boundingBox(Object labelOrObstacle)
```

These methods are described in *The IlvLabelingModel Class*.

If the label layout algorithm is to support rotated labels, the new labeling model must additionally implement the interface `IlvLabelingModelWithRotation`.

Using advanced features

Describes advanced features including how to define new types of layouts.

In this section

Overview of advanced features

The advanced features are available only for programming in Java™ . If you program graph layout in Java, these advanced features give you a powerful way to adapt or extend graph layouts.

Unless otherwise specified, advanced features cannot be specified in CSS syntax. The graph layout and link renderers use the advanced features internally. Therefore, if you are using a diagram component with CSS styling, you can ignore this section.

Using a graph layout report

Describes what graph layout reports are and how to use them.

In this section

Layout report classes

Graph layout reports are objects used to store information about the particular behavior of a layout algorithm. After the layout is completed, this information is available to be read from the layout report.

Each layout class instantiates a particular class of `ilog.views.graphlayout.IlvGraphLayoutReport` each time the layout is performed. The following table shows the layout classes and their corresponding layout reports.

Layout report classes

Layout Class	Layout Report Class
`IlvTopologicalMeshLayout`	`IlvTopologicalMeshLayoutReport`
`IlvUniformLengthEdgesLayout`	`IlvUniformLengthEdgesLayoutReport`
`IlvTreeLayout`	`IlvGraphLayoutReport`
`IlvHierarchicalLayout`	`IlvGraphLayoutReport`
`IlvLinkLayout`	`IlvGraphLayoutReport`
`IlvRandomLayout`	`IlvGraphLayoutReport`
`IlvBusLayout`	`IlvGraphLayoutReport`
`IlvCircularLayout`	`IlvGraphLayoutReport`
`IlvGridLayout`	`IlvGraphLayoutReport`
`IlvMultipleLayout`	`IlvMultipleLayoutReport`
`IlvRecursiveLayout`	`IlvRecursiveLayoutReport`

Creating a layout report

All layout classes inherit the `performLayout` method from the `IlvGraphLayout` class. This method calls `createLayoutReport` to obtain a new instance of the layout report. This instance is returned when `performLayout` returns. The default implementation in the base layout class creates an instance of `IlvGraphLayoutReport`. Some subclasses override this method to return an appropriate subclass. Other classes, such as `IlvRandomLayout`, do not need specific information to be stored in the layout report and do not override `createLayoutReport`. In this case, the base class `IlvGraphLayoutReport` is used.

When using the layout classes with IBM® ILOG® JViews Diagrammer, you do not need to instantiate the layout report yourself. This is done automatically.

Accessing a layout report

In a diagram component (a subclass of `IlvDiagrammer`), you can access a layout report in the following way:

```
IlvGraphLayoutReport layoutReport =
    diagrammer.getEngine().getNodeLayoutRenderer().
    getGraphLayout().getLayoutReport();
```

Notice that `null` is returned if the layout renderer was never executed (that is, layout was never called).

If you do not use a diagram component, you usually call layout via the method `performLayout` which returns the layout report. The following example shows how to read the information from the layout report in this case:

:

```
 ...
try {
        IlvGraphLayoutReport layoutReport = layout.performLayout();
        if (layoutReport.getCode() ==
                                   IlvGraphLayoutReport.LAYOUT_DONE)
                System.out.println("Layout done.");
        else
                System.out.println("Layout not done, code = " +
                                 layoutReport.getCode());
}
catch (IlvGraphLayoutException e) {
        System.err.println(e.getMessage());
}
```

Information stored in a layout report

The base class `IlvGraphLayoutReport` stores the following information:

- *Code*
- *Layout time*
- *Percentage of completion*
- *Additional information*

Code

This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

- `LAYOUT_DONE` appears if the layout was performed successfully.
- `STOPPED_AND_VALID` appears if the layout was performed but was stopped before completion, either because the layout time elapsed or because the method `stopImmediately` was called. The positions of nodes and links are valid at the stopping point because the layout algorithm uses an iterative mechanism.
- `STOPPED_AND_INVALID` appears if a (noniterative) layout was performed but was stopped before completion, either because the layout time elapsed or because the method `stopImmediately` was called. The positions of nodes and links are not valid at the stopping point. Often, they have not yet been changed at all.
- `NOT_NEEDED` appears if the layout was not performed because no changes occurred in the grapher and parameters since the last time the layout was performed successfully.
- `EMPTY_GRAPHER` appears if the grapher is empty.

To read the code, use the method:

```
int getCode()
```

Layout time

This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

```
long getLayoutTime()
```

Percentage of completion

This field contains an estimation of the percentage of the layout that has been completed. This can be used if the layout algorithm supports the generic percentage completion calculation feature. (See *Percentage of completion calculation*.) It is typically used inside layout event listeners that are described in the following section. To access the percentage, use the method:

```
int getPercentageComplete()
```

Additional information

Additional information for particular layout algorithms is stored by the subclasses of `IlvGraphLayoutReport`. For details, see the reference documentation of these classes:

- `IlvTopologicalMeshLayoutReport`
- `IlvUniformLengthEdgesLayoutReport`
- `IlvMultipleLayoutReport`
- `IlvRecursiveLayoutReport`

Using event listeners

All layout classes support two kinds of events: layout events and parameter events. The listening mechanism therefore provides:

- *Graph layout event listeners*
- *Parameter event listeners*

Graph layout event listeners

The layout event listening mechanism provides a way to inform the end user of what is happening during the layout. At times, a layout algorithm may take a long time to execute, especially when dealing with large graphs. In addition, an algorithm may not converge in some cases. No matter what the situation, the end user should be informed of the events that occur during the layout. This can be done by implementing a simple progress bar or by displaying appropriate information, such as the percentage of completion after each iteration or step.

The layout event listener is defined by the `GraphLayoutEventListener` interface. To receive the layout events delivered during the layout, a class must implement the `GraphLayoutEventListener` interface and must register itself using the `addGraphLayoutEventListener` method of the `IlvGraphLayout` class.

When you implement the `GraphLayoutEventListener` interface, you must implement the `layoutStepPerformed` method. The layout algorithm will call this method on all the registered layout event listeners, passing the layout report as an argument (see *Using a graph layout report*). In this way, you can read information about the current state of the layout report. (For example, you can read this information after each iteration or step of the layout algorithm).

The following example shows how to implement a layout event listener:

```
class LayoutIterationListener
  implements GraphLayoutEventListener
{
  public void layoutStepPerformed(GraphLayoutEvent event)
  {
    IlvGraphLayoutReport layoutReport = event.getLayoutReport();
    System.out.println("percentage of completion: " +
                       layoutReport.getPercentageComplete());
  }
}
```

Then, register the listener on the layout instance as follows:

```
layout.addGraphLayoutEventListener(new LayoutIterationListener());
```

Parameter event listeners

The layout parameter event listeners mechanism provides a way to inform the end user that any layout parameter has changed. This is useful when the layout parameter values are displayed in a dialog box that needs to be updated to indicate parameter changes.

The parameter event listener is defined by the `GraphLayoutParameterEventListener` interface. To receive the layout parameter events, a class must implement the `GraphLayoutParameterEventListener` interface and must register itself using the `addGraphLayoutParameterEventListener` method of the `IlvGraphLayout` class.

When you implement the `GraphLayoutParameterEventListener` interface, you must implement the `parametersUpToDate` method. The layout class will call this method on all the registered layout parameter event listeners. The layout parameter event contains a flag accessible by the `isParametersUpToDate` method:

♦ It returns `true` if the event occurs at the end of a run of the layout when the layout is considered up-to-date with respect to the layout parameters.

♦ It returns `false` if the event occurs when any layout parameter has changed.

The following example shows how to implement a layout parameter event listener.

```
class LayoutParameterListener
  implements GraphLayoutParameterEventListener
{
  public void parametersUpToDate(GraphLayoutParameterEvent event)
  {
    if (!event.isParametersUpToDate())
      System.out.println("Any layout parameter has changed.”);
  }
}
```

Then, register the listener with the layout instance as follows:

```
layout.addGraphLayoutParameterEventListener(new LayoutParameterListener());
```

Redrawing the grapher after layout

When a layout algorithm is executed, it moves the nodes and/or reshapes the links of the graph. If the graph is displayed on a screen, its display must be updated to reflect the changes made by the layout.

- ♦ If you use a diagram component (a subclass of `IlvDiagrammer`)), the updating is done automatically.
- ♦ If you call layout by using the method `performLayout`, you have more detailed control on the redraw mechanism, as explained in the following sections:

  IBM® ILOG® JViews provides complete flexibility, concerning the redraw of an `IlvGrapher` instance that has undergone layout. Your choice will depend on your particular application. The following scenarios are possible:

  - *Automatic and Selective Redraw*
  - *Nonautomatic and Complete Redraw*
  - *Delayed Redraw*
  - *No Redraw at All*

Automatic and Selective Redraw

If you just want the grapher to be automatically redrawn after the layout, simply call the following method with the value `true` for the `redraw` argument:

```
IlvGraphLayoutReport performLayout(boolean force, boolean redraw)
```

When you do this, an `initReDraws()`/ `reDrawViews()` session is initiated automatically. When the nodes and the links are moved or reshaped, the value `true` is passed for the `redraw` argument of the appropriate methods of `IlvGrapher`. At the end of the layout, the `initReDraws/reDrawViews` session is ended. This produces a selective redraw of the invalid regions of the views where the graph is displayed.

Nonautomatic and Complete Redraw

If all the nodes are moved by the layout, it may be more efficient to redraw the entire graph at the end of the layout instead of using the mechanism of the invalid regions provided by `IlvGrapher`. In this case, you can use the following code:

```
try {
  layout.performLayout(false, false); // argument redraw at false
} catch (IlvGraphLayoutException e) {
  e.printStackTrace();
} finally {
  // redraw in the final clause to ensure that the redraw
  // is performed even if an exception occurs
  grapher.reDraw();
}
```

This completely redraws the grapher in all its visible views. Alternatively, you can call the method `repaint` on some of its views.

Delayed Redraw

After the layout, you may want to perform other changes in the grapher before redrawing. You can start the `initReDraws()`/ `reDrawViews()` session on your own to control the point in time when the redraw is performed. You can use the following code:

```
grapher.initReDraws();
try {
  layout.performLayout(false, true);
} catch (IlvGraphLayoutException e) {
  e.printStackTrace();
} finally {
  grapher.moveObject(..., false); // some other changes in the grapher
  grapher.reDrawViews();
}
```

No Redraw at All

Sometimes, the layout may need to be performed without any display of the grapher. For instance, this can be done to automatically produce `.ivl` files containing the result of the layout for future use. To avoid any redraw during the layout, just call the method `performLayout(boolean, boolean)` with the value `false` for the `redraw` argument.

Note: During animated layout (for layouts supporting this option), the grapher needs to be redrawn after each step to produce the animation effect. Therefore, you need to pass the value `true` for the `redraw` argument of the method `performLayout(boolean, boolean)`. In this case, the `initReDraws()`/ `reDrawViews()` session is automatically used for each animation step. Users should not add their own `initReDraw/reDrawViews` session because this would prevent the graph from being redrawn during the animation.

Using the Graph Model

Describes the graph model.

In this section

Overview of the graph model

The IlvGraphModel class defines a suitable, generic API for graphs that have to be laid out with IBM® ILOG® JViews Diagrammer graph layout algorithms.

All the layout algorithms provided in IBM® ILOG® JViews Diagrammer are designed to lay out a graph model. This allows applications to benefit from the graph layout algorithms whether or not they use the IBM® ILOG® JViews grapher (IlvGrapher). However, to make things very simple for the common case of applications that manipulate an IlvGrapher, it is not mandatory to work directly with the graph model except for some advanced features such as filtering (see *Using filtering to lay out a part of an IlvGrapher*).

Graph model and SDM model

Defines the SDM model and the graph model.

There are two key concepts: the *graph model* and the *SDM model* of the diagram component (subclass of `IlvDiagrammer`). It is important to avoid confusion between them.

The SDM model

The SDM model represents the application objects. Application objects typically have only logical properties, not graphic properties (position, size, shape, and so on). The rendering process produces graphic objects for the SDM model objects.

The Graph Model

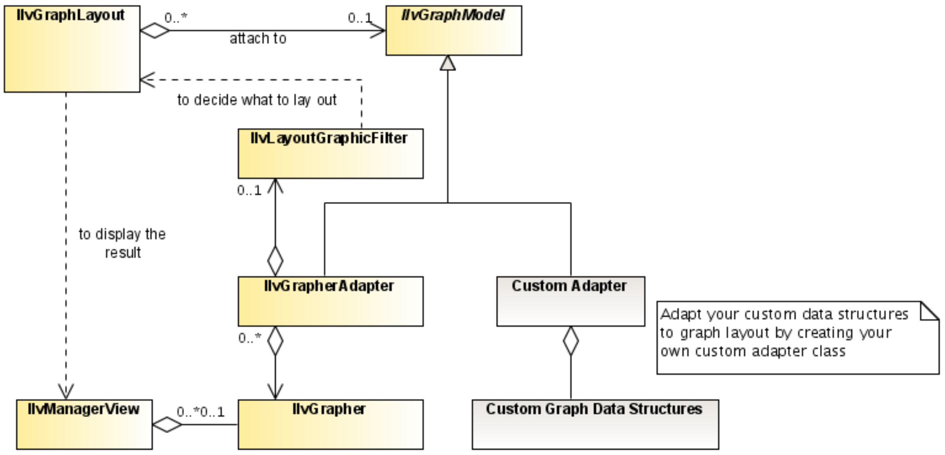
The graph model of the layout algorithms is an abstraction of the graphic properties of these graphic objects., not an abstraction of the application objects.

The graph model concept

With a graph model, you can use already-built graphs, nodes, and links that have been developed without IBM® ILOG® JViews and apply the layout algorithms of IBM® ILOG® JViews Diagrammer. The graph model provides the basic, generic operations for performing the layout.

A subclass must be written to adapt the graph model to specific graph, node, and link objects. This subclass plays the role of an “adapter” or bridge between the application objects and the graph model. This often makes it much easier to add graph features to existing applications.

The following figure shows the relationship between the graph model and graph layout algorithms, IBM® ILOG® JViews graphers, non-JViews graphers, and manager views.



Graph Model in the IBM® ILOG® JViews Diagrammer Graph Layout Framework

You can see from this diagram that instead of using a concrete graph class such as `IlvGrapher` directly, the layout algorithms interact with the graph via the graph model. This is the key for achieving a truly generic graph layout framework. Note that the use of an `IlvManagerView` to display the result of the layout is not mandatory.

The IlvGraphModel class

The `IlvGraphModel` class is an abstract Java™ class. Because it does not provide a concrete implementation of a graph data structure, a complete implementation must be provided by "adapter" classes. The adapters extend the `IlvGraphModel` class and must use an underlying graph data structure. A special adapter class called `IlvGrapherAdapter` is provided so that an `IlvGrapher` can be used as the underlying graph data structure.

Note: If an application uses the `IlvGrapher` class, the grapher can be attached directly to the layout instance without explicitly using a graph model. (See the `attach(ilog.views.IlvGrapher)` method.) In this case, the appropriate adapter (`IlvGrapherAdapter`) will be created internally. This adapter can be retrieved using the `getGraphModel()` method, which will return an instance of `IlvGrapherAdapter`.

Most of the methods defined in the `IlvGraphModel` class have a name and definition very similar to the corresponding methods of the `IlvGrapher` class. The main difference is that the arguments of the `IlvGraphModel` methods are `java.lang.Object` instead of `IlvGraphic` or `IlvLinkImage`. The methods can be divided into several categories that provide information on the structure of the graph, the geometry of the graph, modification of the graph geometry, and notification of changes in the graph.

This section is divided as follows:

- *Information on the Structure of the Graph*
- *Information on the Geometry of the Graph*
- *Modification of the Geometry of the Graph*
- *Notification of Changes*
- *Storing and Retrieving Data Objects ("Properties")*

Information on the Structure of the Graph

The following methods of the `IlvGraphModel` class allow the layout algorithms to retrieve information on the structure of the graph:

```
Enumeration getNodesAndLinks()
```

```
Enumeration getNodes()
```

```
int getNodesCount()
```

```
Enumeration getLinks()
```

```
int getLinksCount()
```

```
boolean isNode(Object obj)
```

```
boolean isLink(Object obj)
```

```
Enumeration getLinks(Object node)
```

```
int getLinksCount(Object node)
```

```
Enumeration getLinksFrom(Object node)
```

```
int getLinksFromCount(Object node)
```

```
Enumeration getLinksTo(Object node)
```

```
int getLinksToCount(Object node)
```

```
Enumeration getNeighbors(Object node)
```

```
int getNodeDegree(Object node)
```

```
Object getFrom(Object link)
```

```
Object getTo(Object link)
```

```
Object getOpposite(Object link, Object node)
```

```
boolean isLinkBetween(Object node1, Object node2)
```

The following methods are provided for use with nested graphs (see also *Nested layouts*):

```
IlvGraphModel getParentModel()
```

```
IlvGraphModel getRootModel()
```

```
IlvGraphModel getGraphModel(Object subgraph)
```

```
IlvGraphModel createGraphModel(Object subgraph)
```

```
Enumeration getSubgraphs()
```

```
int getSubgraphsCount()
```

```
boolean isSubgraph(Object obj)
```

```
Enumeration getInterGraphLinks()
```

```
int getInterGraphLinksCount()
```

```
boolean isInterGraphLink(Object obj)
```

Information on the Geometry of the Graph

The following methods of the `IlvGraphModel` class allow the layout algorithms to retrieve information on the geometry of the graph:

```
IlvRect boundingBox(Object nodeOrLink)
```

```
IlvRect boundingBox()
```

```
IlvPoint[] getLinkPoints(Object link)
```

```
IlvPoint getLinkPointAt(Object link, int index)
```

```
int getLinkPointsCount(Object link)
```

```
float getLinkWidth(Object link)
```

The `boundingBox` method is called by a layout algorithm whenever it needs to get the position and the dimensions of a node or a link. The other methods are used mainly by link layout algorithms.

Modification of the Geometry of the Graph

The following methods of the `IlvGraphModel` class allow a layout algorithm to modify the geometry of the graph:

```
void moveNode(Object node, float x, float y, boolean redraw)
```

```
void reshapeLink(Object link, IlvPoint fromPoint, IlvPoint[] points, int
startIndex, int length, IlvPoint toPoint, boolean redraw)
```

```
void move(float x, float y, boolean redraw)
```

Layout algorithms that compute new coordinates for the nodes use the `moveNode` method. Link layout algorithms that compute new shapes for the links call one of the `reshapeLink` methods.

Notification of Changes

The following methods of the `IlvGraphModel` class allow a layout algorithm to be notified of changes in the graph:

```
void addGraphModelListener(GraphModelListener listener)
```

```
void removeGraphModelListener(GraphModelListener listener)
```

```
void fireGraphModelEvent(GraphModelEvent event)
```

```
void fireGraphModelEvent(Object nodeOrLink, int type, boolean adjusting)
```

```
void adjustmentEnd()
```

A “change” in the graph can be a structure change (that is, a node or a link was added or removed) or a geometry change (that is, a node or a link was moved or reshaped). The graph model event listener mechanism provides a means to keep the layout algorithms informed of these changes. When the layout algorithm is restarted on the same graph, it is able to detect whether the graph has changed since the last time the layout was successfully performed. If necessary, the layout can be performed again. If there is no change in the graph, the layout algorithm can avoid unnecessary work by not performing the layout. To know whether the previous layout is still valid or it must be redone, the layout algorithms call the following method of the model:

```
boolean isLayoutNeeded()
```

The graph model event listener is defined by the `GraphModelListener` interface. To receive the graph model events (that is, instances of the `GraphModelEvent`class), a class must implement the `GraphModelListener` interface and must register itself using the `addGraphModelListener(ilog.views.graphlayout.GraphModelListener)` method of the `IlvGraphModel` class.

Note: The creation of the graph model event listener is handled transparently by the `IlvGraphModel` class. Therefore, there is usually no need to manipulate this listener directly.

Storing and Retrieving Data Objects ("Properties")

The following methods of the `IlvGraphModel` class allow a layout algorithm to store data objects for each node, link, or graph:

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

```
Object getProperty(Object nodeOrLink, String key)
```

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

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

The layout algorithm may need to associate a set of properties with the nodes and links of the graph or with the graph itself. Properties are a set of `key-value` pairs, where the `key` is a `String` object and the `value` can be any kind of information value.

Note: Creating a property and associating it with a node, a link, or a graph is handled transparently by the layout algorithm whenever it is necessary. Therefore, there is usually no need to manipulate the properties directly. However, if needed, you can do this in your own subclass of `IlvGraphLayout`.

Using the class IlvGrapherAdapter

The `IlvGrapherAdapter` class is a concrete subclass of `IlvGraphModel` that allows an `IlvGrapher` to be laid out using the layout algorithms provided in IBM® ILOG® JViews Diagrammer. It provides an implementation for all the abstract methods of `IlvGraphModel`. It also provides an overridden implementation of some nonabstract methods of `IlvGraphModel` to improve efficiency by taking advantage of the characteristics of the `IlvGrapher`.

If an application uses the `IlvGrapher` class, the grapher can be attached directly to the layout instance without explicitly using the adapter. (See the method `attach(ilog.views.IlvGrapher)`.) In this case, an `IlvGrapherAdapter` is created internally by the layout class. The adapter can be retrieved using the method `getGraphModel()`, which will return an instance of `IlvGrapherAdapter`.

Notice that such an internally created adapter is not allowed to be attached to any other layout instance, nor to be used in any way once the method `detach()IlvGraphLayout.detach()` has been called on the layout instance.

In case you need to be able to do any of the above operations, directly create the instance of `IlvGrapherAdapter` and attach it using `attach(ilog.views.graphlayout.IlvGraphModel)`).

To know whether a given `IlvGraphModel` instance has been created using `attach(ilog.views.IlvGrapher)`), you can use the method `getOriginatingLayout()`. This method returns a non-`null` object if the model has not been created using `IlvGraphLayout.attach (IlvGrapher)`.

Additionally, the `IlvGrapherAdapter` class provides a way to filter the `IlvGrapher`. By using the filtering mechanism, you specify a particular set of nodes and links that have to be taken into account by the layout algorithm. (See *Using filtering to lay out a part of an IlvGrapher*.)

The `IlvGrapherAdapter` class allows you to specify the order of nodes returned by the methods `getNodes()IlvGrapherAdapter.getNodes()` and `getNodesAndLinks()`. For this purpose you can provide your own implementation of a `java.util.Comparator` that defines the order of the nodes. Then specify this comparator by using the method `setNodeComparator (java.util.Comparator)`.

The `IlvGrapherAdapter` class also allows you to specify the `IlvTransformer` that has to be used for computing the geometry of the graph. (See *Choosing the layout coordinate space*.)

Note: For details on how to write your own adapter, see *Laying out a non-JViews grapher*.

Laying out a non-JViews grapher

Important: To understand this section better, read section *Using the Graph Model* first.

It is sometimes necessary to add graph layout features to an existing application. If the application already uses the IBM® ILOG® JViews grapher (`IlvGrapher`) to manipulate and display graphs, using the graph layout algorithms provided in IBM® ILOG® JViews Diagrammer is a straightforward process. No adapter has to be written.

However, the case may arise where an application uses its own classes for nodes, links, and graphs, and where, for some reason, you do not want to replace these classes with IBM® ILOG® JViews Diagrammer classes. To enable the graph layout algorithms to work with these graph objects, a custom adapter (that is, a subclass of `IlvGraphModel`) must be written.

The adapter must implement all the abstract methods of the `IlvGraphModel` class. The nonabstract methods of this class have a default implementation that is really functional. However, they may not be optimal because they do not take advantage of the characteristics of the underlying graph implementation. For better performance, the following nonabstract methods can be overridden in the adapter class:

```
int getNodesCount()
```

```
int getLinksCount()
```

```
int getLinksCount(Object node)
```

```
int getLinksFromCount(Object node)
```

```
int getLinksToCount(Object node)
```

```
int getLinkPointAt(Object link, int index)
```

```
int getSubgraphsCount()
```

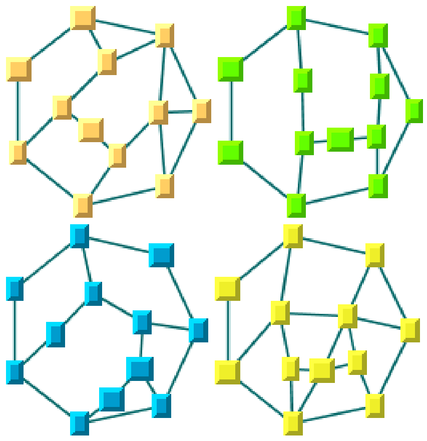
```
int getInterGraphLinksCount()
```

The efficiency of the layout algorithm depends directly on the efficiency of the implementation of the adapter class and the underlying graph data structure.

Laying out connected components of a disconnected graph

IBM® ILOG® JViews Diagrammer provides special support for the layout of a disconnected graph.

If a graph is composed of several connected components or contains isolated nodes (nodes without any links), it can be desirable to apply the layout algorithm separately on each connected component and then to position the connected components using a specialized layout algorithm (usually, `IlvGridLayout`). The following figure shows an example of a graph containing four connected components. Simply by enabling the layout of the connected components on the regular layout instance (here, `IlvTopologicalMeshLayout`), the connected components are automatically identified and laid out individually. Finally, the four connected components are positioned using a highly customizable placement algorithm (`IlvGridLayout`).



Automatic layout of connected components in a disconnected graph

To indicate whether a subclass of `IlvGraphLayout` supports this feature, use the method in the class `IlvGraphLayout`:

```
boolean supportsLayoutOfConnectedComponents()
```

The default implementation returns `false`. A subclass can override this method in order to return `true` to indicate that this feature is supported.

IBM® ILOG® JViews Diagrammer allows you to enable the layout of the connected components using the method:

```
void setLayoutOfConnectedComponentsEnabled(boolean enable)
```

To obtain the current setting:

```
boolean isLayoutOfConnectedComponentsEnabled()
```

The default value is the value returned by the following method:

```
boolean isLayoutOfConnectedComponentsEnabledByDefault()
```

The default implementation of this method in `IlvGraphLayout` returns `false`. For some of the layout classes, it is appropriate that this feature is enabled by default. Therefore `IlvUniformLengthEdgesLayout` overrides this method to return `true`.

If enabled on a layout class that supports this feature, the method `performLayout` of the class `IlvGraphLayout` cuts the attached graph model into connected components and lays out each connected component separately.

How does the layout of connected components feature work when this mechanism is enabled in the layout classes that support this feature? Instead of directly calling the method `layout (boolean)` to perform the layout on the entire graph, the method `performLayout(boolean, boolean)` first cuts the graph into connected components. Then, each connected component is laid out separately by a call of the method `layout`. To do this, the attached graph is temporarily changed into internally generated graphs corresponding to each of the connected components of the original graph. Finally, the layout instance returned by the method:

```
IlvGraphLayout getLayoutOfConnectedComponents()
```

is used to position the connected components. To specify the layout instance that places the connected components, use the following method:

```
void setLayoutOfConnectedComponents(IlvGraphLayout layout)
```

If no layout instance is specified using this method, the method `getLayoutOfConnectedComponents` returns an instance of `IlvGridLayout`. Its layout region parameter is set by default to the rectangle (`0`, `0`, `800`, `800`). Its “layout mode” parameter is set to `TILE_TO_ROWS`.

Note: The Tree, Hierarchical, and Circular layouts contain built-in support for disconnected graphs. For the Tree and Hierarchical layouts, the result can be different from the result of the generic mechanism (the layout of connected components feature) provided by the base class `IlvGraphLayout`. Depending your particular needs, you can use either the generic mechanism or the built-in support.

Saving layout parameters and preferred layouts

Explains how to save a graph and its layout parameters and preferred layouts in a file.

In this section

Overview of saving

Important: To understand saving better, read section *Using the Graph Model* first.

There are many ways to store your graph and your layout parameters:

♦ Diagram components use XML files for the data and CSS files for the rendering parameters.

♦ Diagram components can use a database.

♦ Graphers can be stored in `.ivl` files.

This topic deals only with `.ivl` files. It is not relevant for applications that use XML files, CSS files, or databases.

Layout parameters are stored in the layout classes directly, not in the `IlvGrapher`. The advantage of this is that the layout parameters are independent of the graph model. However, the disadvantage is that a layout parameter setting is lost whenever a graph is saved to an `.ivl` file and reloaded later. To overcome this disadvantage, the `IlvGrapherAdapter` class allows you to transfer the graph layout settings to instances of `IlvNamedProperty` that can be stored in `.ivl` files and to recover the settings from these named properties at a later time. (See *Save parameters to named properties*.)

The following method indicates whether a layout class supports this mechanism:

```
supportsSaveParametersToNamedProperties();
```

It returns `true` if the layout class can transfer the parameter settings to named properties.

Saving layout parameters to .ivl files

The following example shows how to save the `IlvGrapher` including all layout parameter settings into an `.ivl` file. The example assumes that an `IlvGrapher` is attached to three instances `layout1`, `layout2` and `layout3`.

```
IlvGrapherAdapter adapter1 = (IlvGrapherAdapter)layout1.getGraphModel();
IlvGrapherAdapter adapter2 = (IlvGrapherAdapter)layout2.getGraphModel();
IlvGrapherAdapter adapter3 = (IlvGrapherAdapter)layout3.getGraphModel();
// transfer the layout parameters to named properties
if (layout1.supportsSaveParametersToNamedProperties())
  adapter1.saveParametersToNamedProperties(layout1, false);
if (layout2.supportsSaveParametersToNamedProperties())
  adapter2.saveParametersToNamedProperties(layout2, false);
if (layout3.supportsSaveParametersToNamedProperties())
  adapter3.saveParametersToNamedProperties(layout3, false);
// assume that adapter1, adapter2 and adapter3 work on the same grapher
// save the grapher with all 3 sets of named layout properties to file
grapher.write("abcd.ivl");
// remove the named layout properties because they are no longer needed
adapter1.removeParametersFromNamedProperties();
adapter2.removeParametersFromNamedProperties();
adapter3.removeParametersFromNamedProperties();
```

In this example, *different* grapher adapters of the same `IlvGrapher` are supposed to be attached to `layout1`, `layout2` and `layout3`. If, in fact, the *same* grapher adapter (more precisely, grapher adapters that all work on the same set of nodes and links) is attached to all three layout instances, one call of the `removeParametersFromNamedProperties` method at the end is sufficient.

Take a look at the mechanism in detail:

```
String saveParametersToNamedProperties(IlvGraphLayout layout, boolean
withDefaults);
```

This method creates named properties for the input layout and transfers the complete layout parameter settings to these properties. The property name is an automatically generated, unique string that is returned. If the flag `withDefaults` is `false`, the created layout properties are persistent only if they contain a setting that is not the default setting. This means that the default values are not stored in the `.ivl` file and the file has a smaller size. If the flag `withDefaults` is `true`, the created properties are always persistent, that is, the default values are stored in the `.ivl` file as well.

The named properties that are created require additional memory. Therefore, it is recommended to remove them as soon as they are no longer needed. To remove the named layout properties, you can use one of the following methods:

♦ `removeParametersFromNamedProperties()`

  This method removes all named layout properties from the grapher.

♦ `removeParametersFromNamedProperties(java.lang.String)`

This method removes only the named layout properties that match the input property name.

- `removeParametersFromNamedProperties(java.lang.Class)`

  This method removes all named layout properties that fit the input layout class.

The named layout properties are subclasses of `IlvGraphLayoutGrapherProperty`, `IlvGraphLayoutNodeProperty`, and `IlvGraphLayoutLinkProperty`. See the Java™ *API Reference Manual* for details of these classes.

Saving preferred layouts to .ivl files

The class `IlvDefaultLayoutProvider` allows you to specify the layout instances to be used for each graph. The layout provider can then be used for the recursive layout of a nested graph. (For details, see *Nested layouts*.)

You can set your preferred layouts using one of the methods `setPreferredLayout` of the class `IlvDefaultLayoutProvider` and save these preferred layouts to `.ivl` files using the method:

```
boolean savePreferredLayoutsToNamedProperties(IlvDefaultLayoutProvider
provider, boolean withParameters, boolean withDefaults, boolean traverse);
```

The method takes the Layout Provider as an argument and several flags:

- `withParameters`: if the flag is `true`, the parameters of the preferred layout instances are saved. In this case it is not necessary to use the method `saveParametersToNamedProperties`. Otherwise, only the name of the class of the preferred layout is saved, without its parameters. Therefore, after loading the `.ivl` file, the preferred layout will have, in this case, the default values for all its parameters.
- `withDefaults`: the flag has the same meaning as for the method `saveParametersToNamedProperties`.
- `traverse`: if the flag is `true`, the method applies recursively to the subgraphs. Otherwise, the method saves only the preferred layout of the grapher adapter on which the method is called.

The following code example shows how to save the preferred layouts to `.ivl` files.

```
IlvGrapher grapherA = new IlvGrapher();
IlvGrapher grapherB = new IlvGrapher();

// fill the graphers with nodes and links;
// grapherB is added as a subgraph of grapherA
grapherA.addNode(grapherB, false);

// Create the grapher adapter for the topmost graph
IlvGrapherAdapter adapterA = new IlvGrapherAdapter(grapherA);

// Get a grapher adapter for the subgraph
IlvGraphModel adapterB = adapterA.getGraphModel(grapherB);

// create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// specify the preferred layouts for each grapher
// (this automatically attaches the layouts)
provider.setPreferredLayout(adapterA, new IlvTreeLayout());
provider.setPreferredLayout(adapterB, new IlvGridLayout());
...
```

```
// Save the settings to named properties
adapterA.savePreferredLayouts(provider, true, false, true);

// Save the nested grapher to an .ivl file
grapherA.write("abcd.ivl")
```

Loading layout parameters from .ivl files

The following example shows how to load and recover the parameters of the three layout instances when the layout settings are stored in an `.ivl` file:

```
// Read the IVL file. This reads all named properties as well.
grapher.read("abcd.ivl");
IlvGrapherAdapter adapter1 = (IlvGrapherAdapter)layout1.getGraphModel();
IlvGrapherAdapter adapter2 = (IlvGrapherAdapter)layout2.getGraphModel();
IlvGrapherAdapter adapter3 = (IlvGrapherAdapter)layout3.getGraphModel();
// Transfer the parameter settings from the named properties to the layouts.
adapter3.loadParametersFromNamedProperties(layout3);
adapter2.loadParametersFromNamedProperties(layout2);
adapter1.loadParametersFromNamedProperties(layout1);
// just to be sure that no named layout properties remain in the memory
adapter1.removeParametersFromNamedProperties();
adapter2.removeParametersFromNamedProperties();
adapter3.removeParametersFromNamedProperties();
```

When reading an `.ivl` file, you usually do not know how many named layout properties are stored in the file. In the previous example, if there are less than three sets of named layout properties stored in the `.ivl` file, any unsuccessful call of the method `loadParametersFromNamedProperties(ilog.views.graphlayout.IlvGraphLayout)` has simply no effect, that is, it does not change the parameters of the corresponding layout. If there are more than three sets, the final calls of the method `removeParametersFromNamedProperties()` guarantee that no memory is wasted by the remaining unused layout properties. As mentioned in *Saving layout parameters to .ivl files*, only one call to the method `removeParametersFromNamedProperties` is necessary if only one grapher adapter is attached to all three layout instances.

Note: Parameters are loaded in the reverse order with respect to the order in which they are stored. This is not important if all three layout instances are of different classes because a layout automatically loads only parameters that fit the layout class. However, if, for example, all three layouts are instances of `IlvTreeLayout`, the last saved set of named properties for any Tree Layout is the first set of named properties that is loaded for a Tree Layout.

To load layout parameters, use one of the following methods:

♦ `loadParametersFromNamedProperties(ilog.views.graphlayout.IlvGraphLayout)`

  This method transfers a set of layout parameters from the named layout properties to the input layout. It automatically determines which layout properties fit the input layout. If several sets fit, it transfers the set that was stored last, removes this set of named layout properties from the grapher, and returns `true`. If no set fits and loading cannot be done, it returns `false`.

♦ `loadParametersFromNamedProperties(ilog.views.graphlayout.IlvGraphLayout, java.lang.String)`

This method transfers a set of layout parameters from the layout properties having the input property name to the input layout. It returns `true` if successful, and `false` otherwise.

♦ `loadParametersFromNamedProperties(ilog.views.graphlayout.IlvGraphLayout, java.lang.String)`

  This method transfers a set of layout parameters from the layout properties having the input property name to a newly created instance of `IlvGraphLayout`. It returns the new instance. It returns `null` if no set of layout properties with the input name is found.

Loading preferred layouts from .ivl files

The settings of the preferred layouts can be loaded from `.ivl` files using the method:

```
boolean loadPreferredLayoutsFromNamedProperties(IlvDefaultLayoutProvider
provider, boolean withParameters, boolean traverse);
```

The method takes the Layout Provider as an argument and several flags:

- `withParameters`: if the flag is `true`, the parameters of the preferred layout instances are loaded. In this case it is not necessary to use the method `loadParametersFromNamedProperties`. Otherwise, after loading the `.ivl` file, the preferred layout will have, in this case, the default values for all its parameters.

- `traverse`: if the flag is `true`, the method applies recursively to the subgraphs. Otherwise, the method loads only the preferred layout of the grapher adapter on which the method is called.

This method reads the named properties stored in the file and sets the preferred layout (if any has been stored) on the Layout Provider, using its method `setPreferredLayout`.

The following example shows how to load and recover the preferred layouts (with their parameters) when the preferred layout settings are stored in an `.ivl` file:

```
IlvGrapher grapher = new IlvGrapher();

// Create the grapher adapter for the topmost graph
IlvGrapherAdapter adapter = new IlvGrapherAdapter(grapher);

// Load the graphers from the .ivl file
grapher.read("abcd.ivl");

// Create the layout provider
IlvDefaultLayoutProvider provider = new IlvDefaultLayoutProvider();

// Load the preferred layouts into the provider
// (layout parameters are also read)
adapter.loadPreferredLayoutsFromNamedProperties(provider, true, true);

// Now the provider can be used to perform the recursive layout

adapter.performLayout(provider, true, true, true);

// Detach the layouts when the provider is no longer needed
provider.detachLayouts(adapter, true);
// Dispose the topmost adapter when no longer needed
adapter.dispose();
```

Additional information for expert users

Interface parameters

Some layout classes allow an interface as input parameter. For instance, the method `setNodeBoxInterface` sets an `IlvNodeBoxInterface` (see `IlvLinkLayout`, `IlvShortLinkLayout`, and so on). If an application uses a node box class that implements the node box interface, this can only be stored to an `.ivl` file if the node box class also implements the `IlvPersistentObject` interface. Otherwise, the node box class is not saved to the `.ivl` file.

Compatibility issues

An `.ivl` file that contains layout properties can be loaded only when the package `ilog.views.graphlayout` and its subpackages are available.

Defining your own type of layout

If you develop your own layout algorithms by subclassing `IlvGraphLayout` and want to save the layout parameters of your own layout algorithm in `.ivl` files, you should subclass `IlvGraphLayoutGrapherProperty`, `IlvGraphLayoutNodeProperty`, and `IlvGraphLayoutLinkProperty`. (See *Defining your own type of layout*.) You can find example code by referring to these classes in the Java™ *API Reference Manual*. You should also override the following methods of `IlvGraphLayout` to return your subclasses:

```
protected IlvGraphLayoutGrapherProperty createLayoutGrapherProperty(
    String name, boolean withDefaults)
{
    return new MyOwnLayoutGrapherProperty(name, this, withDefaults);
}
protected IlvGraphLayoutNodeProperty createLayoutNodeProperty(
    String name, IlvGraphic node, boolean withDefaults)
{
    return new MyOwnLayoutNodeProperty(name, this, node, withDefaults);
}
protected IlvGraphLayoutLinkProperty createLayoutLinkProperty(
    String name, IlvGraphic link, boolean withDefaults)
{
    return new MyOwnLayoutLinkProperty(name, this, link, withDefaults);
}
```

Further applications of layout properties

The layout properties can be serialized. If you prefer to use the standard serialization mechanism of Java instead of `.ivl` files, it is recommended to use the layout properties as well because it guarantees that only the parameters are serialized and not any temporary data that may exist in the layout instance.

The following code shows an easy way to copy the parameter setting from `layout1` to `layout2`:

```
IlvGrapherAdapter adapter1 = (IlvGrapherAdapter)layout1.getGraphModel();
IlvGrapherAdapter adapter2 = (IlvGrapherAdapter)layout2.getGraphModel();
adapter1.saveParametersToNamedProperties(layout1, true);
adapter2.loadParametersFromNamedProperties(layout2);
```

The following code shows an easy way to undo temporary changes of layout parameters:

```
IlvGrapherAdapter adapter = (IlvGrapherAdapter)layout.getGraphModel();
adapter.saveParametersToNamedProperties(layout, true);
... change layout parameters
... work with changed parameters
// restore the layout parameters as they were before
adapter.loadParametersFromNamedProperties(layout);
```

Using filtering to lay out a part of an IlvGrapher

Filter support

Important: To understand this topic better, read the topic *Using the Graph Model* first.

Applications sometimes need to perform a layout algorithm on a subset of the nodes and links of a graph. The support for such partial layouts is a filtering mechanism.

Built-in filtering

For applications that use `IlvGrapher`, a filtering feature is built into the `IlvGrapherAdapter` class. To do a partial layout, the `IlvGrapherAdapter` instance needs a way to know which nodes and links to include in the layout. This is the role of the “filter” class, `IlvLayoutGraphicFilter`.

Custom filtering

If the graph is not an `IlvGrapher`, the custom adapter should support the filtering of a graph. (See *Laying out a non-JViews grapher*.) The methods that are related to the structure of the graph (`getNodes`, `getLinks`, `getNeighbors`, and so on as shown in *Information on the Structure of the Graph*) must behave just as if the graph has changed in some way. They must take into account only the nodes and links that belong to the part of the graph that must be laid out.

The graphic filter class

The `IlvLayoutGraphicFilter` class implements the interface `IlvGraphicFilter`, that is, its method:

```
boolean accept(IlvGraphic nodeOrLink)
```

If a filter is specified, the `IlvGrapherAdapter` calls the `accept` method for each node or link whenever necessary. If the method returns `true`, the `IlvGrapherAdapter` considers the node or the link as part of the graph that needs to be laid out. Otherwise, it ignores the node or the link.

To specify a filter on an `IlvGrapherAdapter`, use the following method of the `IlvGrapherAdapter` class:

```
void setFilter(IlvLayoutGraphicFilter filter)
```

To remove the filter, call the `setFilter` method with a `null` argument.

To obtain the filter that has been specified, use the method:

```
IlvLayoutGraphicFilter getFilter()
```

Note: All overridden implementations of the `accept` method must respect the following rule: a link cannot be accepted by the filter if any of its end nodes (origin or destination nodes) are not accepted.

There are two ways to filter an `IlvGrapher`: by layers or by graphic objects. The two methods can be combined.

Filtering by layers

Inside an `IlvGrapher`, nodes and links can be managed by layers. (See the `IlvManagerLayer`class). IBM® ILOG® JViews allows you specify that only nodes and links belonging to certain layers have to be taken into account when performing the layout. Use the following methods of the `IlvGrapherAdapter` class:

```
void addLayer(IlvManagerLayer layer)
```

```
boolean removeLayer(IlvManagerLayer layer)
```

```
boolean removeAllLayers()
```

```
boolean isLayerAdded()
```

To get an enumeration of the manager layers to be taken into account during the layout, use the method:

```
Enumeration getLayers()
```

If no layers have been specified or all the specified layers have been removed, all layers in the `IlvGrapher` are used. In this case, the `getLayers` method returns `null`.

When at least one layer is specified, an `IlvLayoutGraphicFilter` is created internally if it has not already been specified using the `setFilter` method. The default implementation of its `accept` method will automatically check whether a node or a link received as an argument belongs to one of the specified layers.

Filtering by graphic objects

To filter nodes and links individually, you need to write a custom subclass of `IlvLayoutGraphicFilter`.. You must embed the filtering rules in the implementation when you override the `accept` method. For example, an application could use user properties to “mark” nodes and links to be accepted by the filter. The filter class could then be written as follows:

```
public class LayoutFilter
  extends IlvLayoutGraphicFilter
{
  public LayoutFilter()
  {
  }
  public boolean accept(IlvGraphic obj)
  {
    Object prop = obj.getProperty("markedObj");
    if (prop == null)
      return false;
    // accept a link only if its two end-nodes are accepted
    if (obj instanceof IlvLinkImage) {
      IlvLinkImage link = (IlvLinkImage)obj;
      return (link.getFrom().getProperty("markedObj") != null &&
              link.getTo().getProperty("markedObj") != null);
    }
    return true;
  }
}
```

Choosing the layout coordinate space

Describes how to choose the appropriate coordinate space for a layout and how to specify the corresponding mode.

In this section

General considerations about layout and coordinates

Important: To understand this section better, read section *Using the Graph Model* first.

The distinction between zoomable and nonzoomable objects, and the notion of transformer (`IlvTransformer`), are outside the level of the layout framework.

Graph layout algorithms have to deal with the geometry of the graph, that is, the position and shape of the nodes and links. They interact with the geometry of the graph using generic methods of the graph model (`IlvGraphModel`), such as `boundingBox(Object nodeOrLink)`.

The layout algorithms consider the geometry of the graph exactly as it is provided by the graph model. From the point of view of the layout algorithms, the distinction between zoomable and nonzoomable objects is completely transparent. Therefore, when writing a layout algorithm, you do not need to be concerned with such issues.

However, graph layout algorithms must also deal with the layout of an `IlvGrapher`.

The nodes of an `IlvGrapher` object can be any graphic object, that is, any subclass of `IlvGraphic`. The position and size of the nodes are given by their `boundingBox (IlvTransformer t)` method and usually depend on the transformer used for their display. Therefore, when an `IlvGrapher` has to be laid out, the geometry of the grapher must be considered for a given value of the transformer.

Instead of dealing with zoomable/nonzoomable objects and transformers at the level of the layout algorithms, the IBM® ILOG® JViews graph layout package delegates this task to the `IlvGrapherAdapter` object.

Transformers for graphers

Generally speaking, the layout of an IlvGrapher depends on the transformer. The most natural transformer value that could be chosen is the “identity” transformer.

An identity transformer has no translation, zoom, or rotation factors. In terms of IBM® ILOG® JViews, this would mean that the geometry of the `IlvGrapher` would be considered in the manager coordinates, not in the manager view coordinates (transformed coordinates). However, the special case of *nonzoomable* graphic objects must be taken into account. For this case, the idea of simply using the geometry of the grapher in manager coordinates is not pertinent.

Nonzoomable graphic objects as nodes

A graphic object is said to be zoomable if its bounding box follows the zoom level. Otherwise, the object is nonzoomable. (To know whether a graphic object is zoomable, use its boolean zoomable() method, or check its documentation.)

If all the nodes and links of an `IlvGrapher` object are zoomable graphic objects, a layout obtained on the basis of the graph geometry in manager coordinates will look the same for any value of the transformer used for the display. Simply speaking, the drawing of the graph will just be zoomed, or translated.

When at least one nonzoomable graphic object is used as a node in an `IlvGrapher`, the geometry of the grapher in manager coordinates can no longer be used. When drawn with different transformer values (for instance, at different zoom levels), the same `IlvGrapher` can look very different.

When a grapher contains nonzoomable graphic objects, it may not be appropriate to deal with the geometry of the `IlvGrapher` based on the bounding boxes of the graph objects systematically computed for an identity transformer (manager coordinates). To ensure that the drawing of the laid-out graph is always correct, even when nonzoomable graphic objects are present, the transformer used for the display must be considered.

Reference transformer for grapher

The reference transformer is the transformer that is currently being used for the display of the IlvGrapher. The IlvGrapherAdapter may need to compute the geometry of the graph for this transformer.

How a reference transformer is used

For a simple example of how a reference transformer is used, consider the `boundingBox (java.lang.Object)` method. This abstract method of the `IlvGraphModel` class is implemented in the `IlvGrapherAdapter`. To compute the bounding box, it calls the `IlvGrapher` method of the graphic object that it receives as an argument. However, it does not handle zoomable objects and nonzoomable objects in the same way.

If the graphic object is zoomable, the `boundingBox(java.lang.Object)boundingBox(Object nodeOrLink)` method of the `IlvGrapherAdapter` returns the bounding box in manager coordinates by calling `IlvGraphic.boundingBox(null)`.

If the graphic object is nonzoomable, the `boundingBox(java.lang.Object)boundingBox (Object nodeOrLink)` method computes the bounding box according to the reference transformer and returns a rectangle obtained by applying the inverse transformation to this rectangle. (See the `inverse(ilog.views.IlvRect)IlvTransformer.inverse(IlvRect rect)` method.)

The geometry of the `IlvGrapher` is computed in such a manner that the resulting drawing inside an `IlvManagerView` using the reference transformer will look fine.

Reference views

Optionally, an `IlvManagerView` can be specified as a *reference view* for the `IlvGrapherAdapter`. If a reference view is specified, its current transformer (at the moment when the layout is started) is automatically used as the reference transformer. Usually, applications use the same manager view that is used for the display of the `IlvGrapher` as the reference view (but this is not mandatory).

To specify the reference view, use the following method:

```
void setReferenceView(IlvManagerView view)
```

To get the current reference view, use the method:

```
IlvManagerView getReferenceView()
```

If no view has been specified as the reference view, the method returns `null`.

Specifying a reference transformer

You can specify a reference transformer explicitly using the method:

```
void setReferenceTransformer(IlvTransformer transformer)
```

The current reference transformer is returned by the method:

```
IlvTransformer getReferenceTransformer()
```

In most cases, it is not necessary to specify a reference transformer because the last method automatically chooses it according the following rules:

♦ If a reference transformer is specified, the specified transformer is returned.

♦ If a reference view has been specified, the transformer of the reference view is returned.

♦ If the `IlvGrapher` attached to the `IlvGrapherAdapter` has at least one manager view, the transformer of the first manager (as returned by the method `IlvManager.getViews()`) is returned.

The only cases where you may need to specify a reference transformer or a reference view are the following:

♦ The `IlvGrapher` contains nonzoomable objects (that is, the layout cannot be correctly computed independently of the transformer used for drawing the graph) and more than one manager view is attached to the grapher.

♦ The `IlvGrapher` contains nonzoomable objects and you want to perform the layout without attaching a manager view to the grapher. (Therefore, the default rule for choosing the current transformer of the first manager view as the reference transformer cannot be applied.)

If a grapher containing nonzoomable objects is displayed simultaneously in several views, you can use the `setReferenceView(ilog.views.IlvManagerView)` method to indicate the view for which you want the drawing of the graph to be optimal.

If you specified a reference transformer but want to reset this setting and go back to the default behavior, call the method `setReferenceTransformer(ilog.views.IlvTransformer)` with a `null` argument.

Note that if you override the `setReferenceTransformer(ilog.views.IlvTransformer)` method, you must call `super.setReferenceTransformer` to notify the `IlvGrapherAdapter` that the reference transformer has changed.

Note also that a call to the `setReferenceView` method overrides the effect of a call to the `setReferenceTransformer(ilog.views.IlvTransformer)` method. In the same way, a call to the `setReferenceTransformer(ilog.views.IlvTransformer)` method overrides the effect of a call to the `setReferenceView(ilog.views.IlvManagerView)` method.

Specifying the mode for layout coordinates

By default, the `IlvGrapherAdapter` considers the geometry of the nodes and links of an `IlvGrapher` in a special coordinate space which is appropriate for most of the cases. In some situations, it can be useful to specify a different coordinate space.

To specify the coordinate space, the class `IlvGrapherAdapter` provides the following method:

```
void setCoordinatesMode(int mode)
```

The valid values for `mode` are:

- ♦ `IlvGraphLayout.MANAGER_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager (that is, the `IlvGrapher` encapsulated by the adapter) without applying any transformation.

  This mode should be used if you visualize the graph at zoom level 1, or you do not visualize it at all, or the grapher contains only fully zoomable objects. In all these cases there is no need to take the transformer into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in manager coordinates. The reference transformer and the reference view are not used.

- ♦ `IlvGraphLayout.VIEW_COORDINATES`

  The geometry of the graph is computed in the coordinate space of the manager view. More exactly, all the coordinates are transformed using the current reference transformer.

  This mode should be used if you want the dimensional parameters of the layout algorithms to be considered as being specified in manager view coordinates.

- ♦ `IlvGraphLayout.INVERSE_VIEW_COORDINATES`

  The geometry of the graph is computed using the coordinate space of the manager view and then applying the inverse transformation using the reference transformer. This mode is equivalent to the "manager coordinates" mode if the geometry of the graphic objects strictly obeys the transformer. (A small difference may exist because of the limited precision of the computations.)

  On the contrary, if some graphic objects are either nonzoomable or semizoomable (for example, links with a maximum line width), this mode gives different results than the manager coordinates mode. These results are optimal if the grapher is visualized using the same transformer as the one taken into account during the layout.

  Note that in this mode the dimensional parameters of the layout algorithms are considered as being specified in manager coordinates.

The default mode is `IlvGraphLayout.INVERSE_VIEW_COORDINATES`.

To obtain the current choice, use the following method:

```
long getCoordinatesMode()
```

The mode for coordinates can also be specified directly on the layout instances. For details, see *Coordinates mode*.

Defining your own type of layout

Describes how to develop a custom graph layout algoithm if you need one.

In this section

A sample custom layout algorithm

If the layout algorithms provided with IBM® ILOG® JViews Diagrammer do not meet your needs, you can develop your own layout algorithms by subclassing `IlvGraphLayout`.

When a subclass of `IlvGraphLayout` is created, it automatically fits into the generic IBM® ILOG® JViews Diagrammer layout framework and benefits from its infrastructure:

- generic parameters: see *Base class parameters and features*
- notification of progress: see *Using event listeners*
- capability to lay out any graph object using the generic graph model: see *Using the Graph Model*
- capability to apply the layout separately for the connected components of a disconnected graph: see *Laying out connected components of a disconnected graph*
- capability to lay out nested graphs (see *Nested layouts*), and so on.

Example

To illustrate the basic ideas for defining a new layout, the following simple example shows a possible implementation of the simplest layout algorithm, the Random Layout. The new layout class is called `MyRandomLayout`.

The following shows the skeleton of the class:

```
public class MyRandomLayout
  extends IlvGraphLayout
{
  public MyRandomLayout()
  {
  }

  public MyRandomLayout(MyRandomLayout source)
  {
    super.source(source);
  }

  public IlvGraphLayout copy()
  {
    return new MyRandomLayout(this);
  }

  protected void layout(boolean redraw)
  {
    ...
  }
}
```

The constructor with no arguments is empty. The `copy` constructor and the `copy` method are implemented; they are used when laying out a nested graph (see *Nested layouts*).

Then, the abstract method `layout(boolean)` of the base class is implemented as follows:

```
 protected void layout(boolean redraw)
  {
    // obtain the graph model
    IlvGraphModel graphModel = getGraphModel();

    // obtain the layout report
    IlvGraphLayoutReport layoutReport = getLayoutReport();

    // obtain the layout region
    IlvRect rect = getCalcLayoutRegion();
    float xMin = rect.x;
    float yMin = rect.y;
    float xMax = rect.x + rect.width;
    float yMax = rect.y + rect.height;

    // initialize the random generator
    Random random = (isUseSeedValueForRandomGenerator()) ?
        new Random(getSeedValueForRandomGenerator()) :
        new Random();

    // browse the objects in the grapher
    Enumeration nodes = graphModel.getNodes();
    while (nodes.hasMoreElements()) {
        Object node = nodes.nextElement();

        // skip fixed nodes
        if (isPreserveFixedNodes() && isFixed(node)))
            continue;

        // compute coordinates
        float x = xMin + (xMax - xMin) * random.nextFloat();
        float y = yMin + (yMax - yMin) * random.nextFloat();

        // move the node to the computed position
        graphModel.moveNode(node, x, y, redraw);

        // notify listeners on layout events
        callLayoutStepPerformedIfNeeded();
    }

    // set the layout report code
    layoutReport.setCode(IlvGraphLayoutReport.LAYOUT_DONE);
}
...
```

Note that the `layout` method is `protected`, which is the access type of the method in the base class. This will not prevent a user outside the package containing the class from performing the layout because it is started using the public method `performLayout`.

Implementing the layout method

Depending on the characteristics of the layout algorithm, some of the steps required may be different or unnecessary, or other steps may be needed.

Depending on the particular implementation of your layout algorithm, other methods of the `IlvGraphLayout` class may need to be overridden. For instance, if your subclass supports some of the generic parameters of the base class, you must override the `supports [ParameterName]` method (see *Base class parameters and features*). For further information about the class `IlvGraphLayout`, refer to the API reference documentation.

Note: If you want to save the layout parameters of your new layout algorithm in `.ivl` files, you should override the methods `createLayoutGrapherProperty(java.lang.String, boolean)`, `createLayoutNodeProperty(java.lang.String, ilog.views.IlvGraphic, boolean)`, `createLayoutLinkProperty(java.lang.String, ilog.views.IlvGraphic, boolean)`, and subclass `IlvGraphLayoutGrapherProperty`, `IlvGraphLayoutNodeProperty`, and `IlvGraphLayoutLinkProperty`. See *Saving layout parameters and preferred layouts* for more explanation.

To implement the `layout` method in the sample custom layout algorithm:

1. Obtain the graph model (`getGraphModel()` on the layout instance).

```
IlvGraphModel graphModel = getGraphModel();
```

2. Obtain the instance of the layout report that is automatically created when the `performLayout` method from the superclass is called (`getLayoutReport()` on the layout instance). See *Using a graph layout report*.

```
IlvGraphLayoutReport layoutReport = getLayoutReport();
```

3. Obtain the layout region parameter to compute the area where the nodes will be placed.

```
IlvRect rect = getCalcLayoutRegion();
```

4. Initialize the random generator.

```
Random random = (isUseSeedValueForRandomGenerator()) ?
        new Random(getSeedValueForRandomGenerator()) :
        new Random();
```

   (For information on the seed value parameter, see *Random generator seed value*.)

5. Get an enumeration of the nodes (`getNodes()` on the graph model instance).

```
Enumeration nodes = graphModel.getNodes();
```

6. Browse the nodes, skipping fixed nodes (`isFixed(node)` on the layout instance) if asked by the user (`isPreserveFixedNodes()` on the layout instance).

```
while (nodes.hasMoreElements()) {
   Object node = nodes.nextElement();
...
```

(For details on fixed nodes, see *Preserve fixed nodes*).

7. Move each node to the newly computed coordinates inside the layout region (`graphModel.moveNode`).

```
graphModel.moveNode(node, x, y, redraw);
```

8. Notify the listeners on layout events that a new node was positioned (`callLayoutStepPerformedIfNeeded()` on the layout instance). This allows the user to implement, for example, a progress bar if a layout event listener was registered on the layout instance.

```
callLayoutStepPerformedIfNeeded();
```

(For details on event listeners, see *Using event listeners*.)

9. Finally, set the code in the layout report.

```
layoutReport.setCode(IlvGraphLayoutReport.LAYOUT_DONE);
```

Once you have implemented your own layout algorithm `MyRandomLayout`, you can use it directly in Java™ .

Once you have implemented your own layout algorithm `MyRandomLayout`, you can add it to a CSS file to use it in a diagram component. Since your new layout algorithm is not one of the predefined graph layout algorithms, you need to specify it as fully qualified in CSS:

```
SMD {
    GraphLayout: true;
}
GraphLayout {
    graphLayout: "mypackage.MyRandomLayout";
    // ... additionally, any Bean property of MyRandomLayout can
    // be specified here ...
}
```

FAQs about using the layout algorithms

The following list of FAQs provides some helpful suggestions for using the layout algorithms. You may find some answers to questions that come up when using the graph layout package.

FAQs about the layout algorithms

Question	Answer
I perform the layout and nothing happens (no node is moved). Why?	One possible reason may be: the layout algorithms provided in IBM® ILOG® JViews Diagrammer are all designed to do nothing, by default, if no change occurred in the graph since the last time the layout was performed successfully on the same graph. A change means that a node was moved, or a node or link was added, removed, or reshaped. Note that you can force the layout to be performed again, even if no change occurred, by calling the `performLayout(boolean, boolean)` method with a `true` value for the force argument. (In the Composer demonstration, you can choose this option in the Options menu.) Another possible reason may be: an error or a special case occurred during the layout. First, you should check whether the `performLayout()` method has thrown an exception. If no exception was thrown, call the `getCode()` method on the instance of the layout report returned by the `performLayout` method. Check this value with respect to the documentation of the appropriate layout report class. (For details, see *Using a graph layout report*.)
With the Uniform Length Edges algorithm, after having performed the layout once, I don’t see any movement even if I use the force layout option. Why?	The reason is probably that the first time you performed the layout, the algorithm reached the convergence. When the layout is performed again, it detects that the convergence has been already reached and stops. If you really want to continue working, for instance in order to “declutter” a particular part of the graph, you may need to move one or several nodes in order to change the initial configuration. (The algorithm is dependent on the initial configuration.)
After performing the layout, the graph is laid out far from its initial position. Why?	Most of the layout algorithms use a layout region parameter to control the size and position of the layout. (For details, see *Layout region*.) Depending on the value of this parameter, the nodes may be moved far from their initial positions. To know whether a layout algorithm is designed to use a layout region parameter, check the documentation to see if the layout class overrides the `supportsLayoutRegion()` method of the base class in order to return `true`. Other algorithms have a different mechanism that allows you to specify the desired location of the layout. It may happen that the default value of the location parameter is such that the graph is laid out far from its initial position.
When I use certain layout algorithms on certain graphs, there are overlapping nodes. Why and what can I do?	One possible reason may be related to the different ways layout algorithms deal with the size of the nodes: -The Topological Mesh algorithm is not able to explicitly take into account the size of the nodes.

Question	Answer
	- The Tree, Hierarchical, Bus, and Grid algorithms always avoid overlapping nodes. (The Link algorithm does not move the nodes. It only reshapes the links such that the crossings and overlaps are reduced. The size of the nodes is taken into account.) - The Uniform Length Edges algorithm (with the option "Respect Node Sizes" enabled) and the Circular algorithm, in many cases, succeed in avoiding overlapping nodes. In any case, if the layout algorithm supports the layout region mechanism (see *Layout region*), you should try to increase the size of the layout region. For example, if your graph contains hundreds of nodes, it is not reasonable to use a small layout region, such as 600x600. There will be not enough space for all the nodes. You should try a larger layout region, for example 5000x5000. The optimal size of the layout region depends, of course, not only on the number of nodes, but also on their size. If the nodes are relatively large with respect to the size of the layout region, it may be necessary to adjust some of the parameters (for instance, the preferred link length for the Uniform Length Edges Layout).
In some networks, there are two (or more) subnetworks that are not connected. How will this affect the layout algorithms?	This depends on the layout class you use: - `IlvTopologicalMeshLayout`: It will work on the connected component of the graph that contains the starting node. (You can specify this node as a parameter.) If the "starting node" is not specified, it is automatically chosen in an arbitrary way. The nodes of the other "connected components" will not be moved. You may want to perform the layout separately on each connected component using different layout regions and starting node settings. This is what you get automatically when you enable the "layout of connected components" parameter. (See *Layout of connected components*.) - `IlvUniformLengthEdges`: This algorithm supports disconnected graphs, but usually it is better to rely on the automatic "layout of connected components" parameter. (See *Layout of connected components*.) - `IlvBusLayout`: It will work on the "connected component" of the graph that contains the "bus object." (You must specify the bus object as a parameter.) The other nodes that are not connected to the bus will not be moved. You may need to perform the layout separately on each connected component. This is what you get automatically when you enable the "layout of connected components" parameter. (See *Layout of connected components*.) - `IlvCircularLayout`, `IlvHierarchicalLayout`, `IlvTreeLayout`: They have built-in support for disconnected graphs. Alternatively, you can use the automatic support from the base class. (See *Layout of connected components*.)

Question	Answer
	- `IlvLinkLayout`, `IlvGridLayout`, `IlvRandomLayout`: These algorithms support both connected and disconnected graphs. Their behavior is the same for both categories of graphs.
There are some attributes of the network that we know about (for instance, we know what the core switch is and what the center should be). Are such attributes taken into account by the layout algorithm?	It depends on the layout algorithm. - The Circular Layout is designed to allow you to specify information about the physical topology of the network. You can specify which nodes belong to the same cluster (ring or star), the order of the nodes on the cluster, and which node is the center of a star cluster. - In the Tree Layout, you can specify the root node. - In the Bus Layout algorithm, you can specify the bus object. - In the Hierarchical Layout algorithm, you can specify node position indices and level indices, as well as relative positioning constraints.
If I use IBM® ILOG® JViews Diagrammer on different computers or with different Java™ Virtual Machines (JVM™) or both, I sometimes get different layouts for the same graph and with the same parameters. Why?	There are two possible reasons: 1. Different computers and JVMs may be slower or faster. If the layout algorithm you use stops the computation when the specified allowed time has elapsed, a slower computer or JVM will cause the computation to stop earlier. That may be the cause of different results. This may happen even with the same computer and JVM if the charge of the computer is increased. You may need to increase the allowed time specification when running on a slower computer or JVM. 2. If you use a layout algorithm that uses the random generator and if you use the default option for the seed value (that is, the system clock is used), you get different results for each successive run of the layout on the same graph. This allows you to obtain alternative results and to chose the one you prefer. If you want to prevent different results for successive runs, you can specify a constant seed value.
I use the Link Layout algorithm to lay out the links (representing routes) of a network of graphical objects (towns) geo-positioned on a cartographical map. When several links connect to the same side of a node, they overlap, while I expect them to respect the “link offset” (or the “grid size”) parameter of the Link Layout. Why?	Some dimensional parameters of the layout algorithms need to be chosen with respect to the size of the nodes. This is the case of the “link offset” and the “bypass distance” parameters for the Short Link Layout and the grid size for the Long Link Layout. Indeed, their default values are not appropriate when the nodes are very large. Often, nodes placed on a map, for instance a world map, have a very large size. Compared to this size, the default values of the parameters are so small that they appear to be zero. The solution is to increase the values of the dimensional parameters, taking into account the size of the nodes. If different nodes have different sizes, either the medium or the largest size of the nodes can be used to compute the parameters as a fraction of this size.

Releasing resources used during the layout of a grapher

Various objects need to be created during the layout process. Most commonly, these are:

- ♦ Layout instances (subclasses of `IlvGraphLayout`)
- ♦ Grapher adapters (subclasses of `IlvGrapherAdapter`)
- ♦ Other adapters (subclasses of `IlvGraphModel`)
- ♦ Layout providers.

  For recursive layout, you may also instantiate layout providers (subclasses of `IlvDefaultLayoutProvider`). See also *Recursive layout*.
- ♦ Property objects

  Some of the layout parameters are internally stored as property objects attached to the grapher object or to its nodes and links.

If you use a diagram component (a subclass of `IlvDiagrammer`) with styling, all created objects are automatically released when they are no longer used so that obsolete objects can be garbage-collected and memory leaks (in the Java™ sense) are avoided.

Rules for releasing resources

If you program graph layout directly in Java, you must respect some rules to ensure that all these allocated objects are correctly released:

1. When a layout instance instantiated by your code is no longer useful, call the method `detach()` on it to ensure that no grapher or graph model is still attached to it. Note that you can freely reuse a layout instance once the previously attached model has been detached.
2. Layout parameters that are specific to a node or a link are cleaned when calling `IlvGraphLayout.detach()`. This cleaning is done only for nodes and links that are still in the grapher when the `detach()` method is called. If per-node or per-link parameters have been specified and the node or the link needs to be removed before the `detach()` method can be called, you can call the methods `cleanNode` or `cleanLink` of the class `IlvGraphLayout` to perform the cleaning for the node or the link. However, you only need to do so if the removed node or link is reused by your code after removal. Otherwise, if your code does not keep any reference to it, the node or link will be garbage collected anyway, together with the property objects eventually stored by the layout.
3. When a grapher adapter (or other graph models) instantiated by your code is no longer useful, call the method `dispose()` on it to ensure that the resources it has used are released. Note that an adapter (or graph model) must not be used once it has been disposed.
4. When a layout provider (an instance of `IlvDefaultLayoutProvider`) instantiated by your code is no longer useful, call the method `detachLayouts(model, true)` on it, passing as arguments the graph models that have been used for performing a recursive layout with this provider.

Using graph layout Beans

Shows you how to use IBM® ILOG® JViews Framework Beans and graph layout Beans when you create an applet within an Integrated Development Environment (IDE).

In this section

Overview

Before you can create an application or applet within an Integrated Development Environment (IDE), you must install the IBM® ILOG® JViews Framework Beans into your IDE. Detailed installation instructions are given in Installing IBM® ILOG® JViews Beans in an IDE in *The Essential JViews Framework*.

Graph layout classes available as Beans

The following classes are provided as graph layout Beans:

`IlvBusLayout` displays bus network topologies, that is, a set of nodes connected to a bus node.

`IlvCircularLayout` displays graphs representing interconnected ring and/or star network topologies.

I `IlvGridLayout` arranges disconnected nodes in rows or in columns or on a grid.

I `IlvHierarchicalLayout` arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.

I `IlvLinkLayout` reshapes the links of a graph without moving the nodes.

I `IlvRandomLayout` moves the nodes of the graphs at randomly computed positions inside a user-defined area.

`IlvTopologicalMeshLayout` can be used to lay out cyclic graphs.

`IlvTreeLayout` arranges the nodes of a tree horizontally or vertically, starting from the root of the tree.

`IlvUniformLengthEdgesLayout` can be used to lay out any type of graph and allows you to specify a preferred length of the links.

`IlvJGraphLayoutProgressBar` a Swing JProgressBar toolbar that automatically displays the progress of the layout process.

The Beans listed in this topic are classes of the graph layout API.

If you want to use graph layout integrated in a diagram component, it is better to use IBM® ILOG® JViews Diagrammer Beans. See JViews Diagrammer classes available as beans in *Using the Designer*.

Creating a simple applet using Beans

This topic explains how to:

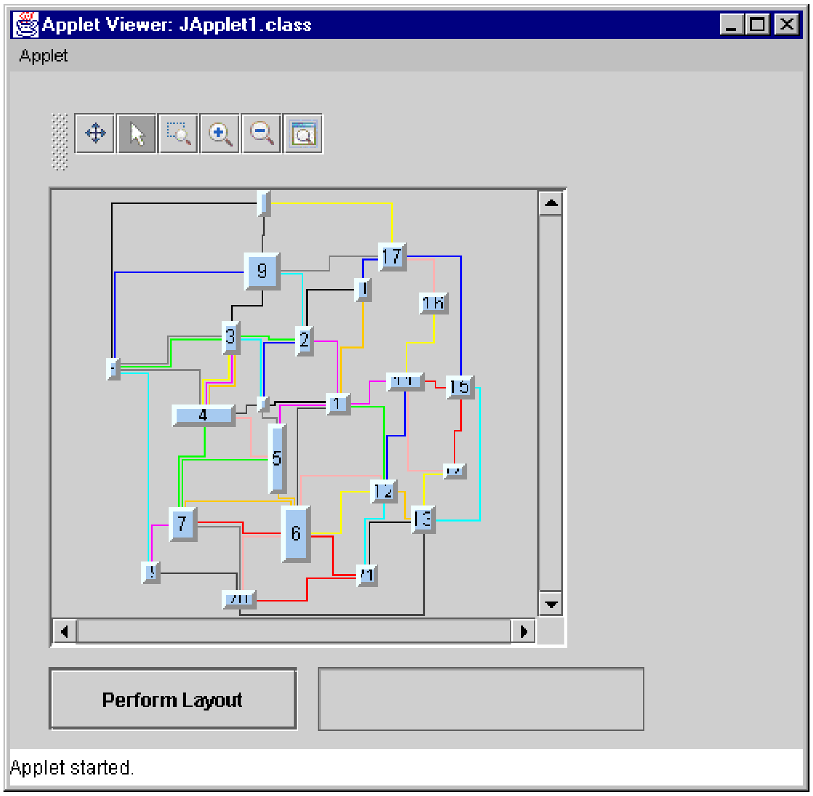
- ♦ Create an IBM® ILOG® JViews applet using IBM® ILOG® JViews Framework Beans
- ♦ Add graph layout features using the graph layout Beans

The sample applet is a simple JFC/Swing applet that displays a graph and provides graph layout capabilities. No coding is necessary.

For more information on IBM® ILOG® JViews Framework Beans, see Framework classes available as JavaBeans(TM)

in *The Essential JViews Framework*

The following figure shows the application panel created in this tutorial.



Final JFC/Swing applet

This example shows how to create an IBM® ILOG® JViews applet using a typical IDE procedure, which consists of:

1. *Creating the manager view*

2. *Setting the properties of the manager view*
3. *Creating a grapher and associating a manager view*
4. *Loading an .ivl file into the grapher*
5. *Adding a control toolbar Bean*
6. *Adding a graph layout Bean*
7. *Adding a Swing Bean*
8. *Adding user interaction*
9. *Adding a progress bar to the Applet*
10. *Testing the result*

For information on the concepts that underlie JavaBeans™ , refer to the Web site: *http://java.sun.com/products/javabeans*

You are assumed to be familiar with the manipulation of JavaBeans within your IDE.

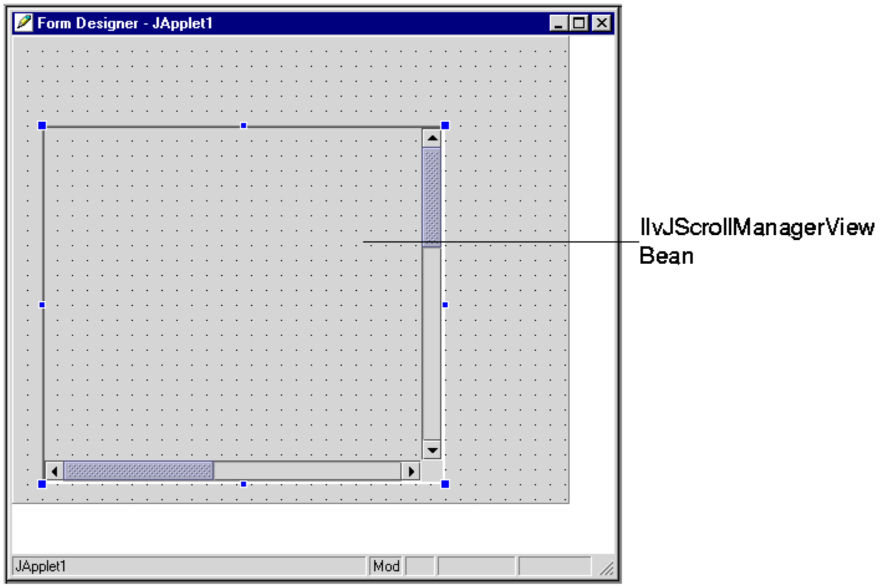
Creating the manager view

To create the manager view:

1. Create a new project as a JFC/Swing applet or application.
2. On the Beans toolbar, click the JViews tab to display the JViews Framework Beans.
3. On the IBM® ILOG® JViews Framework Beans toolbar, click the `IlvJScrollManagerView` Bean icon and drag it to the Form Designer.

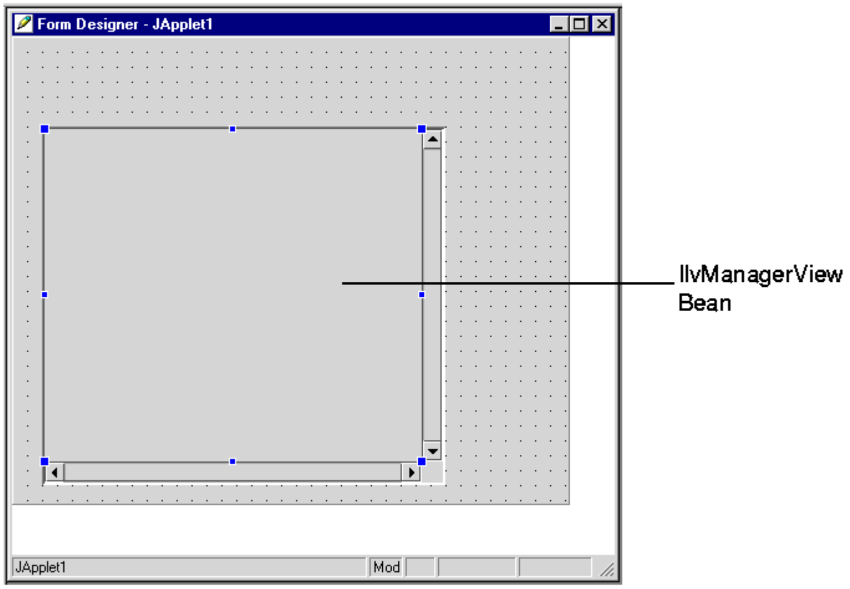
Note:
1. You will notice two scroll manager view icons on the toolbar. Place the pointer over the icon to read the name and choose the one with "J" in its prefix, `IlvJScrollManagerView`.
2. IBM® ILOG® JViews Framework Swing Beans have the letter "J" in the prefix of the Bean name. You could also create the same type of application using only AWT controls. To do so, you would simply use the `IlvScrollManagerView` Bean that is an AWT control instead of the `IlvJScrollManagerView` Bean. However, there is no AWT equivalent of the `IlvJGraphLayoutProgressBar` Bean

4. Drag the handles of the `IlvJScrollManagerView` Bean until it looks approximately like this:



5. On the JViews Beans toolbar, click the `IlvManagerView` icon and drag it inside the `IlvJScrollManagerView` Bean.

   The result is fairly similar to what you obtained in the previous step, except that you can now select the manager view.



Note: If you were to compile and run the project at this point, you would see that the `IlvJScrollManagerView` allows you to scroll through the contents of the `IlvManagerView` Bean. (At the moment, the manager view is empty, so there is nothing to scroll.)

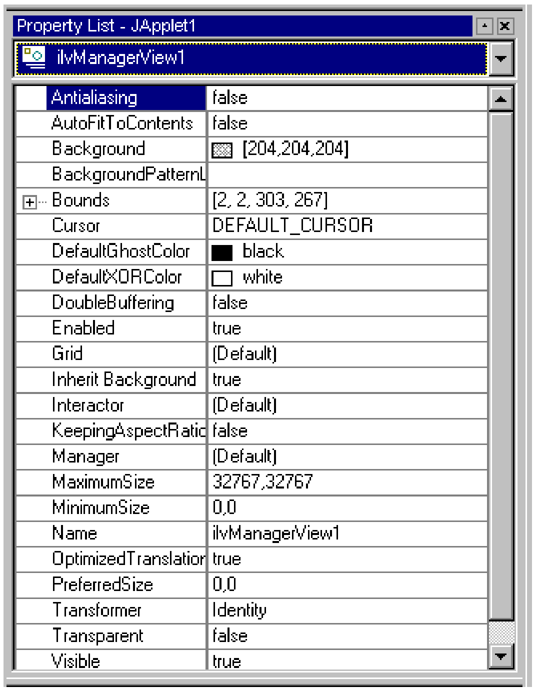
Setting the properties of the manager view

The next stage involves changing two properties of the `IlvManagerView` Bean to make sure that the double-buffering mechanism will be used and that the zoom level of the manager views will always remain the same along the x-axis and y-axis.

To change the manager view properties:

1. Make sure that the `IlvManagerView` Bean is selected in the Form Designer.

   When the `IlvManagerView` Bean is selected in the Form Designer, its properties are displayed in the Property List.



Property List - JApplet1

ilvManagerView1

Antialiasing	false
AutoFitToContents	false
Background	▨ [204,204,204]
BackgroundPatternL	
⊞ Bounds	[2, 2, 303, 267]
Cursor	DEFAULT_CURSOR
DefaultGhostColor	■ black
DefaultXORColor	□ white
DoubleBuffering	false
Enabled	true
Grid	(Default)
Inherit Background	true
Interactor	(Default)
KeepingAspectRatic	false
Manager	(Default)
MaximumSize	32767,32767
MinimumSize	0,0
Name	ilvManagerView1
OptimizedTranslatior	true
PreferredSize	0,0
Transformer	Identity
Transparent	false
Visible	true

2. Click the value field of the `DoubleBuffering` property. Change the value from `false` to `true`. This will ensure that the double-buffering mechanism will be used.

3. Click the value field of the `KeepingAspectRatio` property. Change the value from `false` to `true`. This will ensure that the zoom level remains the same along the x-axis and y-axis.

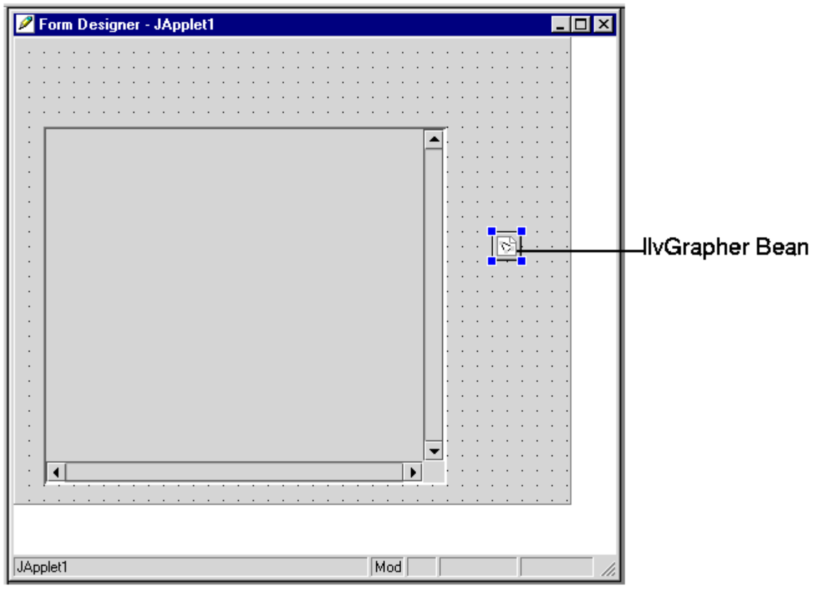
Creating a grapher and associating a manager view

This stage involves creating an `IlvGrapher` Bean and associate a manager view with it. The `IlvGrapher` Bean provides the data structure that contains the grapher to display.

To create the `IlvGrapher` Bean and associate a manager view with the grapher:

1. On the IBM® ILOG® JViews Framework Beans toolbar, click the `IlvGrapher` icon and drag it into the Form Designer.

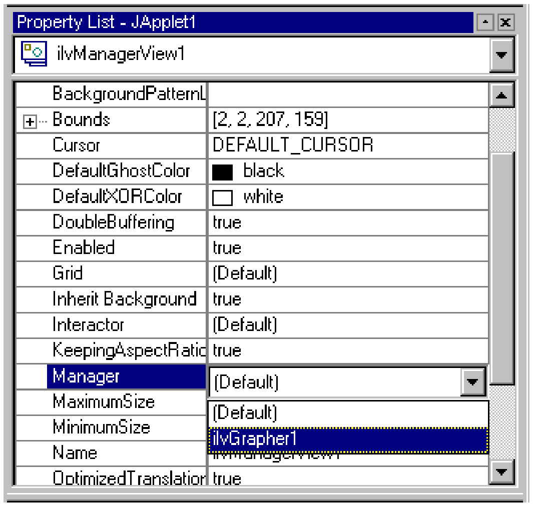
   The `IlvGrapher` class is not a graphical Bean, so it is not managed in the same way by the various IDEs. The following figure shows the grapher as a small object inside the Form Designer.



2. To associate the manager view with the grapher, select the `IlvManagerView` Bean so that its properties appear in the Property List.

   You now need to set the `Manager` property of the `IlvManagerView` Bean to the new `IlvGrapher` Bean. (Keep in mind that the `IlvGrapher` class is a subclass of `IlvManager`.)

3. In the Property List window, click the value column of the `Manager` property. Change `Default` to `ilvGrapher1`.



BackgroundPatternL	
Bounds	[2, 2, 207, 159]
Cursor	DEFAULT_CURSOR
DefaultGhostColor	■ black
DefaultXORColor	□ white
DoubleBuffering	true
Enabled	true
Grid	(Default)
Inherit Background	true
Interactor	(Default)
KeepingAspectRatio	true
Manager	(Default)
MaximumSize	(Default)
MinimumSize	ilvGrapher1
Name	ilvManagerView1
OptimizedTranslation	true

Once this operation is done, the IlvManagerView can display the contents of the IlvGrapher Bean. You can create several IlvManagerView Beans and associate them with the same IlvGrapher Bean. This allows you to have several views of the same graph.

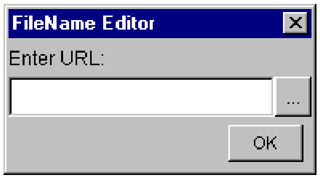
Loading an .ivl file into the grapher

The next stage involves loading an .ivl file into the IlvGrapher Bean so that the contents of the .ivl file are displayed in the manager view.

To load an .ivl file, do the following:

1. In the Form Designer, select the IlvGrapher Bean so that its properties appear in the Property List.
2. Click the value field of the FileName property and then click the ... button.

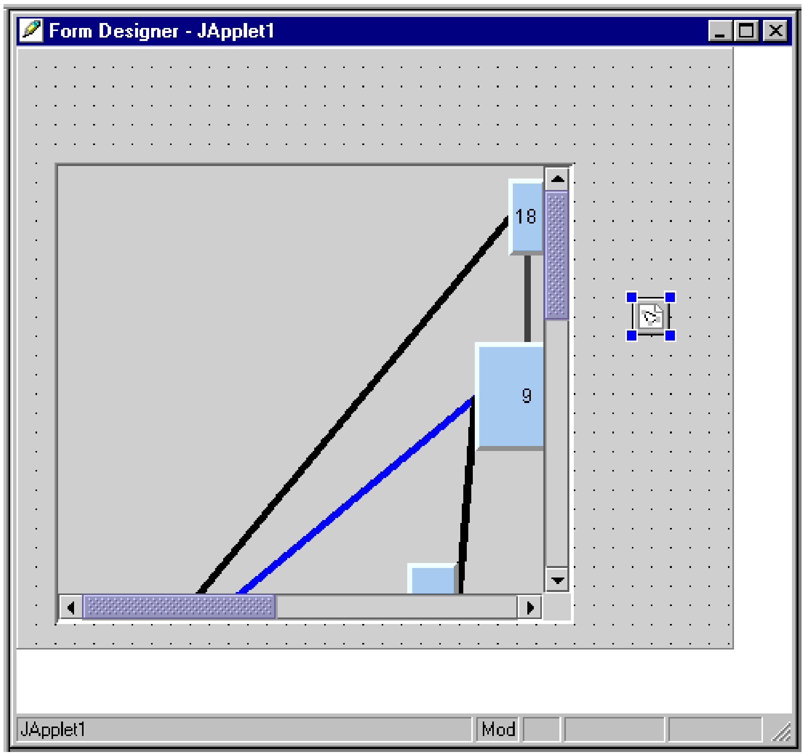
   The FileName Editor dialog box appears.

3. To specify the .ilv file, click the ... button in the FileName Editor.

   The Choose URL dialog box appears.
4. Go to the <installdir>/jviews-diagrammer86/data/graphlayout/link directory.
5. Select the sample1.ivl file and click Open.
6. Click OK in the FileName Editor dialog box.

   The file is automatically displayed in the IlvManagerView Bean. As you can see, only a portion of the graph is visible in the manager view.

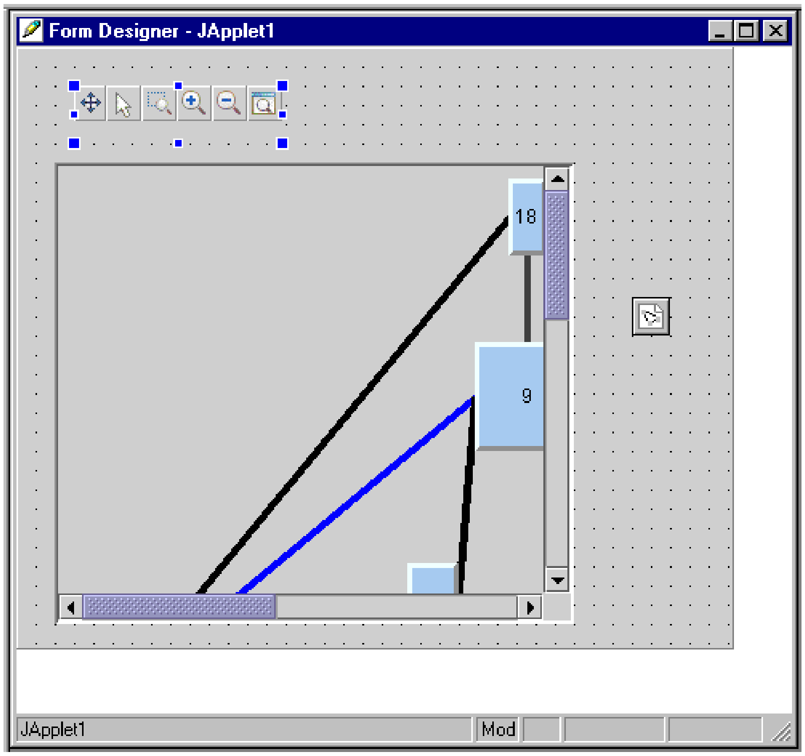


Adding a control toolbar Bean

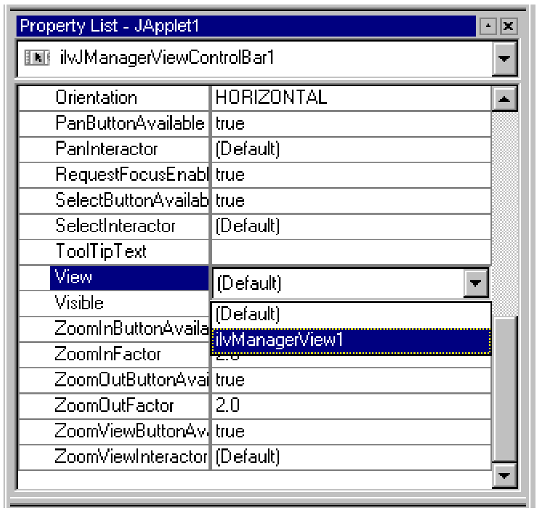
This stage involves adding a toolbar and associating the toolbar with the manager view. The toolbar allows the user to control the zoom level of the view and to pan the view.

To add the toolbar:

1. On the IBM® ILOG® JViews Framework Beans toolbar, click the `IlvJManagerViewControlBar` icon and drag it to the Form Designer.



2. Make sure that the `IlvJManagerViewControlBar` Bean is selected in the Form Designer so that its properties appear in the Property List.
3. To associate the toolbar with the manager view, click the value field of the `View` property and select `ilvManagerView1`.


Property List - JApplet1

ilvJManagerViewControlBar1

Orientation	HORIZONTAL
PanButtonAvailable	true
PanInteractor	(Default)
RequestFocusEnabl	true
SelectButtonAvailab	true
SelectInteractor	(Default)
ToolTipText	
View	(Default)
Visible	
ZoomInButtonAvaila	
ZoomInFactor	
ZoomOutButtonAvai	true
ZoomOutFactor	2.0
ZoomViewButtonAv	true
ZoomViewInteractor	(Default)

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

Adding a graph layout Bean

This stage involves adding the capability to apply a link layout to the graph displayed in the applet.

To add the required graph layout Bean:

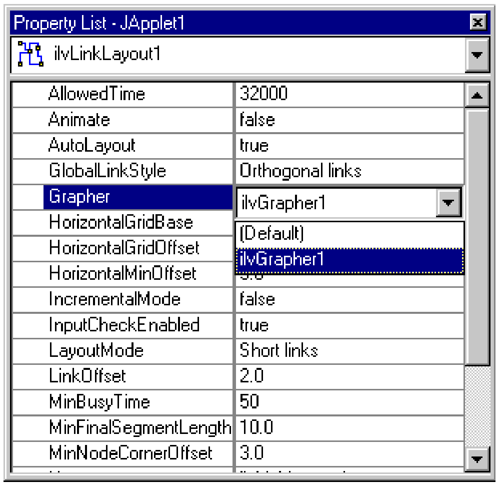
1. On the Beans toolbar, click the JViews graph layout tab to display the JViews Graph Layout Beans.

2. On the JViews Graph Layout Beans toolbar, click the `IlvLinkLayout` Bean icon and drag it to the Form Designer. Place it under the `IlvGrapher` Bean.

   You are now going to associate the `IlvLinkLayout` Bean with the `IlvGrapher` object.

3. Make sure that the `IlvLinkLayout` Bean is selected in the Form Designer. Its properties should be displayed in the Property List.

4. In the Property List window, click the value field of the `Grapher` property. Change `Default` to `ilvGrapher1`. This indicates to the layout Bean which grapher is to be laid out.



Property List - JApplet1

ilvLinkLayout1

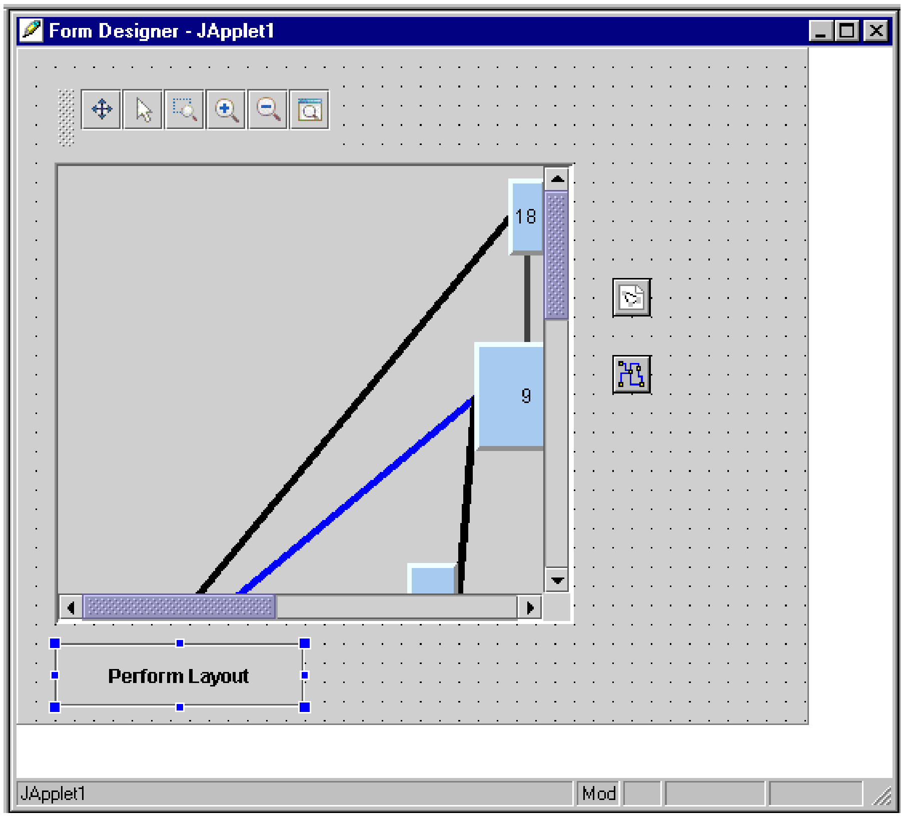
AllowedTime	32000
Animate	false
AutoLayout	true
GlobalLinkStyle	Orthogonal links
Grapher	ilvGrapher1
HorizontalGridBase	(Default)
HorizontalGridOffset	ilvGrapher1
HorizontalMinOffset	
IncrementalMode	false
InputCheckEnabled	true
LayoutMode	Short links
LinkOffset	2.0
MinBusyTime	50
MinFinalSegmentLength	10.0
MinNodeCornerOffset	3.0

Adding a Swing Bean

This stage involves adding a Swing Bean to the applet. This Bean will allow you to launch the layout process on the graph.

To add the Swing Bean:

1. On the Beans toolbar, click the Swing tab to display the Swing Beans.
2. On the Swing Beans toolbar, click the JButton Bean icon and drag it to the Form Designer. Place this button below the `IlvJScrollManagerView` bean.
3. Make sure that the `JButton` Bean is selected in the Form Designer. Its properties should be displayed in the Property List.
4. In the Property List window, click the value field of the `Text` property. Change `jbutton` to `Perform Layout`.



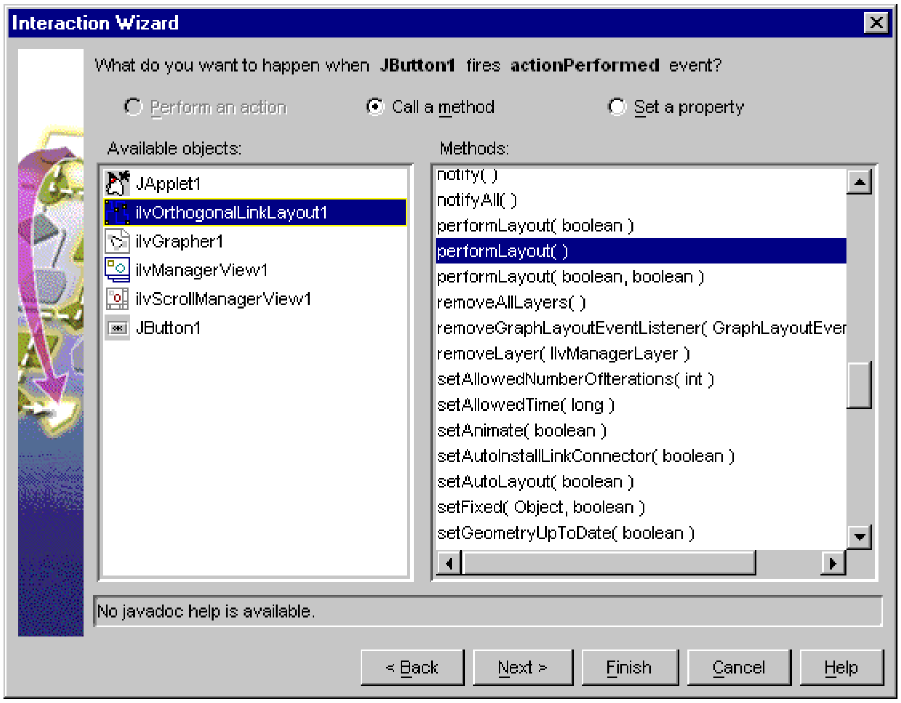
Adding user interaction

This stage involves defining the action that will be performed when the user clicks the Perform Layout button.

To define the user interaction:

1. With the right mouse button, click the `JButton` Bean in the Form Designer.
2. From the pop-up menu that appears, select Add interaction.

   The Interaction Wizard appears.
3. Click Next on the first page of the wizard.
4. In the Events area, make sure that `actionPerformed` is selected. Click Next.
5. On the next page of the Interaction Wizard, click “Call a method”.
6. In the Available objects area, select `IlvLinkLayout1`.
7. In the Methods area, select `performLayout()`.


Interaction Wizard

What do you want to happen when **JButton1** fires **actionPerformed** event?

Perform an action [✓ Call a method] Set a property

Available objects:

- JApplet1
- ilvOrthogonalLinkLayout1
- ilvGrapher1
- ilvManagerView1
- ilvScrollManagerView1
- JButton1

Methods:

```
notify( )
notifyAll( )
performLayout( boolean )
performLayout( )
performLayout( boolean, boolean )
removeAllLayers( )
removeGraphLayoutEventListener( GraphLayoutEver
removeLayer( IlvManagerLayer )
setAllowedNumberOfIterations( int )
setAllowedTime( long )
setAnimate( boolean )
setAutoInstallLinkConnector( boolean )
setAutoLayout( boolean )
setFixed( Object, boolean )
setGeometryUpToDate( boolean )
```

No javadoc help is available.

8. Click Finish to apply the interaction. The Interaction Wizard closes.

Now when the user clicks the Perform Layout button, the layout is performed and the links of the graph are reshaped to orthogonal. (The links appear as alternating horizontal and vertical segments.)

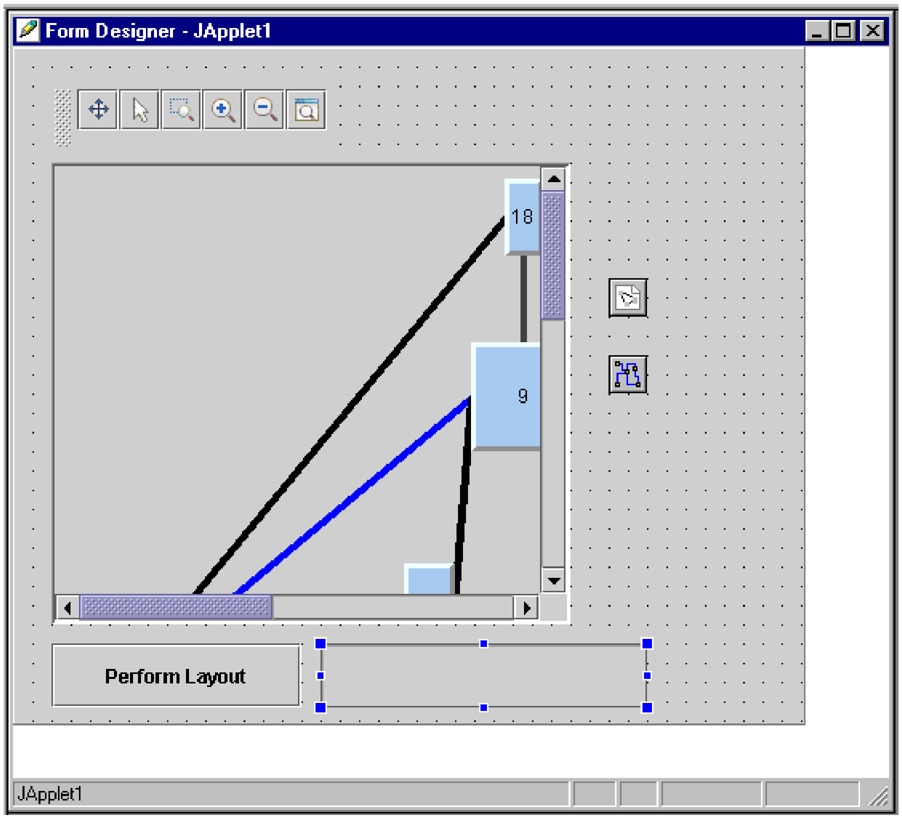
Adding a progress bar to the Applet

This stage involves adding a progress bar to your applet. It is not mandatory. In some cases, a layout algorithm may take a long time to perform a layout and it may be useful to keep the user informed of the activity of the layout.

To add the progress bar:

1. On the Beans toolbar, click the JViews Graph Layout tab to display the JViews Graph Layout Beans.

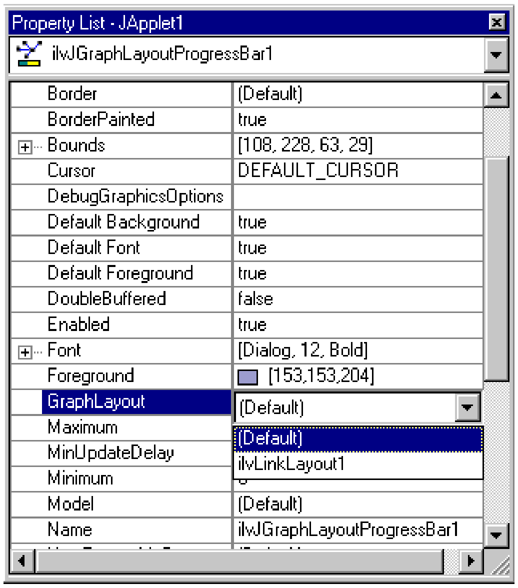
2. On the JViews Graph Layout Beans toolbar, click the `IlvJGraphLayoutProgressBar` Bean icon and drag it to the Form Designer. Place in under the `IlvJScrollManagerView` Bean next to the Perform Layout button.



You are now going to associate the `IlvLinkLayout` Bean with the `IlvJGraphLayoutProgressBar` object.

3. Make sure that the `IlvJGraphLayoutProgressBar` Bean is selected in the Form Designer. Its properties should be displayed in the Property List.

4. In the Property List window, click the value field of the `GraphLayout` property. Change `Default` to `ilvLinkLayout1`.


Property List - JApplet1

ilvJGraphLayoutProgressBar1

Border	(Default)
BorderPainted	true
Bounds	[108, 228, 63, 29]
Cursor	DEFAULT_CURSOR
DebugGraphicsOptions	
Default Background	true
Default Font	true
Default Foreground	true
DoubleBuffered	false
Enabled	true
Font	[Dialog, 12, Bold]
Foreground	[153,153,204]
GraphLayout	(Default)
Maximum	
MinUpdateDelay	
Minimum	
Model	(Default)
Name	ilvJGraphLayoutProgressBar1

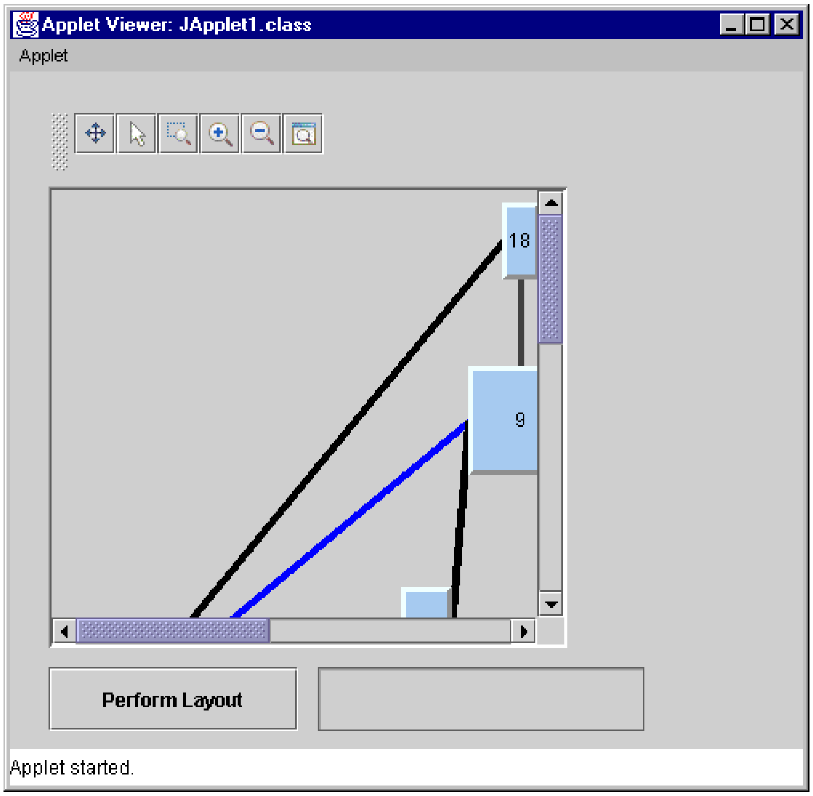
Testing the result

Now that the applet has been created, you can test the result.

To test the resulting applet:

1. Select Project>Execute to execute the applet.

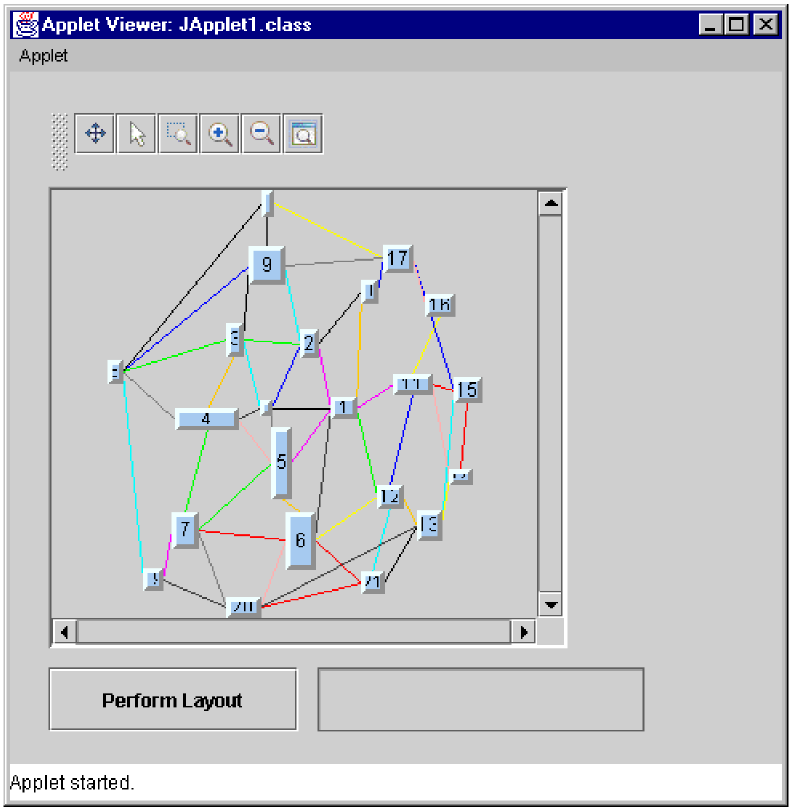
   Initially, the resulting application should appear as shown in the following image (the graph is not yet laid out).



2. You can use the icons in the toolbar to manipulate the graph displayed in the manager view. The toolbar contains the following icons:

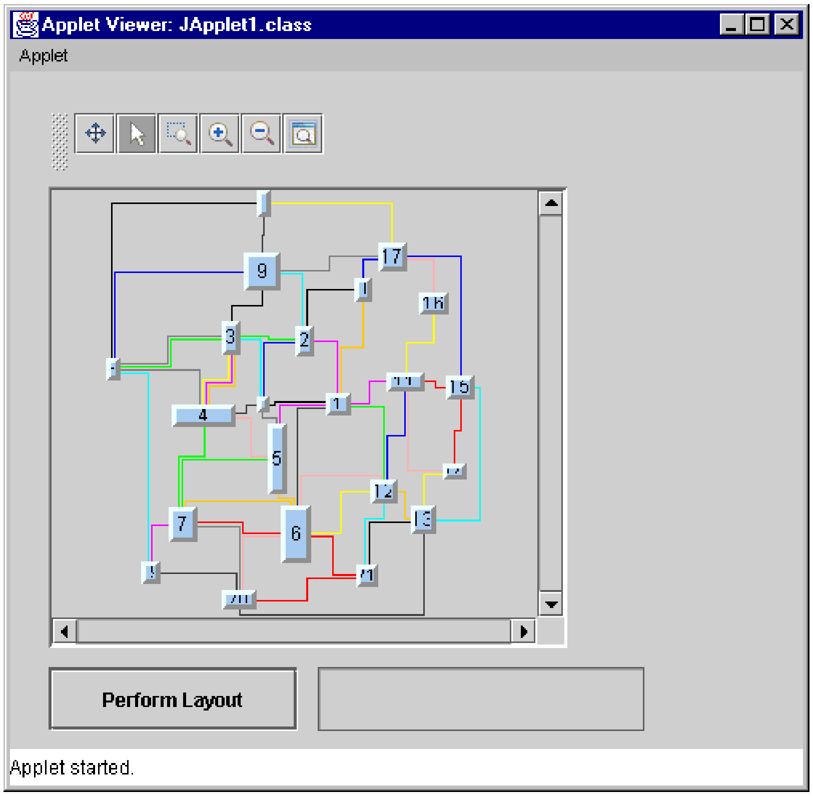
- The Pan icon to pan the content of a view.
- The Select arrow icon to select and edit objects in the view.
- The Interactive zoom icon to drag a rectangle over an area that you want to zoom.
- The Zoom-in and the zoom-out icons to view the graph at different zoom levels.
- The Fit-to-content icon to size the graph so that it fits entirely in the manager view.

  When you click the Fit-to-content icon, the applet window appears as shown in the following image.



3. To lay out the graph, click the Perform Layout button.

As the layout is being performed, you should see the progress indicated in the progress bar at the bottom of the window. When the layout is completed, the applet window should appear similar to the window in the following image.



Applet started.

This completes the graph layout Beans tutorial. For information on how to save your project along with the type of files that are generated when saving, refer to the documentation of your IDE.

Index