IBM ILOG JViews Diagrammer V8.6

Developing hypergraphs

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

Copyright

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

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

Trademarks

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

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

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

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

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

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

Notices

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

Table of contents

Using hypergraphs....5

Introducing hypergraphs....6

Managing nodes and hyperedges....9

Connecting nodes with hyperedges....10

Differences between IlvLinkImage and IlvHyperEdge....12

Hyperedge ends....14

Retrieving incident hyperedges at nodes....18

Contact points....21

Overview....22

Using hyperedge connectors....23

Calculating contact points....25

Visible and invisible hyperedge connectors....27

Hyperedge pin connectors....28

Other predefined hyperedge connectors....31

Segmented hyperedges....33

Overview....34

Hyperedge segments....35

The angle of a segment....37

Segment operations....41

Class summary....44

Using more advanced features....45

© Copyright IBM Corp. 1987, 2009

Nested hypergraphs......47
Overview......48
Intergraph hyperedges......49
Adding intergraph hyperedges......51
Accessing intergraph hyperedges......53
Advanced use of hypergraphs......54
Index......55

Using hypergraphs

Introduces the concepts of hypergraphs and hyperedges and tells you how to use them.

In this section

Introducing hypergraphs
Describes the class structure of hypergraphs.

Managing nodes and hyperedges
Describes how to connect nodes using hyperedges, the differences between `IlvLinkImage` and `IlvHyperEdge`, hyperedge ends, and how to retrieve incident hyperedges at nodes.

Contact points
Describes how to use hyperedge connectors and calculate contact points, and discusses visible and invisible hyperedge connectors, hyperedge pin connectors, and other predefined types of hyperedge connectors.

Segmented hyperedges
Describes hyperedge segments, their angles and operations, and provides a summary of the classes used to work with them.

Class summary
Provides a summary of the classes needed for hypergraphs.

© Copyright IBM Corp. 1987, 2009

Introducing hypergraphs

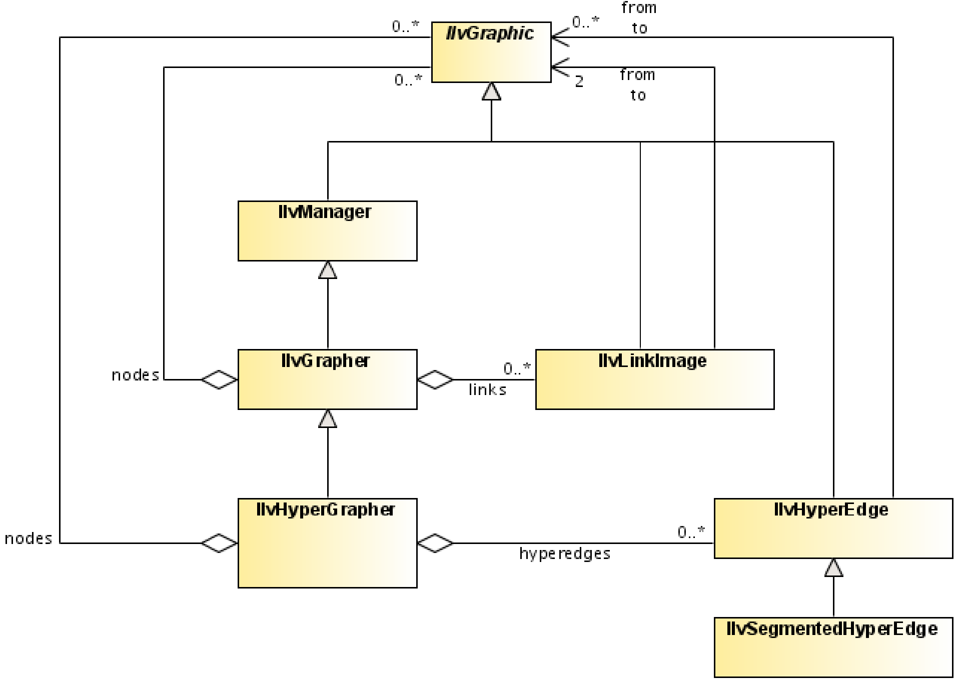
Hypergraphs typically occur in applications such as electrical signal diagrams, multiflow visualization, network management, and UML diagrams.

Hypergraphs are implemented with the class `IlvHyperGrapher`. Hypergraphs are based on graphers and extend them with the concept of hyperedges. Hyperedges are links that have multiple sources and multiple sinks.

A hypergraph is an instance of the class `IlvHyperGrapher`, a subclass of `IlvGrapher` (and of `IlvManager`). While a normal graph can only contain nodes and links, a hypergraph can contain three types of object:

- Nodes, as in an `IlvGrapher`.
- Links, as in an `IlvGrapher`, that connect a source node to a target node
- Hyperedges, which connect multiple nodes and cannot exist in an `IlvGrapher`.

The following diagram shows the associations between the classes relevant to hypergraphs.



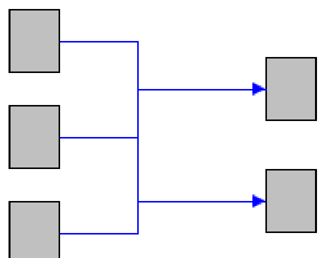
Classes for hypergraphs

A normal link has exactly two end nodes: a source and a target. Sometimes the source node is called the origin and the target is called the destination. If the link is directional, an

arrowhead is drawn at the target node to indicate the direction of the link. A normal graph has nodes and normal links.

A hyperedge is similar to a link, but can have multiple source nodes and multiple target nodes; that is, 0, 1, 2, 3, ... nodes. A hyperedge with 0 source nodes and 0 target nodes is feasible, even though it may not make sense in many applications. There is no restriction in JViews Diagrammer on how many source or target nodes may exist.

A hypergraph is a graph that can contain hyperedges.



Several nodes connected by one hyperedge

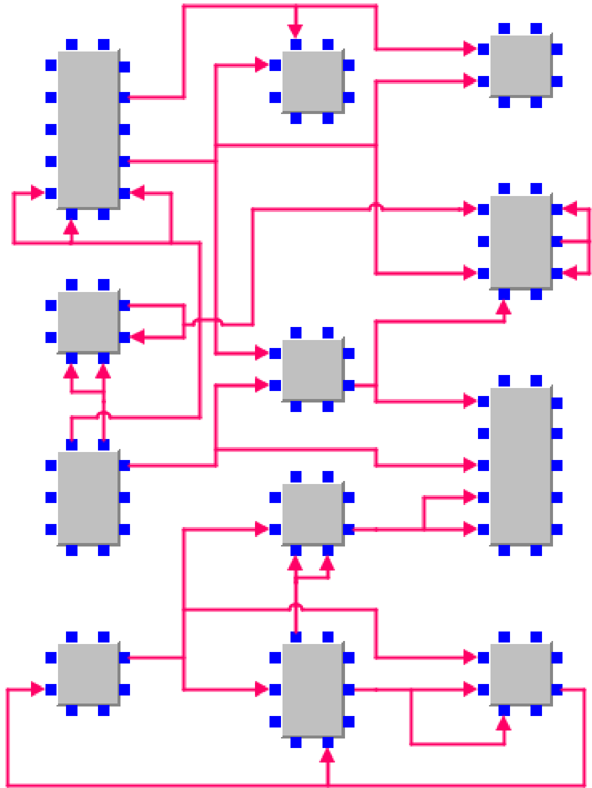
In JViews Diagrammer, graphs, nodes, links, hypergraphs , and hyperedges are implemented by the classes listed in the following table.

Classes for graphs and hypergraphs

Class Name	Description
`IlvGraphic`	The base class of every drawable object, that is, for nodes, links, graphs, hyperedges, and so on. These objects are referred to as graphic objects. Every graphic object other than a link or a hyperedge is always treated as a node.
`IlvGrapher`	A normal graph with normal nodes and links.
`IlvLinkImage` and its subclasses	A normal link.
`IlvHyperGrapher`	A hypergraph that can contain nodes, links, and hyperedges. It is a subclass of `IlvGrapher`. Therefore, it can also contain normal links. An `IlvHyperGrapher` object is derived from `IlvGrapher` and from `IlvManager`, so it may sometimes be referred to as a grapher or a manager, depending on the context (functionality typical of `IlvGrapher` or `IlvManager`). When the functionality under discussion is outside these contexts, the object is called a hypergraph.
`IlvHyperEdge`	A hyperedge. This class is a subclass of `IlvGraphic` and *not* of `IlvLinkImage`. Therefore, hyperedges are incompatible with normal

Class Name	Description
	links. Even so, the end nodes of hyperedges are normal nodes (instances of `IlvGraphic`).
`IlvSegmentedHyperEdge`	A special subclass of `IlvHyperEdge`. Therefore, this is also a hyperedge.

The following diagram shows an example of a hypergraph with many hyperedges used in an electrical signal diagram. The hyperedges are shown in red.



An electrical signal diagram with hyperedges

Managing nodes and hyperedges

Describes how to connect nodes using hyperedges, the differences between `IlvLinkImage` and `IlvHyperEdge`, hyperedge ends, and how to retrieve incident hyperedges at nodes.

In this section

Connecting nodes with hyperedges
Shows how to add a node to a hypergraph, how to connect several nodes via a hyperedge, and provides an example of creating a hypergraph with four nodes and a hyperedge.

Differences between IlvLinkImage and IlvHyperEdge
Describes the `IlvLinkImage` and `IlvHyperEdge` classes and discusses orientation and changing the end nodes of a link.

Hyperedge ends
Describes how hyperedges can have not only multiple end nodes, but multiple ends at the same node.

Retrieving incident hyperedges at nodes
Defines the methods for retrieving links or hyperedges incident to a node.

Connecting nodes with hyperedges

To connect multiple nodes by a hyperedge, you must create the nodes and add them to the hypergraph. This works in `IlvHyperGrapher` exactly in the same way as in `IlvGrapher`.

To add a node:

- Call one of the following methods:
  - `void addNode(IlvGraphic obj, boolean redraw)`
  - `addNode(ilog.views.IlvGraphic, int, boolean)`

To connect several nodes by a hyperedge:

1. Create the hyperedge.

   The constructor of the hyperedge is:

   `IlvHyperEdge(ilog.views.IlvGraphicVector, ilog.views.IlvGraphicVector)`

   While the constructor of `IlvLinkImage` takes only one node as the from or the to node, the constructor of `IlvHyperEdge` takes a vector of nodes as from and to nodes.

2. Add the hyperedge to the hypergraph with one of the following methods:

   - `addHyperEdge(ilog.views.hypergraph.IlvHyperEdge, boolean)`
   - `addHyperEdge(ilog.views.hypergraph.IlvHyperEdge, int, boolean)`

     The layer parameter in the second method specifies the manager layer where the hyperedge is placed. If the layer parameter is missing, the link insertion layer, which is obtained by using the method `getLinkInsertionLayer()`, is used to place the hyperedge.

The following code example shows how to create a hypergraph, four nodes, and a hyperedge.

Creating a hypergraph with four nodes and a hyperedge

```
IlvHyperGrapher grapher = new IlvHyperGrapher();
IlvGraphicVector fromNodes = new IlvGraphicVector();
IlvGraphicVector toNodes = new IlvGraphicVector();
IlvGraphic node1 = new IlvLabel(new IlvPoint(0,0), "node 1");
IlvGraphic node2 = new IlvLabel(new IlvPoint(100,0), "node 2");
IlvGraphic node3 = new IlvLabel(new IlvPoint(100,30), "node 3");
IlvGraphic node4 = new IlvLabel(new IlvPoint(100,60), "node 4");
grapher.addNode(node1, false);
grapher.addNode(node2, false);
grapher.addNode(node3, false);
grapher.addNode(node4, false);
fromNodes.addElement(node1);
toNodes.addElement(node2);
toNodes.addElement(node3);
toNodes.addElement(node4);
```

```
IlvHyperEdge edge = new IlvHyperEdge(fromNodes, toNodes);
grapher.addHyperEdge(edge, false);
```

Differences between IlvLinkImage and IlvHyperEdge

The classes `IlvLinkImage` and `IlvHyperEdge` differ in the following areas:

- *Orientation*
- *Changing end nodes of a link*

Orientation

Normal links (`IlvLinkImage`) have a flag that indicates whether they are oriented. If the links are oriented, an arrowhead will be drawn. The orientation flag is necessary because, if an arrowhead were always drawn, then there would be no way of drawing a link between two nodes that did not represent a flow direction.

Hyperedges are always oriented. Therefore, an arrowhead is drawn at the target nodes and this arrowhead cannot be switched off.

This feature is not a restriction. If you want to draw a hyperedge between several nodes without an arrowhead, create a hyperedge that has all these nodes as sources, but has no target node.

You can also create a hyperedge that has only target nodes, but no source node. In this case, an arrowhead is drawn at all end nodes of the hyperedge.

You can see that hyperedges are much more flexible than normal links.

If a hyperedge has exactly one source node and one target node, then it will look like, and behave approximately as, a normal oriented link.

If a hyperedge has exactly two source nodes and zero target nodes, then it will look like a normal unoriented link.

Changing end nodes of a link

To change the end nodes of an `IlvLinkImage` object, you must remove the `IlvLinkImage` from the graph. Then you must change the end nodes and reinsert the link into the graph.

Changing the end nodes of an `IlvHyperEdge` object is more convenient. The end nodes can be changed while the hyperedge remains in the hypergraph. For example, you can add an empty hyperedge with zero end nodes to the hypergraph. Then you can add source and target nodes later. You must encapsulate all operations that change the end nodes of a hyperedge into `applyToObject(ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)` sessions. See Modifying geometric properties of objects in *The Essential Features of JViews Framework* user documentation for more details.

The operations that can be carried out on `IlvHyperEdge` are shown in the following table.

Operations on IlvHyperEdge

Operation	Description
`setFrom(nodeVector)`	Removes all old source nodes from the hyperedge and adds all nodes contained in the input vector as new source nodes of the hyperedge.
`setTo(nodeVector)`	Removes all old target nodes from the hyperedge and adds all nodes contained in the input vector as new targets of the hyperedge.
`addFrom(node)`	Adds the input node to the list of source nodes of the hyperedge.
`addTo(node)`	Adds the input node to the list of target nodes of the hyperedge.
`removeFrom(node)`	Removes the input node from the list of source nodes of the hyperedge.
`removeTo(node)`	Removes the input node from the list of target nodes of the hyperedge.

The following code example shows how to create an empty hyperedge, add it to the hypergraph, and then add some source nodes.

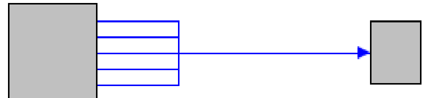
Creating an empty hyperedge and adding source nodes

```
IlvHyperGrapher grapher = new IlvHyperGrapher();
grapher.setMinHyperEdgeEndCount(0);
...
IlvHyperEdge edge = new IlvHyperEdge();
grapher.addHyperEdge(edge, false);
...
grapher.applyToObject(edge,
  new IlvApplyObject() {
    public void apply(IlvGraphic obj, Object arg) {
      ((IlvHyperEdge)obj).addFrom(node1);
      ((IlvHyperEdge)obj).addFrom(node2);
      ((IlvHyperEdge)obj).addFrom(node3);
    }
  }, arg, redraw);
```

Hyperedge ends

Normal links (IlvLinkImage) can be incident to one node at most twice: if the link is a selfloop, that is, if the source and target of the link is the same node, then the link will occur in the list of incident links at the node twice. Both ends of the link are easily distinguishable, because one is the source end and the other is the target end.

Not only can a hyperedge (IlvHyperEdge) have multiple end nodes, but it can also have multiple ends at the same node. Therefore this link is incident to the node multiple times. For example, if you call addFrom(ilog.views.IlvGraphic) multiple times for the same node at the same hyperedge, this hyperedge will have multiple source ends at that node. See the following figure.



Multiple ends at the same node

How can you distinguish the different ends of the hyperedge at the node?
The interface IlvHyperEdgeEnd serves as the end indicator of a hyperedge. Instances of this interface are called the hyperedge ends. The default implementation is IlvDefaultHyperEdgeEnd.

The hyperedge ends technically belong to the hyperedge and not to the node. They point to the end node.

If a source or target node is added to the hyperedge, a new hyperedge end is created internally and connects the hyperedge to the node. If the hyperedge is removed from the hypergraph, all ends of the hyperedge will also be removed.

If a hyperedge connects to one node multiple times, there will be multiple different instances of IlvHyperEdgeEnd stored in the hyperedge. All these instances point to the node. The hyperedge ends can be distinguished from each other, even though the corresponding end node is the same.

The following table shows methods included in the API of IlvHyperEdgeEnd.

Methods of IlvHyperEdgeEnd

Method	Description
getHyperEdge()	Returns the hyperedge that contains the specified end.
getNode()	Returns the node that is connected to the hyperedge by the end.
getPosition(ilog.views. IlvTransformer, boolean)	Returns the position of the end. The input transformer indicates the coordinate system (untransformed manager coordinates or transformed view coordinates if t is the transformer for drawing

Method	Description
	the hyperedge). Usually, you should pass the value `true` for the flag.
`setPosition(ilog.views.IlvPoint, ilog.views.IlvTransformer)`	Sets the position of the end.

The following table shows the API of `IlvHyperEdge` in more detail. It explains how the operations influence the hyperedge ends and which additional API exists for manipulating the hyperedge ends directly at the hyperedge.

Methods of IlvHyperEdge

API	Description
`addFrom(ilog.views.IlvGraphic)`	Creates a new hyperedge end that points to the input node. The hyperedge end is stored as source in the hyperedge. Finally, the hyperedge end is returned, so that you can use it to set the precise position of the hyperedge end. Note that this modifies the geometry of the hyperedge and therefore must be encapsulated in an `applyToObject(ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)` session.
`addTo(ilog.views.IlvGraphic)`	Creates a new hyperedge end that points to the input node. The hyperedge end is stored as target in the hyperedge. Finally, the hyperedge end is returned, so that you can use it to set the precise position of the hyperedge end. Note that this modifies the geometry of

API	Description
	the hyperedge and therefore must be encapsulated in an `applyToObject` session.
`removeFrom(ilog.views.IlvGraphic)`	Collects *all* source hyperedge ends that point to the input node. These ends are removed from the sources of the hyperedge.
`removeTo(ilog.views.IlvGraphic)`	Collects *all* target hyperedge ends that point to the input node. These ends are removed from the targets of the hyperedge.
`removeFrom(ilog.views.hypergraph.IlvHyperEdgeEnd)`	Removes *one* source hyperedge end. Other hyperedge ends are not affected, even if they point to the same node.
`removeTo(ilog.views.hypergraph.IlvHyperEdgeEnd)`	Removes *one* target hyperedge end. Other hyperedge ends are not affected, even if they point to the same node.
`getFrom()`	Returns the source nodes of the hyperedge. Each node occurs at the most once in the enumeration, even if multiple hyperedge ends point to this node.
`getTo()`	Returns the target nodes of the hyperedge. Each node occurs at the most once in the enumeration, even if multiple hyperedge ends point to this node.
`getFromCount()`	Returns the number of source nodes of the hyperedge.
`getToCount()`	Returns the number of target nodes of the hyperedge.
`getFromEnds()`	Returns the source hyperedge ends.
`getToEnds()`	Returns the target hyperedge ends.
`getFromEndsCount()`	Returns the number of source hyperedge ends.
`getToEndsCount()`	Returns the number of target hyperedge ends.

As a general rule, when the end nodes are retrieved or counted, a node occurs once only, even if there are multiple ends at a node. When the ends are retrieved or counted, multiple ends that point to the same node occur as distinguishable items. Therefore, the following equation is true:

```
edge.getFromCount() <= edge.getFromEndsCount()
edge.getToCount() <= edge.getToEndsCount()
```

Important: You should never remove source nodes or source ends while iterating over the objects returned by `getFrom()` or `getFromEnds()`.

You should never remove target nodes or target ends while iterating over the objects returned by `getTo()` or `getToEnds()`.

The class `IlvHyperEdge` has other useful API, such as API for retrieving the ends or end nodes in an array, which is convenient if you need to remove the ends, or for checking whether a given end is source or target. See the reference documentation of `IlvHyperEdge` for details.

Retrieving incident hyperedges at nodes

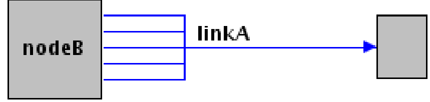
Nodes are any `IlvGraphic` object that is added to the graph by a call to `addNode(ilog.views.IlvGraphic, boolean)`. Instances of `IlvLinkImage` or `IlvHyperEdge` cannot be added as nodes to the hypergraph; they must be added as links (by a call to `addLink(ilog.views.IlvLinkImage, boolean)`) or as hyperedges (by a call to `addHyperEdge(ilog.views.hypergraph.IlvHyperEdge, boolean)`).

The same node can serve as end node for normal links and for hyperedges. `IlvHyperGrapher` provides different API for retrieving links or hyperedges that are incident to a node.

Methods for retrieving links or hyperedges incident to a node

API	Description
`getLinksFrom(ilog.views.IlvGraphic)`	Returns the normal links (`IlvLinkImage`) that have the node as source, but does not return any hyperedges.
`getLinksTo(ilog.views.IlvGraphic)`	Returns the normal links (`IlvLinkImage`) that have the node as target, but does not return any hyperedges.
`getHyperEdgesFrom(ilog.views.IlvGraphic)`	Returns the hyperedges (`IlvHyperEdge`) that have the node as source, but does not return any normal links.
`getHyperEdgesTo(ilog.views.IlvGraphic)`	Returns the hyperedges (`IlvHyperEdge`) that have the node as target, but does not return any normal links.
`getLinksFromCount(ilog.views.IlvGraphic)`	Returns the number of normal links that have the node as source.
`getLinksToCount(ilog.views.IlvGraphic)`	Returns the number of normal links that have the node as target.
`getHyperEdgesFromCount(ilog.views.IlvGraphic)`	Returns the number of hyperedges that have the node as source.
`getHyperEdgesToCount(ilog.views.IlvGraphic)`	Returns the number of hyperedges that have the node as target.

A hyperedge can have the same source node or the same target node multiple times. Say that a hyperedge, `linkA`, has the node, `nodeB`, five times as source node. (See the following figure.) By calling the methods shown in table *Methods for retrieving links or hyperedges incident to a node*, `linkA` would occur once only, even if it is incident to a node multiple times; that is, `grapher.getHyperEdgesFrom(nodeB)` would return `linkA` once only and `grapher.getHyperEdgesFromCount(nodeB)` would count `linkA` once only.



Multiple ends at the same node

Multiple occurrences of the same source or target node are distinguished by the hyperedge end. The hyperedge `linkA` has five different hyperedge ends that point to `nodeB`. Therefore, `IlvHyperGrapher` provides methods for retrieving the hyperedge ends incident to a node. These methods are shown in the following table.

Methods for retrieving hyperedge ends

Method	Description
`getHyperEdgeEndsFrom(ilog.views.IlvGraphic)`	Returns the hyperedge ends (`IlvHyperEdgeEnd`) of those hyperedges that have the node as source.
`getHyperEdgeEndsTo(ilog.views.IlvGraphic)`	Returns the hyperedge ends (`IlvHyperEdgeEnd`) of those hyperedges that have the node as target.
`getHyperEdgeEndsFromCount(ilog.views.IlvGraphic)`	Returns the number of hyperedge ends of those hyperedges that have the node as source.
`getHyperEdgesToCount(ilog.views.IlvGraphic)`	Returns the number of hyperedge ends of those hyperedges that have the node as target.

In the example illustrated by the figure *Multiple ends at the same node*, `grapher.getHyperEdgeEndsFrom(nodeB)` returns five different ends for the link `linkA` and `grapher.getHyperEdgeEndsFromCount(nodeB)` counts `linkA` five times, because it actually counts the ends of `linkA`.

Contact points

Describes how to use hyperedge connectors and calculate contact points, and discusses visible and invisible hyperedge connectors, hyperedge pin connectors, and other predefined types of hyperedge connectors.

In this section

Overview
Provides an overview of hyperedge connectors and their contact points.

Using hyperedge connectors
Describes the class used to define hyperedge connectors.

Calculating contact points
Defines the methods used to create or manipulate hyperedges and shows how to implement a new hyperedge connector and retrieve the end points of a hyperedge.

Visible and invisible hyperedge connectors
Discusses the distinctions between visible and invisible hyperedge connectors, and the different methods used to work with them.

Hyperedge pin connectors
Describes the subclass and methods used to define hyperedge pin connectors.

Other predefined hyperedge connectors
Lists additional predefined hyperedge connectors available in IBM® ILOG® JViews Diagrammer.

Overview

A hyperedge connects the nodes at the point retrieved by the method `getPosition(ilog.views.IlvTransformer, boolean)`. The default implementation `IlvDefaultHyperEdgeEnd` stores the connection in the end itself, relative to the node position. If the node is moved, the hyperedge end will follow the node. The stored connection point can be moved freely by a call to `setPosition(ilog.views.IlvPoint, ilog.views.IlvTransformer)`.

The effect is similar to that of `IlvFreeLinkConnector` on `IlvLinkImage`.

There is a facility for restricting the connection point to certain predefined points by using the hyperedge connector. For example, you can restrict the connection to pin points on the node border or to the center of the node. When a hyperedge connector is used, the connection point is no longer freely movable. It is controlled by the hyperedge connector, which decides whether and how the connection point can be changed.

Using hyperedge connectors

The purpose of the class `IlvHyperEdgeConnector` is comparable to that of the class `IlvLinkConnector` for links. The class `IlvHyperEdgeConnector` is the base class of all hyperedge connectors. It computes the connection points of `IlvHyperEdge` objects at nodes. Subclasses of the abstract base class `IlvHyperEdgeConnector` can be implemented to obtain different contact points.

A hyperedge connector can be attached to a hyperedge or to a node.

If it is attached to a hyperedge, it will control all connection points of this hyperedge at the nodes.

If it is attached to a node, it will control all connection points of hyperedges at the node, except for those hyperedges that do not have their own connector.

To specify that a hyperedge connector is to be used:

1. Allocate the connector.
2. Attach it to the node (see the following code example).

Attaching a hyperedge connector to a node

```
IlvHyperEdgeConnector connector = new IlvHyperEdgeCenterConnector();
connector.attach(node, redraw);
```

-- or --

- Attach it to the hyperedge (see the following code example).

Attaching a hyperedge connector to a hyperedge

```
IlvHyperEdgeConnector connector = new IlvHyperEdgeCenterConnector();
connector.attach(hyperedge, redraw);
```

The same hyperedge connector cannot be shared between several nodes or hyperedges. You need to allocate a new connector for each node or each hyperedge.

If you want to stop using a hyperedge connector, detach it:

```
connector.detach(redraw);
```

When you attach or detach a hyperedge connector, the end points of the hyperedges can change. Therefore, you must specify by using the redraw flag whether the hyperedges are to be redrawn.

To access a hyperedge connector that is responsible for a specific hyperedge end:

- Use the call shown in the following code example:

Accessing the hyperedge connector for a specific hyperedge end

```
IlvHyperEdgeEnd hyperEdgeEnd = ...
```

```
IlvHyperEdgeConnector connector = IlvHyperEdgeConnector.Get(hyperEdgeEnd)
;
```

The call to this method returns the connector of the hyperedge that has the specified end or, if the edge has no connector, it will return the connector of the node connected to that end. If it returns null, no connector at all is used.

To access the connector attached to a specific node or hyperedge:

♦ Use the call shown in the following code example:

Accessing the hyperedge connector for a specific node or hyperedge

```
IlvHyperEdgeConnector connector =
    IlvHyperEdgeConnector.GetAttached(nodeOrHyperEdge);
```

The call to this method returns the hyperedge connector of the node or of the hyperedge, even if this hyperedge connector is not currently responsible for the calculation of the end points of any hyperedge.

Calculating contact points

Interactors that allow you to create or manipulate hyperedges typically call the following methods of `IlvHyperEdgeConnector` to connect a hyperedge end to or disconnect it from a node:

- `connect(ilog.views.hypergraph.IlvHyperEdgeEnd, ilog.views.IlvPoint, ilog.views.IlvTransformer)`
- `disconnect(ilog.views.hypergraph.IlvHyperEdgeEnd)`

To implement a new hyperedge connector:

- Call the methods:

  `getClosestConnectionPoint(ilog.views.hypergraph.IlvHyperEdgeEnd, ilog.views.IlvPoint, ilog.views.IlvTransformer)`

  This method is called when a hyperedge end needs to be connected to a node at position p. This position might be unsuitable for the connector. Therefore, the method `getClosestConnectionPoint` can return the closest suitable connection point. It can be useful to store this connection point internally in the connector, so that it can be retrieved quickly.

  -- or --

  `getConnectionPoint(ilog.views.hypergraph.IlvHyperEdgeEnd, ilog.views.IlvTransformer)`

  This method is called when a hyperedge end is already connected. It returns the connection point for the hyperedge end. This can be the connection point that was previously stored. If no connection point was previously stored, the method must calculate an appropriate connection point for the end.

When the `disconnect` method is called, the storage of the connection point can finally be cleared to avoid memory leaks.

Note: The class `IlvHyperEdgeConnectorWithCache` implements hyperedge connectors with a cache storage for connection points that handles the storing and cleaning up of the cache automatically.

To retrieve the end points of a hyperedge:

- Call the method:

  `getPosition(ilog.views.IlvTransformer, boolean)`

See *Hyperedge ends*.

If the flag `checkConnector` is true, this method asks the connector for the position by calling `getConnectionPoint` at `IlvHyperEdgeConnector`.

If the flag `checkConnector` is false, the method does not ask the connector for the position.

Usually, when a hyperedge is drawn or manipulated, the value true is always passed for the parameter `checkConnector`. Therefore, the hyperedge ends at the point returned by `getConnectionPoint` from the connector.

Visible and invisible hyperedge connectors

In contrast to IlvLinkConnector, the class IlvHyperEdgeConnector is a subclass of IlvGraphic and can display the contact points in the hypergraph. For example, it is useful to display the pins of a pin connector permanently at the nodes that represent electrical circuits in a signal diagram.

Not all subclasses of IlvHyperEdgeConnector use this facility. Like IlvGraphic, some connector classes need to display visible parts. Like IlvLinkConnector, other connector classes are only a logical computation facility for the contact points. Thus, the class IlvHyperEdgeConnector is a mix of a graphic object and a connection calculation algorithm.

Visible hyperedge connectors need to implement the method isGraphic() to return true. They also need to implement all the usual methods of IlvGraphic, namely draw(java.awt.Graphics, ilog.views.IlvTransformer), boundingBox(ilog.views.IlvTransformer), and contains(ilog.views.IlvPoint, ilog.views.IlvPoint, ilog.views.IlvTransformer).

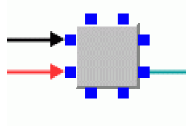
Invisible hyperedge connectors need to implement the method isGraphic() to return false. They do not need to implement the usual methods of IlvGraphic, because when isGraphic() returns false these connectors will not be used as graphic objects.

Visible hyperedge connectors can only be attached to nodes, not to hyperedges. The position of a visible hyperedge connector is always the same position as the node. When the node is moved, the hyperedge connector is automatically moved with the node. When the node is removed or inserted into a different hypergraph, the hyperedge connector is automatically removed and inserted into the same hypergraph as the node; that is, you need only to attach the connector to the node and the entire handling of the hyperedge connector as a graphic object is done automatically.

The only predefined visible hyperedge connector is IlvHyperEdgePinConnector. All other predefined hyperedge connectors are invisible connectors. See IlvHyperEdgeConnector for additional information.

Hyperedge pin connectors

The subclass `IlvHyperEdgeConnector` allows hyperedges to connect to pins at the node. (See the following figure.) Optionally, the pins can be visible. Thus, the pin connector is a visible connector, that is, `isGraphic()` returns true. The pin connector contains a set of pins.



Pin connector

A pin is implemented by the class `IlvHyperGrapherPin` or its subclasses. Before a pin can be used, it must be added to the corresponding pin connector. The API of `IlvHyperEdgePinConnector` includes the methods listed in the following table.

Methods of IlvHyperEdgePinConnector

Method	Description
`addPin(ilog.views.hypergraph.edgeconnector.IlvHyperGrapherPin, boolean)`	Adds the pin to the hyperedge pin connector. If the redraw flag is true and the pin is visible, it will be redrawn.
`removePin(ilog.views.hypergraph.edgeconnector.IlvHyperGrapherPin, boolean)`	Removes the pin from the hyperedge pin connector. If the redraw flag is true and the pin was visible, the area of the pin will be redrawn.
`getPins()`	Returns all pins of the hyperedge pin connector.
`getPinsCount()`	Returns the number of pins in the hyperedge pin connector.
`getClosestPin(ilog.views.hypergraph.IlvHyperEdgeEnd, ilog.views.IlvPoint, ilog.views.IlvTransformer)`	Returns the closest pin that is suitable for the hyperedge end at the input point p.
`connect(ilog.views.hypergraph.IlvHyperEdgeEnd, ilog.views.hypergraph.edgeconnector.IlvHyperGrapherPin)`	Connects the hyperedge end to the input pin. The pin must belong to the hyperedge pin connector. The connector must be attached to a node and the hyperedge end must point to this node.

The class `IlvHyperGrapherPin` is a concrete class that is ready to use. It displays a pin as a small rectangle at the border of the node.

You can also create subclasses of `IlvHyperGrapherPin`. You can specify whether it is possible to connect only one or several hyperedge ends to the same pin. You can restrict even more tightly which hyperedge ends can connect to a pin by overriding the method `allow(ilog.views.hypergraph.IlvHyperEdgeEnd)`.

The API of `IlvHyperGrapherPin` includes the methods listed in.

Methods of IlvHyperGrapherPin

Method	Description
`getConnectedEnds()`	Returns the hyperedge ends that are currently connected to the pin.
`isConnected(ilog.views.hypergraph.IlvHyperEdgeEnd)`	Tests whether the input hyperedge end is connected to this pin.
`allow(ilog.views.hypergraph.IlvHyperEdgeEnd)`	Returns whether the hyperedge end is allowed to be connect to the pin.
`getPosition(ilog.views.IlvTransformer)`	Returns the position of the pin in transformed view coordinates.
`setPosition(ilog.views.IlvPoint, ilog.views.IlvTransformer)`	Sets the position of the pin in transformed view coordinates.
`setSelected(boolean)`	Sets whether the pin is currently selected. A selected pin is drawn in a different color from unselected pins. Usually, pins get selected during interactions that manipulate pins.
`setMovable(boolean)`	Sets whether the pin is movable interactively. Interactors check whether pins are movable before calling `setPosition` on the pin.
`setAllowMultiConnection(boolean)`	Sets whether multiple hyperedge ends can be connected to the same pin. If false is passed, a different pin must be used for each hyperedge end.

The position of the pin is the position where the pin is drawn. It is the center point of the pin, since pins can have a size.

To create a pin, use the constructor:

```
IlvHyperGrapherPin(IlvPoint proportionalLocation,
                   IlvPoint absoluteLocation,
                   float size,
                   int direction)
```

The position of the pin is specified by a proportional part and by an absolute part. The proportional part is a position relative to the bounding box of the node. If the proportional location is (0,0), it will be the top left corner of the node. If the proportional location is (1,1), it will be the bottom right of the node. The real position is calculated in the following way:

```
pin.position = node.position + proportionalLocation * node.size +
absoluteLocation
```

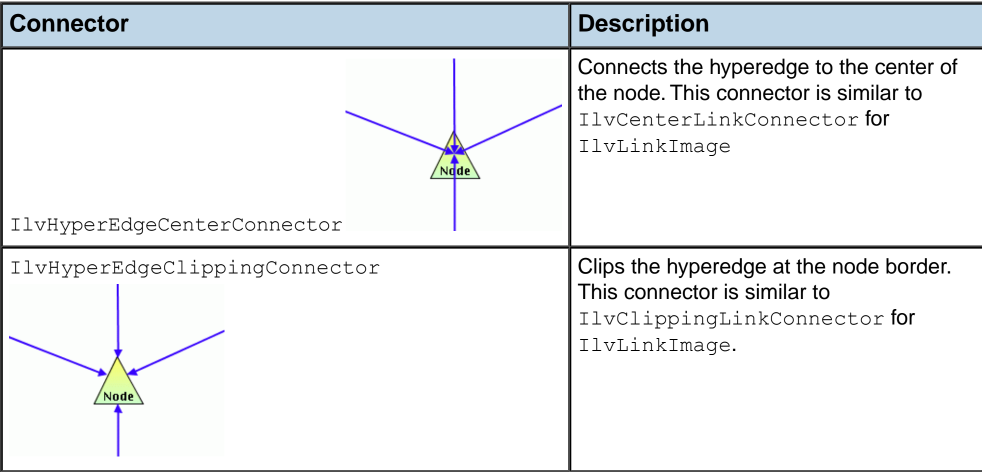
Hyperedges can connect to the pin position (the center point of the pin). Since pins have a size, it is sometimes useful to connect the hyperedges to a different point. If the pin is on the left side of the node, it is better to connect the hyperedge to the left side of the pin. If the pin is on the right side of the node, it is better to connect the hyperedge to the right side of the pin.

This behavior can be controlled by a direction parameter passed to the constructor of the pin. If you pass the direction 0, the pin will decide heuristically which is the best point of the pin to connect a hyperedge to.

Other predefined hyperedge connectors

Other hyperedge connectors are available as shown in the following table.

More predefined hyperedge connectors



Connector	Description
IlvHyperEdgeCenterConnector	Connects the hyperedge to the center of the node. This connector is similar to `IlvCenterLinkConnector` for `IlvLinkImage`
IlvHyperEdgeClippingConnector	Clips the hyperedge at the node border. This connector is similar to `IlvClippingLinkConnector` for `IlvLinkImage`.

These hyperedge connectors are invisible connectors, that is, `isGraphic()` returns false.They can be attached to nodes and hyperedges.

Segmented hyperedges

Describes hyperedge segments, their angles and operations, and provides a summary of the classes used to work with them.

In this section

Overview
Provides an overview of segmented hyperedges, and how they differ from standard hyperedges.

Hyperedge segments
Describes segmented hyperedges, segment trees, and how to retrieve the segments of a hyperedge.

The angle of a segment
Defines the different angles of a hyperedge segment.

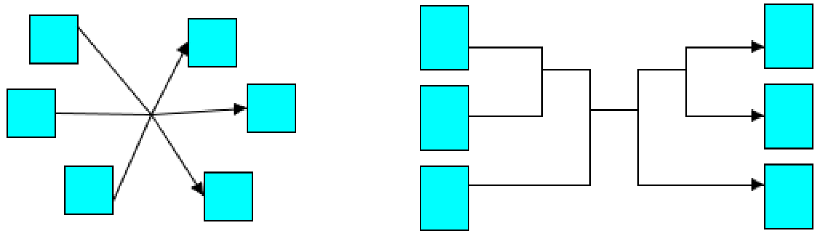
Segment operations
Describes the operations of hyperedge segments, and the methods used to work with them.

Overview

`IlvHyperEdge` is the base class for hyperedges. It is comparable to `IlvLinkImage`, which is the base class for links.

`IlvHyperEdge` draws a star-like shape from a center point of the hyperedge to the ends of the hyperedge.

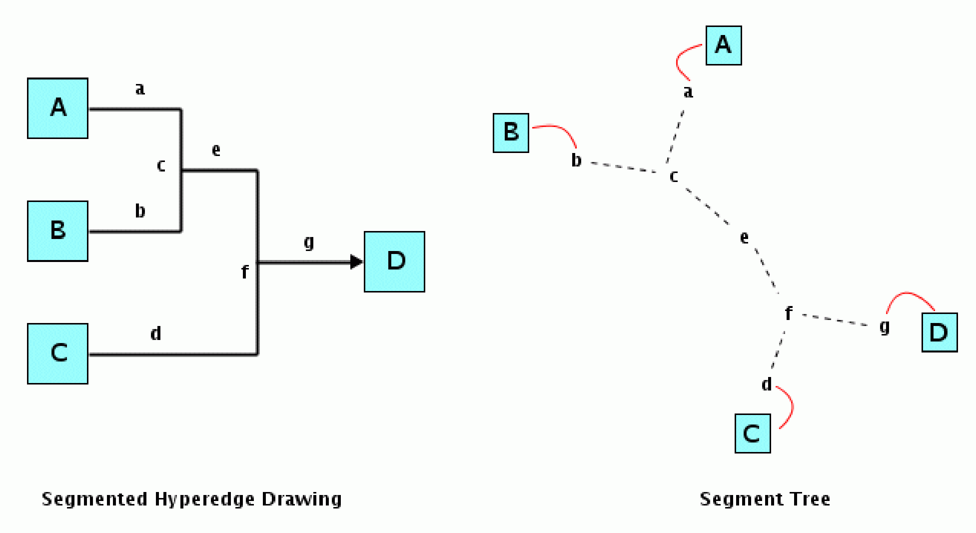
For many applications, the star-like shape is inappropriate, because applications prefer specific shapes, such as a shape consisting of orthogonal segments. To create such shapes, use the class `IlvSegmentedHyperEdge`, which is a subclass of `IlvHyperEdge`. The following figure shows examples of `IlvHyperEdge` (on the left) and `IlvSegmentedHyperEdge` (on the right).



Hyperedge and segmented hyperedge

Hyperedge segments

An `IlvSegmentedHyperEdge` object has a shape that consists of line segments. The line segments form branches that connect all hyperedge ends. Therefore, the segments can be connected to other segments and can be connected to hyperedge ends at a node. We call a segment incident to a node an end segment, and the other segments the inner segments. The segments of a hyperedge form an unrooted, undirected segment tree.



A segmented hyperedge with its segment tree

In the figures above:

- The dashed lines indicate the segments that are incident.
- The red lines indicate which segments are connected to end nodes.
- The segments a, b, d, and g are end segments; the segments c, e, and f are inner segments.

You may be surprised that the segment tree is undirected, even though the hyperedge may be directed. The direction of the hyperedge comes from the fact that arrow heads are drawn at certain end points. This direction is irrelevant for the segment tree; that is, there is no root of the segment tree, but each segment has several (or 0) incident segments. By starting at an arbitrary segment, an algorithm can traverse all incident segments recursively without running into a cycle. Therefore it is an unrooted, undirected segment tree.

A segment is implemented by the inner class `IlvSegmentedHyperEdge.Segment`. To retrieve all segments of a hyperedge, make the call shown in the following code example.

Retrieving all segments of a hyperedge

```
IlvSegmentedHyperEdge seghyperedge = ...
Iterator iterator = seghyperedge.getSegments();
```

```
while (iterator.hasNext()) {
  IlvSegmentedHyperEdge.Segment segment =
    (IlvSegmentedHyperEdge.Segment)iterator.next();
  ...
}
```

The API of `IlvSegmentedHyperEdge.Segment` includes the methods listed in the following table.

Methods of IlvSegmentedHyperEdge.Segment

Method	Description
`getHyperEdge()`	Returns the hyperedge that owns the segment.
`getEnds()`	Returns the hyperedge ends of the segment if it is an end segment. If it returns an empty array, the segment is an inner segment.
`getIncidentSegmentsCount()`	Returns the number of incident segments.
`getIncidentSegment(int)`	Returns the incident segment with index i. The index of incident segments starts at 0.
`isIncident(ilog.views. hypergraph. IlvSegmentedHyperEdge.Segment)`	Tests whether two segments are incident.

If two segments A and B are connected, then A will have B as incident segment and B will have A as incident segment. This allows traversal of the incident segments in any direction. To avoid running into a loop, you only have to make sure that you do not visit the segment again where you immediately came from. The method in the following code example illustrates this concept.

Method for traversal of incident segments in any direction

```
public void visit(IlvSegmentedHyperEdge.Segment segment,
                  IlvSegmentedHyperEdge.Segment cameFrom)
{
  ...
  int n = segment.getIncidentSegmentsCount();
  for (int i = 0; i < n; i++) {
    IlvSegmentedHyperEdge.Segment childSegment =
          segment.getIncidentSegment(i);
    if (childSegment != cameFrom)
      visit(childSegment, segment);
  }
}
...
// visit all segments reachable from startSegment
 visit(startSegment, null);
```

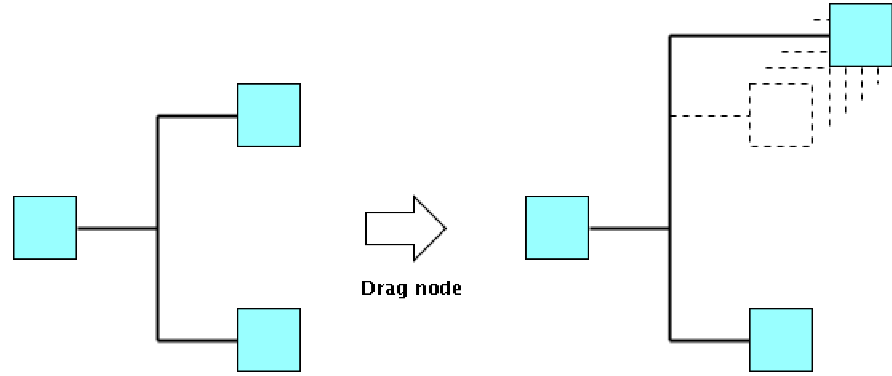
The angle of a segment

Many applications use orthogonal segments, that is, the angle of a segment can be 0 degrees (horizontal) or 90 degrees (vertical).

`IlvSegmentedHyperEdge` supports orthogonal segments and also segments with a different fixed angle. It also supports segments with variable angle.

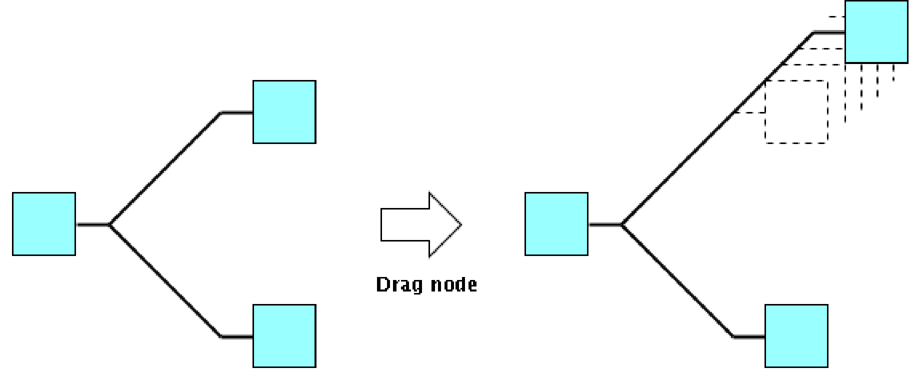
Fixed angle or variable angle

A segment with fixed angle always keeps this angle. When incident segments or end nodes are moved or dragged, a segment will become longer or shorter to stay visually connected, but it will not change its angle. This is shown in figures *Orthogonal segments with fixed angle* and *Segments with fixed angle of 45 degrees*.



Orthogonal segments with fixed angle

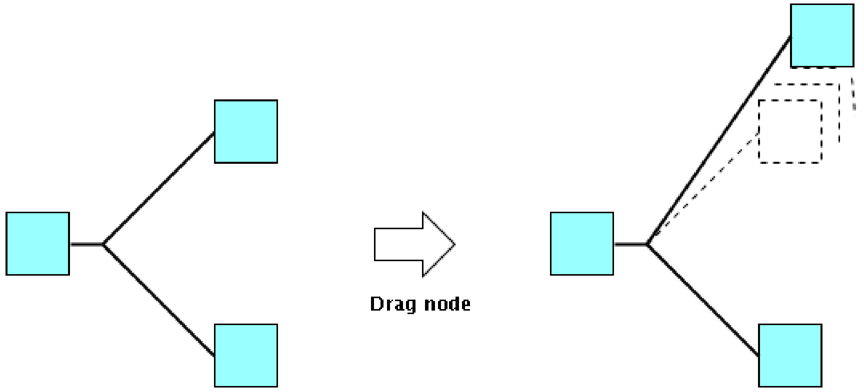
In figure *Orthogonal segments with fixed angle*, when end nodes are dragged, the segments remain orthogonal. They only get longer.



Segments with fixed angle of 45 degrees

In figure *Segments with fixed angle of 45 degrees*, when end nodes are dragged, the segments don't change their angle. They only get longer.

A segment with variable angle always adapts the angle to the situation automatically. When incident segments or end nodes are moved or dragged, the segment will change its angle. This is shown in figure *Segments with variable angle*.



Segments with variable angle

In figure *Segments with variable angle*, when end nodes are dragged, the segments change the angle.

Whether a segment has a variable angle or a fixed angle depends on how the segment was created. This is described later, in *Segment operations*.

To test whether a segment has a variable or a fixed angle, call the method:

```
segment.isVariable()
```

The angle of a segment is always given in degrees between 0 and 180, where 0 means horizontal and 90 means vertical.

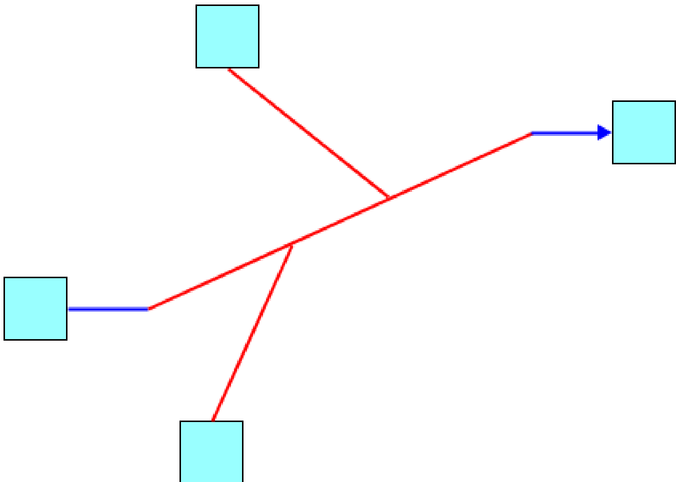
To query the angle of the segment, call the method:

```
segment.getAngle()
```

Constraints of variable and fixed angle segments

In the segment tree, segments of fixed or variable angle can be incident, but there are some constraints:

- ♦ A segment with fixed angle can be incident to any number of other segments with fixed or variable angle.
- ♦ A segment with fixed angle cannot be incident to a segment with the same fixed angle. For example, a horizontal segment cannot be incident to another horizontal segment. If the shape of the hyperedge is orthogonal, then horizontal segments will be incident to vertical segments and vertical segments will be incident to horizontal segments. Segments with the same angle are called colinear. See `isColinear(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment)`. For mathematical reasons, it is required that the angle differs at least by 1 degree.
- ♦ A segment with fixed angle can only be connected to at most one hyperedge end (`getEnds()` returns an array with at most 1 end).
- ♦ A segment with variable angle can be incident to any number of other segments with variable angle.
- ♦ A segment with variable angle usually starts or ends at a node or at a segment. In the latter case, it is said to be terminated by a segment. The terminating segments of a segment with variable angle can have a fixed angle or a variable angle.
- ♦ A segment with variable angle can only be incident to at most two segments with fixed angle. The two segments with fixed angle must terminate the segment with variable angle. See figure *Hyperedge consisting of segments of fixed and variable angle*.
- ♦ A segment with variable angle can be connected to 0, 1, or 2 hyperedge ends (`getEnds()` returns an array with at most 2 ends).



Hyperedge consisting of segments of fixed and variable angle

In figure *Hyperedge consisting of segments of fixed and variable angle*:

- Segments with fixed angle are blue.
- Segments with variable angle are red.
- The long red segment in the middle of the figure is terminated by segments of fixed angle.

Segment operations

This section describes segment operations and their effects.

Complete or incomplete segment tree

Each end of a hyperedge has a segment that is directly incident to the hyperedge end. You can obtain this segment by calling:

```
seghyperedge.getEndSegment(hyperedgeEnd);
```

The segment tree is complete, if each end of the hyperedge is reachable from each other end through a segment traversal. See the method `visit` in section *Method for traversal of incident segments in any direction*.

If this is not the case, the segment tree is incomplete. For example, you have a disconnected forest of segment trees.

To test whether a segment tree is complete, call:

```
seghyperedge.isSegmentSetComplete();
```

Hyperedges with incomplete segment trees can be displayed. Incomplete segment trees look wrong when displayed. They do not convey information on which nodes are connected by the hyperedge. Therefore, you are usually interested only in complete segment trees.

The API of `IlvSegmentedHyperEdge` is flexible enough to allow incomplete segment trees, which simplifies the task of restructuring the segments of a hyperedge.

The segment tree can be made incomplete temporarily, but for the final drawing you are recommended to make the segment tree complete.

Therefore, it is necessary to distinguish between primitive segment operations and safe segment operations.

Primitive operations can make the segment tree incomplete.

Safe segment operations, when applied to a complete segment tree, result in a complete segment tree.

Primitive segment operations

The API on `IlvSegmentedHyperEdge.Segment` allows you to query the information of the segment, but not to manipulate the segments. The manipulation must be done by the API on the hyperedge itself. Any manipulation of the segment tree may change the bounding box of the hyperedge. Therefore, an `applyToObject(ilog.views.IlvGraphic, ilog.views.IlvApplyObject, java.lang.Object, boolean)` session is usually necessary.

The following primitive segment operations allow you to change the segment tree so that it becomes incomplete. The API of `IlvSegmentedHyperEdge` includes the methods listed in the following table.

Methods of IlvSegmentedHyperEdge for making the segment tree incomplete

Method	Description
`addSegment(ilog.views.hypergraph.IlvHyperEdgeEnd, double)`	Creates a new segment that is incident to the hyperedge end. The new segment has a fixed angle.
`addSegment(ilog.views.hypergraph.IlvHyperEdgeEnd, float, float, ilog.views.IlvTransformer)`	Creates a new segment that is incident to the hyperedge end. The new segment has a variable angle. As long as the segment is not terminated by any other segment, it uses the point (x, y) as the other termination point of the segment.
`addSegment(ilog.views.hypergraph.IlvHyperEdgeEnd, ilog.views.hypergraph.IlvHyperEdgeEnd)`	Creates a new segment that is incident to both hyperedge ends. The new segment has a variable angle.
`addSegment(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, double, float, float, ilog.views.IlvTransformer)`	Creates a new segment that is incident to the existing segment. The new segment has a fixed angle. The input point (x,y) determines the position of the segment.
`addSegment(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, float, float, float, float, ilog.views.IlvTransformer)`	Creates a new segment that is incident to the existing segment. The new segment has a variable angle. The input points (x1,y1) and (x2,y2) determine the start and end coordinates of the new segment.
`removeSegment(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment)`	Removes a segment from the segment tree. This makes the segment tree incomplete. Returns the segments that were formerly incident to the removed segment.
`connectSegments(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment)`	Connects two segments so that they become incident. It is not possible to connect segments in a way that would form a cycle in the segment tree. The constraints on variable and fixed angle segments must be observed.
`disconnectSegments(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment)`	Disconnects two segments so that they are no longer incident. This makes the segment tree incomplete.
`isConnected(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment)`	Tests whether two segments are connected.

Safe segment operations

When hyperedge ends are added to or removed from the hyperedge, the segment tree must change to remain complete. The hyperedge has an autoconnect mode that does this automatically.

The following code example shows how to keep the segment tree complete.

Keeping the segment tree complete

```
IlvSegmentedHyperEdge seghyperedge = ...
seghyperedge.setAutoConnect(true);
// ... this will add segments automatically as necessary
seghyperedge.addFrom(node);
seghyperedge.addTo(node);
// ... now the segment tree is complete
// seghyperedge.isSegmentSetComplete() == true
```

The following code example shows how to make the segment tree incomplete.

Making the segment tree incomplete

```
IlvSegmentedHyperEdge seghyperedge = ...
seghyperedge.setAutoConnect(false);
// ... this will not add any segments automatically
seghyperedge.addFrom(node);
seghyperedge.addTo(node);
// ... now the segment tree is incomplete, since the new hyperedge ends
//     are not connected ...
// seghyperedge.isSegmentSetComplete() == false
```

The autoconnect mode is enabled by default.

The API of `IlvSegmentedHyperEdge` contains further operations that keep the segment tree complete.

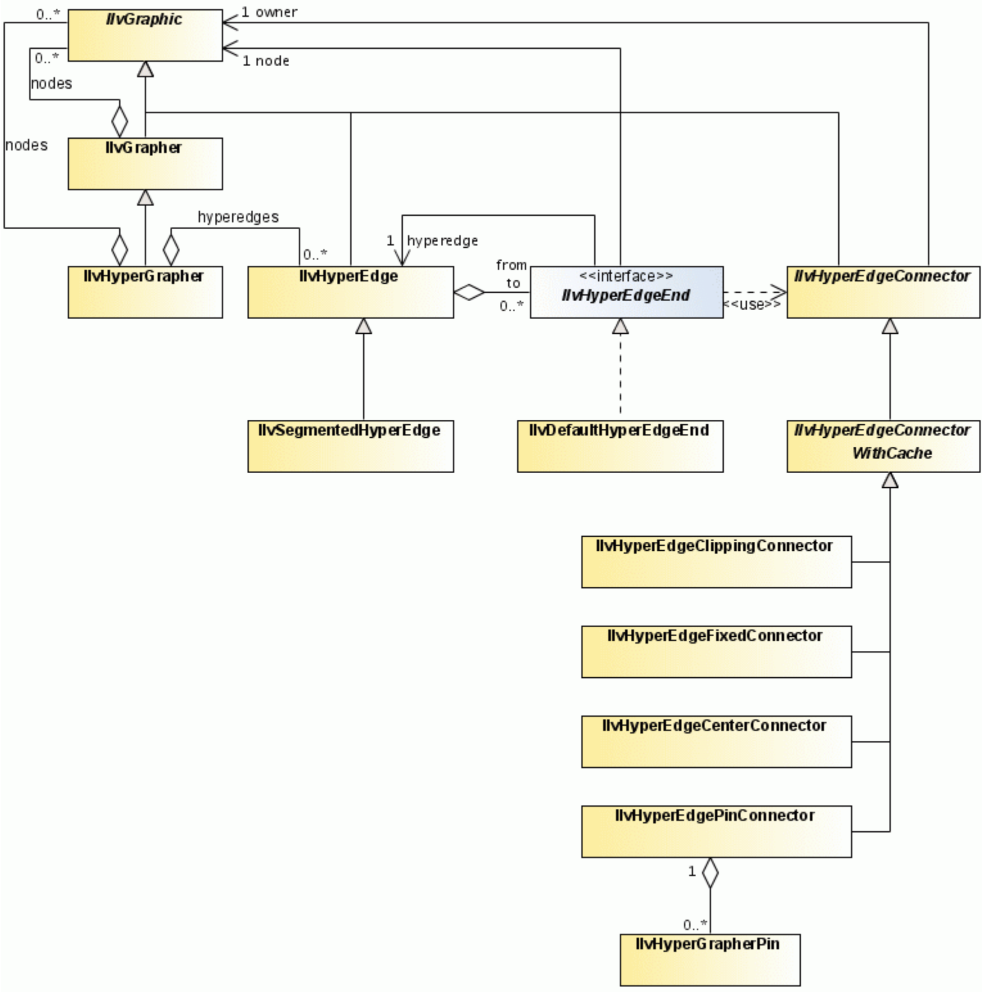
Methods of IlvSegmentedHyperEdge for keeping the segment tree complete

Method	Description
`splitSegment(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, float, float, ilog.views.IlvTransformer)`	Variants of this operation exist. They split a segment into a sequence of segments, that is, a bend will occur at the position where the segment was split. See the Java™ API reference manual for information about the variants.
`joinSegments(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment)`	This is the inverse of the split operation. It joins two segments, so that instead of two, only one segment is used. This operation removes a bend in the shape of the hyperedge.
`setSegmentAngle(ilog.views.hypergraph.IlvSegmentedHyperEdge.Segment, double)`	Changes the angle of the input segment. If the input segment is connected, this will work only if no incident segment is colinear with the new angle.
`setEndSegmentAngle(ilog.views.hypergraph.IlvHyperEdgeEnd, double)`	Changes the angle of the segment that is incident to the input hyperedge end. It reorganizes the segment tree so that the desired angle is possible for the segment at the input end.

See `IlvSegmentedHyperEdge` for more information on segmented hyperedges.

Class summary

The following diagram summarizes the associations between the classes discussed in this documentation. It follows normal UML conventions.



Summary of classes needed for hypergraphs

Using more advanced features

Describes nested hypergraphs and more advanced uses of hypergraphs.

In this section

Nested hypergraphs
Discusses intergraph hyperedges, how to add and access them.

Advanced use of hypergraphs
Discusses an advanced use of hypergraphs: making sure that hyperedges have a minimum number of bends.

© Copyright IBM Corp. 1987, 2009

Nested hypergraphs

Discusses intergraph hyperedges, how to add and access them.

In this section

Overview
Provides an overview of nested hypergraphs.

Intergraph hyperedges
Defines intergraph hyperedges (a hyperedge with its end nodes in different hypergraphs).

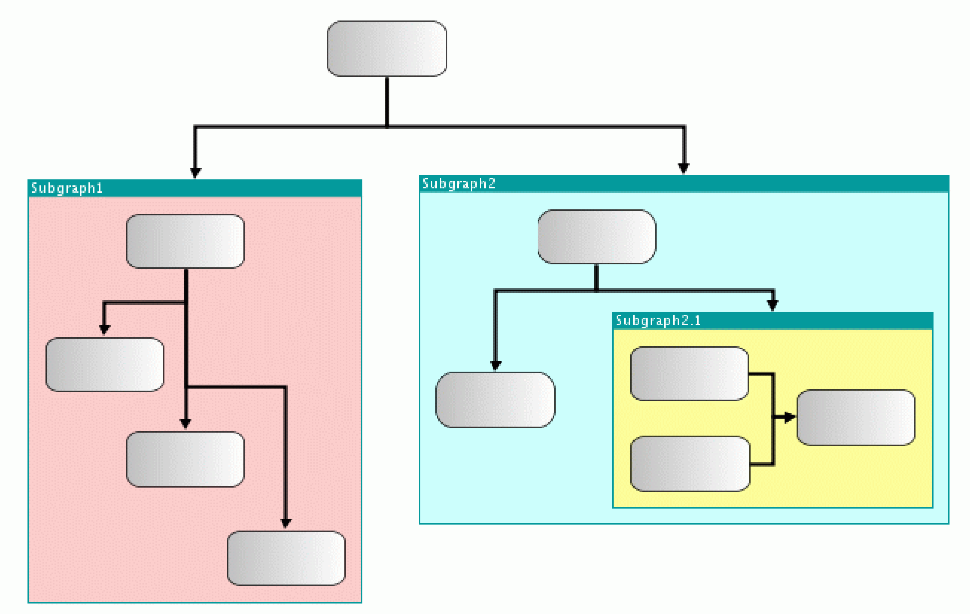
Adding intergraph hyperedges
Shows how to add an intergraph hyperedge in a hierarchy of hypergraphs.

Accessing intergraph hyperedges
Describes the class and methods used to access intergraph hyperedges.

Overview

Instances of `IlvGrapher` can be nested and can be collapsed or expanded. Nested graphers allow you to create applications that define graphs that contain subgraphs with links crossing the graph boundaries (intergraph links). See Nested managers and nested graphers in the *Advanced Features of JViews Framework* user documentation for details.

Similarly, `IlvHyperGrapher` objects can be nested and can contain intergraph hyperedges..



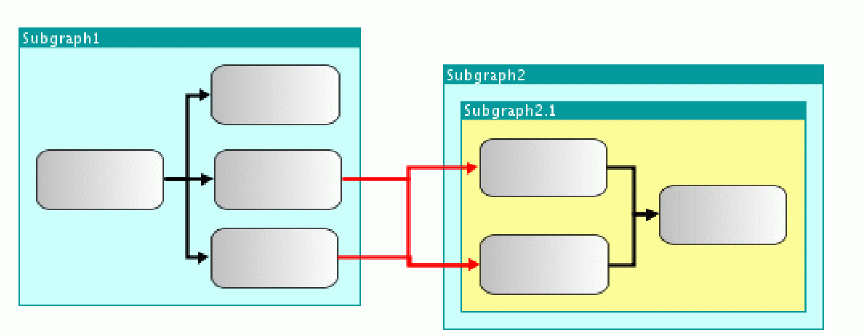
Nested hypergraphs

Intergraph hyperedges

An intergraph hyperedge is a hyperedge with its end nodes in different hypergraphs.

The end nodes must all belong to the same nested hierarchy, that is, they must all be nested inside the same root hypergraph. The concept of intergraph hyperedges is similar to the concept of intergraph links. See Intergraph links in the *Advanced Features of JViews Framework* user documentation.

In the following figure, the intergraph hyperedge is shown in red and normal hyperedges are shown in black.



Intergraph hyperedges and normal hyperedges

To test whether a hyperedge in an intergraph hyperedge, call this method of `IlvHyperEdge`:

```
boolean isInterGraphHyperEdge()
```

This test method works only when the end nodes are already present in the hypergraphs. Since you can add source and target nodes after a hyperedge has been added to a hypergraph, a normal hyperedge can become an intergraph hyperedge. See the following code example.

How a normal hyperedge can become an intergraph hyperedge

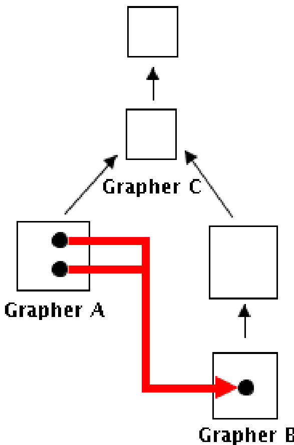
```
IlvHyperGrapher rootGrapher = new IlvHyperGrapher();
...
rootGrapher.addNode(node1, false);
IlvGraphicVector fromNodes = new IlvGraphicVector();
IlvGraphicVector toNodes = new IlvGraphicVector();
fromNodes.addElement(node1);
toNodes.addElement(node1);
IlvHyperEdge edge = new IlvHyperEdge(fromNodes, toNodes);
rootGrapher.addHyperEdge(edge, false);
// the edge is a normal hyperedge
...

IlvHyperGrapher subGrapher = new IlvHyperGrapher();
rootGrapher.addNode(subGrapher, true);
```

```
subGrapher.addNode(node2, false);
grapher.applyToObject(edge,
  new IlvApplyObject() {
    public void apply(IlvGraphic obj, Object arg) {
      ((IlvHyperEdge)obj).addFrom(node2);
    }
  }, arg, redraw);
// now, the edge is an intergraph hyperedge, since it ends at
// node1 and node2 but both are in different graphers
...
```

Note: If the hyperedge is contained in a hypergraph, you can add source or target nodes to the hyperedge only if these nodes belong to the same hypergraph as the hyperedge, or to subgraphs nested inside the hypergraph to which the hyperedge belongs. In all other cases you should remove the hyperedge from its hypergraph and then add the new sources and new targets. Finally, add the hyperedge back to a hypergraph as described in *Adding intergraph hyperedges*.

Adding intergraph hyperedges

Since the intergraph hyperedge has sources and targets in different hypergraphs, the question arises in which hypergraph the hyperedge is stored. As for intergraph links, an intergraph hyperedge must be stored in the first common ancestor of all its end nodes.



Intergraph hyperedge in a hierarchy of hypergraphs

In the above figure, the red intergraph hyperedge that connects the objects of A and B must be stored in hypergraph C, the first common ancestor of A and B.

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

```
static IlvHyperGrapher getLowestCommonHyperGrapher(IlvHyperEdge edge)
```

The following code example shows how to create an intergraph hyperedge.

Creating an intergraph hyperedge

```
...
IlvHyperEdge edge = new IlvHyperEdge(fromNodes, toNodes);
IlvHyperGrapher common = IlvHyperGrapher.getLowestCommonHyperGrapher(edge);
common.addHyperEdge(edge, false);
```

The static utility method `IlvHyperGrapher.addInterGraphHyperEdge` allows you to add the edge directly to the common parent hypergraph. Therefore, this code example can be shortened as follows:

Short code for creating an intergraph hyperedge

```
...
IlvHyperEdge edge = new IlvHyperEdge(fromNodes, toNodes);
IlvHyperGrapher.addInterGraphHyperEdge(edge, false);
```

Accessing intergraph hyperedges

The `IlvHyperGrapher` class provides methods that let you access in an efficient way normal hyperedges and intergraph hyperedges stored in a hypergraph. These methods are shown in the following table:

Methods for accessing normal and intergraph hyperedges

Method	Description
`getHyperEdges()`	Returns all normal hyperedges stored in the hypergraph.
`getHyperEdgesCount()`	Returns the number of normal hyperedges stored in the hypergraph.
`getInterGraphHyperEdges()`	Returns all intergraph hyperedges stored in the hypergraph.
`getInterGraphHyperEdgesCount()`	Returns the number of intergraph hyperedges stored in the hypergraph.

Since intergraph hyperedges are stored in the same way as other graphic objects in the hypergraph or graph, they are also included in the list of all objects contained in the hypergraph that is returned by the `getObjects()` method of the class `IlvManager`:

```
IlvGraphicEnumeration getObjects()
```

Calling the specific methods shown in the above table is much more efficient than traversing all the objects in a hypergraph.

Advanced use of hypergraphs

Making sure that hyperedges have a minimum number of ends

In contrast to normal links, hyperedges can have zero source nodes and zero target nodes. Such an empty hyperedge can be added to a hypergraph. This feature is generally useful, since you can decide to add sources and targets later.

An empty hyperedge is invisible and cannot be moved. Its position is fixed at `(0,0)`.

For many applications a hyperedge with 0 or 1 end only makes no sense, because these hyperedges are usually invisible.

If hyperedges with fewer than `k` ends are not allowed in your application, you can call the method:

```
hypergrapher.setMinHyperEdgeEndCount(k);
```

As a result, all hyperedges with fewer than the minimum number of hyperedge ends are removed from the hypergraph. When a new hyperedge is added to the hypergraph, it must already have the minimum number of ends. If not, the method `addHyperEdge` will fail with an exception.

When an end is removed from a hyperedge, the hyperedge is automatically removed from the hypergraph if the number of remaining ends is smaller than the minimum. By default, the minimum is 2.

Note: There is the following difference between `IlvHyperEdge` and `IlvLinkImage`:

When a node is removed from a graph or hypergraph, all its incident links (`IlvLinkImage`) are also always removed from the graph.

All the hyperedges (`IlvHyperEdge`) incident to a node removed from a hypergraph lose only the ends that point to that node. Hyperedges remain in the hypergraph unless the number of their remaining ends is smaller than the minimum.

Index

A

accessing
    hyperedges **53**
autoconnect mode
    hyperedges **42**

C

colinear
    segments **39**
complete
    segment tree **42**
constraints
    fixed angle segments **39**
    variable angle segments **39**

E

empty hyperedges
    visibility and mobility **54**
end segment **35**

F

fixed angle
    segments **37**
fixed angle segments
    constraints **39**
    testing **37**

G

graphs
    nested **48**

H

hyperedges
    accessing efficiently **53**
    autoconnect mode **42**
    creating intergraph **51**
    empty **54**
    intergraph **49**
    minimum number of ends **54**
    shape **34**
    storing intergraph **51**
hypergraphs
    nested **48**

I

`IlvDefaultHyperEdgeEnd` class **22**
`IlvFreeLinkConnector` class **22**
`IlvGrapher` class **6, 10, 48**
`IlvGraphic` class **18, 27**
`IlvHyperEdge` class **10, 12, 14, 18, 23, 34, 49**
`IlvHyperEdgeCenterConnector` class **31**
`IlvHyperEdgeConnector` class **23, 25, 27, 28**
`IlvHyperEdgeEnd` interface **14**
`IlvHyperEdgePinConnector` class **27, 28**
`IlvHyperGrapher` class **6, 10, 18, 48, 51, 53**
`IlvHyperGrapherPin` class **28**
`IlvLinkConnector` class **23, 27**
`IlvLinkImage` class **10, 12, 14, 18, 22, 34**
`IlvManager` class **6, 53**
`IlvSegmentedHyperEdge` class **34, 35, 37, 41, 42**
`IlvSegmentedHyperEdge.Segment` class **41**
`IlvSegmentedHyperEdge.Segment` inner class **35**
incident
    segment to node **35**
    segments **37**
incomplete
    segment tree **42**
incomplete segment tree
    primitive segment operations **41**
inner segments **35**
intergraph
    creating hyperedges **51**
    hyperedges **49**
    links **48**
    storing hyperedges **51**
    testing hyperedges **49**

© Copyright IBM Corp. 1987, 2009

L

links
    intergraph **48**

M

minimum
    hyperedge ends **54**

N

nesting
    graphs **48**
    hypergraphs **48**

O

operations
    primitive segment **41**
    segment **41**
orthogonal
    segments **37**

P

primitive segment
    operations **41**
    operations for changing to incomplete segment tree **41**

Q

querying
    segment angle **37**

S

segment
    operations **41**
segment tree
    complete **41, 42**
    incomplete **41, 42**
    undirected **35**
    unrooted **35**
segmented hyperedge
    shape **34, 35**
segments
    colinear **39**
    fixed angle **37**
    incident **37**
    incident to a node **35**
    inner **35**
    orthogonal **37**
    querying angle **37**
    variable angle **37**
shape
    of hyperedge **34**
    segmented hyperedge **34, 35**

T

testing
    fixed angle segment **37**
    intergraph hyperedges **49**
    variable angle segment **37**

U

undirected
    segment tree **35**
unrooted
    segment tree **35**

V

variable angle
    segments **37**
variable angle segments
    constraints **39**
    testing **37**