

# yFiles Developer's Guide

---

## yFiles Developer's Guide

Published by yWorks GmbH, the diagramming company.

yWorks GmbH  
Vor dem Kreuzberg 28  
72070 Tübingen  
Germany

email: <[contact@yworks.com](mailto:contact@yworks.com)>  
WWW: <http://www.yworks.com/>

Copyright © 2004-2007 yWorks GmbH

All rights reserved. No part of this publication may be reproduced, stored in a retrieval system, or transmitted, in any form or by any means, electronic, mechanical, photocopying, recording, or otherwise, without the prior written permission of the publisher.

All names and trademarks mentioned in this publication are the property of their respective owners.

---

---

---

## Table of Contents

Changes From Previous Edition .....	xi
Developer's Guide Enhancements .....	xi
New Features of yFiles 2.5 .....	xi
General .....	xii
1. Preface .....	1
Diagrams .....	1
Benefits and Problems .....	1
From Diagrams to Graphs .....	2
Graph-related Problems .....	3
About yFiles .....	4
What it is .....	4
What it's not .....	5
Application Domains .....	6
2. Introducing the yFiles Library .....	9
yFiles Library Components .....	9
yFiles Basic .....	9
yFiles Viewer .....	9
yFiles Layout .....	9
How to Read This Developer's Guide .....	10
First-Time Users .....	10
Experienced Users .....	10
Where to Find Things in yFiles .....	12
3. General Concepts in yFiles .....	13
Basic Graph Structure .....	13
Clear Responsibilities .....	14
Accessing Graph Elements .....	14
Storing Elements .....	14
Binding Data to Graph Elements .....	15
A Word on Subclassing Graph Elements .....	15
4. Working With the Graph Structure .....	17
Creating Graphs and Graph Elements .....	17
Graph Structure Functionality .....	18
More About Class Graph .....	18
Graph Elements .....	20
Complexity .....	22
Advanced Topics .....	23
Events and Listeners .....	25
Structural Changes .....	25
Iteration Mechanisms .....	26
Iteration and Element Removal .....	28
Alternative Iteration Techniques .....	28
Containers .....	30
High-Level Features .....	30
Low-Level Features .....	31
Binding Data to Graph Elements .....	32
Maps and Data Providers .....	32
Notes .....	35
Analyzing Graphs .....	37
Quickly Checking for Graph Characteristics .....	37
Advanced Graph Characteristics .....	37
5. Automatic Graph Layout .....	47
Layout Algorithms .....	47
Advanced Layout Features .....	47
Layout Architecture .....	48
Graph Structure Enhancements .....	48
Advanced Topics .....	50
Layout Infrastructure .....	51
Concepts .....	52
The Layout Stages Concept .....	52

Default Compound Layout Process .....	53
Using Buffered Layout .....	53
Providing Supplemental Layout Data .....	55
Advanced Layout Features .....	56
Hierarchically Organized Graphs .....	56
Port Constraints .....	59
Port Candidates .....	61
Edge/Port Grouping (Bus-style Edge Routing) .....	63
Layout Stages .....	65
Tutorial Demo Code .....	65
Class ComponentLayouter .....	67
Supplemental Layout Data .....	67
Layout Options .....	67
Advanced Layout Techniques .....	68
Class GraphLayoutLineWrapper .....	70
Layout Options .....	71
Major Layout Algorithms .....	72
Circular Layout .....	72
Hierarchical Layout .....	73
Organic Layout .....	73
Orthogonal Layout .....	74
Tree Layout .....	74
Incremental Layout .....	76
Use Cases .....	76
Circular Layout .....	78
Supplemental Layout Data .....	78
Layout Options .....	78
Partition Arrangement Options .....	79
Tree Arrangement Options .....	80
Compact Layout .....	81
Tutorial Demo Code .....	82
Hierarchical Layout .....	83
Supplemental Layout Data .....	83
Layout Options .....	84
Drawing Style Options .....	84
Layer Assignment Options .....	87
Node Order Options .....	90
Advanced Layout Features .....	90
Package Structure .....	93
Tutorial Demo Code .....	94
Hierarchical Layout of Hierarchically Organized Graphs .....	95
Supplemental Layout Data .....	95
Layout Options .....	96
Tutorial Demo Code .....	96
Incremental Hierarchical Layout .....	97
Specifying Hints .....	98
Supplemental Layout Data .....	99
Layout Options .....	100
Drawing Style Options .....	101
Advanced Layout Features .....	101
Related Classes .....	103
Layout of Hierarchically Organized Graphs .....	103
Tutorial Demo Code .....	104
Organic Layout .....	106
Supplemental Layout Data .....	106
Layout Options .....	107
Advanced Layout Features .....	110
Tutorial Demo Code .....	111
Smart Organic Layout .....	112
Supplemental Layout Data .....	112
Layout Options .....	112
Advanced Layout Features .....	113
Quality Versus Time .....	113
Compactness .....	113

---

Node Overlap Removal .....	114
Interactive Organic Layout .....	115
General Usage .....	115
Supplemental Layout Data .....	118
Layout Options .....	118
Tutorial Demo Code .....	119
Orthogonal Layout .....	121
Supplemental Layout Data .....	121
Layout Options .....	121
Advanced Layout Features .....	123
Tutorial Demo Code .....	123
Orthogonal Layout of Hierarchically Organized Graphs .....	124
Supplemental Layout Data .....	124
Layout Options .....	125
Tutorial Demo Code .....	125
Directed Orthogonal Layout .....	126
Supplemental Layout Data .....	126
Layout Options .....	127
Advanced Layout Features .....	127
Tutorial Demo Code .....	128
Compact Orthogonal Layout .....	129
Layout Options .....	129
Tutorial Demo Code .....	129
Tree Layout .....	130
Enhancing the Layout Process .....	130
Directed .....	130
Supplemental Layout Data .....	131
Layout Options .....	131
Advanced Layout Features .....	132
Balloon .....	133
Layout Options .....	133
Advanced Layout Features .....	134
Horizontal/Vertical .....	134
Supplemental Layout Data .....	134
Layout Options .....	135
Compact .....	135
Supplemental Layout Data .....	135
Layout Options .....	136
Generic Tree Layout .....	137
Supplemental Layout Data .....	137
Layout Options .....	137
Advanced Layout Features .....	141
Tutorial Demo Code .....	142
Edge Routing Algorithms .....	143
Organic Routing .....	143
Orthogonal Routing .....	143
Organic Edge Routing .....	145
Routing Options .....	145
Tutorial Demo Code .....	145
Orthogonal Edge Routing .....	147
Supplemental Layout Data .....	148
Routing Options .....	148
Advanced Routing Features .....	151
Enhancing the Routing Process .....	151
Tutorial Demo Code .....	151
Channel Edge Routing .....	153
Supplemental Layout Data .....	153
Routing Options .....	154
Advanced Routing Features .....	154
Related Classes .....	155
Tutorial Demo Code .....	155
Automatic Label Placement .....	156
Labeling Scenarios .....	156
Concepts .....	156

Label Models .....	157
General Labeling .....	158
Integrated Labeling .....	160
Tutorial Demo Code .....	160
6. Displaying and Editing Graphs .....	161
Model-View-Controller Paradigm .....	161
Graph Structure Enhancements .....	162
Class Graph2D .....	162
Advanced Topics .....	163
Bringing Graph Elements to Life: The Realizer Concept .....	164
Binding Realizers .....	164
General Features .....	165
Node Realizers .....	166
Edge Realizers .....	170
Realizer-Related Features .....	178
Label Support .....	178
Related Classes .....	184
Selection State .....	187
Writing Customized Realizers .....	189
Subclassing and Overriding .....	189
Providing Interface Implementations .....	191
Swing User Interface Components as Node Realizers .....	195
Rendering .....	195
Editor Functionality .....	196
Tutorial Demo Code .....	196
View Implementations .....	197
Class Graph2DView .....	197
Class Overview .....	204
Related Classes and Interfaces .....	205
User Interaction .....	208
Mouse-Driven Actions .....	208
Keyboard-Driven Actions .....	208
Label Editor .....	209
Node-Related Editing Capabilities .....	210
EditMode Workings .....	211
Class EditMode .....	212
Class NavigationMode .....	215
Class CellEditorMode .....	215
Class MagnifierViewMode .....	216
Related Classes and Interfaces .....	216
Customizing anEditMode .....	218
Events and Listeners .....	221
Graphical Property Changes .....	221
Selection State Changes .....	221
Printing a Graph's Visual Representation .....	223
Scaling and Clipping .....	223
Title Bar and Footer Support .....	224
Printing Preview .....	224
Creating a Dedicated Printing View .....	224
Tutorial Demo Code .....	225
Animations for Graph Elements .....	226
Animation Framework .....	226
Callback Mechanism With Generic Animations .....	229
Tutorial Demo Code .....	230
Advanced Application Logic .....	231
Undo/Redo .....	231
Clipboard .....	232
Layout Morphing .....	232
7. Graph Hierarchies .....	235
Concepts .....	235
Grouping and Nesting .....	235
Special Proxy Elements .....	235
Working With Graph Hierarchies .....	237
Comparing the Concepts .....	237

Nesting Characteristics .....	237
Managing Graph Hierarchies .....	239
Class HierarchyManager .....	239
Class DefaultHierarchyGraphFactory .....	241
Tutorial Demo Code .....	242
Node Realizers .....	243
Class GroupNodeRealizer .....	243
Interface AutoBoundsFeature .....	244
View Implementations .....	245
Class HierarchyJTree .....	245
Related Classes .....	245
User Interaction .....	247
Class HierarchyEditMode .....	247
Tutorial Demo Code .....	248
Events and Listeners .....	249
Hierarchy Changes .....	249
Automatic Layout .....	250
Class GroupLayoutConfigurator .....	250
8. Input and Output .....	251
Graph Exchange and Graph Export Formats .....	251
Input/Output Handlers .....	252
Reading and Writing Graph Structure Data .....	255
YGF .....	256
Common Usage .....	256
Extending YGF With Custom Data .....	256
Tutorial Demo Code .....	258
GML .....	259
What it Looks Like .....	259
Common Usage .....	260
Extending GML With Custom Data .....	260
Tutorial Demo Code .....	260
GML File Format .....	260
XGML .....	271
What it Looks Like .....	271
Common Usage .....	271
TGF .....	272
What it Looks Like .....	272
Common Usage .....	272
Exporting a Graph's Visual Representation .....	274
Preparing the Graph .....	275
Using GIF and JPG .....	277
Using the Java Image I/O API .....	278
Image Tiling .....	278
Image Maps .....	279
yFiles Extension Packages .....	281
GraphML .....	281
ySVG .....	281
yExport .....	281
9. Settings Dialogs and Components .....	283
Overview .....	283
Model-Delegate Pattern .....	283
Basic Functionality .....	285
Creating Model and View .....	285
Synchronizing Editors and Option Items .....	287
Defining Dependencies Between Option Items .....	288
Undefined Values in Option Items .....	290
Internationalization and Localization .....	290
Serialization of Settings .....	292
Tutorial Demo Code .....	293
Advanced Topics .....	294
Synchronizing Multiple Editors .....	294
Tutorial Demo Code .....	294
10. GraphML Extension Package .....	295
GraphML .....	295

What GraphML Looks Like .....	295
GraphML Default Extension Mechanism .....	295
Customizing the GraphML Extension Mechanism .....	296
Using the yFiles GraphML Extension Package .....	299
Direct Support for Simple Data Types .....	299
General Support for Structured Data .....	300
Support for Custom Realizer Implementations .....	304
yFiles GraphML Post-processors .....	306
Tutorial Demo Code .....	307
Advanced Topics .....	309
yFiles XSLT Support for GraphML .....	309
Reading and Writing Graphs of Arbitrary Type .....	309
11. ySVG Extension Package .....	311
Scalable Vector Graphics .....	311
Using the ySVG Extension Package .....	312
Using SVG Content .....	312
Customizing SVG Content .....	313
Tutorial Demo Code .....	313
GraphML Support .....	314
12. yExport Extension Package .....	315
Vector Graphics File Formats .....	315
Using the yExport Extension Package .....	316
Using the System Clipboard .....	316
A. Obfuscation .....	319
Why Obfuscation Matters .....	319
Name Obfuscation and its Benefits .....	319
Name Obfuscation Using yGuard .....	320
Adjusting Names .....	320
Checking Obfuscation Success .....	320
Tutorial Demo Code .....	321
Glossary .....	323
Index .....	327

---

# Changes From Previous Edition

This edition of the yFiles Developer's Guide relates to major version 2.5 of the yFiles graph visualization library.

## Developer's Guide Enhancements

### New Features of yFiles 2.5

yFiles Basic:

- new generic mechanism for copying graph structures together with any data associated with the graph elements

yFiles Layout:

- new layout stage class `GraphLayoutLineWrapper` which allows to "line-wrap" and "column-wrap" graph layouts. Either width (height) or aspect ratio for the resulting line-wrapped (column-wrapped) diagram can be specified.
- class `IncrementalHierachicLayouter`: added support for hierarchically organized graphs
- new orthogonal layout algorithm which creates compact layouts that fit into a specified aspect ratio
- new orthogonal edge routing algorithm that uses a two phase approach to find edge routes
- new layerer implementation for hierarchical layout that enables user-defined constrained layering. Nodes can be placed into the topmost or bottommost layer, or can be placed into the same layer or a layer preceding/following that of a given reference node.
- class `OrthogonalEdgeRouter`: new support for port candidates
- class `SmartOrganicLayouter`: improved support for processing only a subset of nodes in the presence of unconnected components
- new so-called graph copy factory implementations that use the generic mechanism for copying graph structures together with any data associated with the graph elements

yFiles Viewer:

- class `EditMode` and its associated child modes provide new support for orthogonal edge paths, which includes edge creation, moving edge segments, creating new bends/edge segments, and moving edge ports. Furthermore, resizing nodes and moving nodes/bends have also been modified to take orthogonal edge paths into account. Part of this support is new view mode class `OrthogonalMoveBendsMode`, and enhancements for view mode classes `CreateEdgeMode`, `MovePortMode`, `HotSpotMode`, and `MoveSelectionMode`.
- class `YLabel`: new support for so-called "configurations", i.e., sets of interface implementations which enable convenient customization of label look and feel. New predefined label configurations for `NodeLabel` and `EdgeLabel` provide automatic line-wrapping of a label's text.
- class `Graph2DPrinter`: new support for adding a footer to printed diagrams. The footer is freely customizable.
- new view mode class `MagnifierViewMode` which provides a magnifying glass "tool" that enables local zooming in a `Graph2DView`
- class `DefaultGraph2DRenderer`: new support for so-called "layered painting" which allows to establish arbitrary rendering orders on graph elements
- new class `HierarchyTreeTransferHandler` which adds drag-and-drop functionality to `HierarchyJTree` which allows changing the graph hierarchy using simple mouse drag gestures
- class `Graph2DView`: new support for extensive customization of selection box look
- class `HitInfo`: new constructors for hit-testing with focus on only a specified type of graph elements

- class Graph2DViewActions: improved support for installing default keyboard actions with a [Graph2DView](#)
- class Graph2DView: new [client property](#) that allows to specify the scroll bar unit increment
- new so-called graph copy factory implementations that use the generic mechanism for copying graph structures together with any data associated with the graph elements

## General

yFiles Layout:

- added table for routing algorithms that support the concept of port candidates
- added new section for layout stages, moved and enhanced the description of layout stage class [ComponentLayouter](#)
- moved and enhanced the description of hierarchical layout class [HierarchicGroupLayouter](#)
- reorganized the section on [Advanced Routing Features](#) of class [OrthogonalEdgeRouter](#) and added a [table of data provider look-up keys](#) that can be used with this class
- added class [ChannelEdgeRouter](#) to the tables of routing algorithms that support [port constraints](#) and [port candidates](#), and also to the table of routing algorithms that support [incremental routing](#)
- added classes [SmartOrganicLayouter](#), [TreeLayouter](#), and [BalloonLayouter](#) to the table of layout algorithms that support [incremental layout](#)
- complemented the descriptions of layout algorithms [OrganicLayouter](#), [SmartOrganicLayouter](#), [TreeLayouter](#), and [BalloonLayouter](#) with sections presenting their support for incremental layout
- enhanced the description of class [ChannelEdgeRouter](#) to include a [table of data provider look-up keys](#) that can be used with the default path finding and distribution strategies, and a section on [Advanced Routing Features](#)
- enhanced the descriptions for layout descriptor classes [NodeLayoutDescriptor](#) and [EdgeLayoutDescriptor](#), respectively, which are used by class [IncrementalHierarchicLayouter](#)

Extension Packages:

- updated the description of the GraphML extension package to GraphML version 3.0
- added chapter for yFiles extension package [ySVG](#) which brings support for Scalable Vector Graphics (SVG) to the yFiles graph visualization library
- added chapter for yFiles extension package [yExport](#) which brings support for popular vector graphics file formats, like PDF, SWF, EMF, and EPS to the yFiles graph visualization library
- removed references to the yWMF extension package. yWMF is replaced by yExport, which provides functionality for exporting to EMF file format, the successor of WMF.

# Chapter 1. Preface

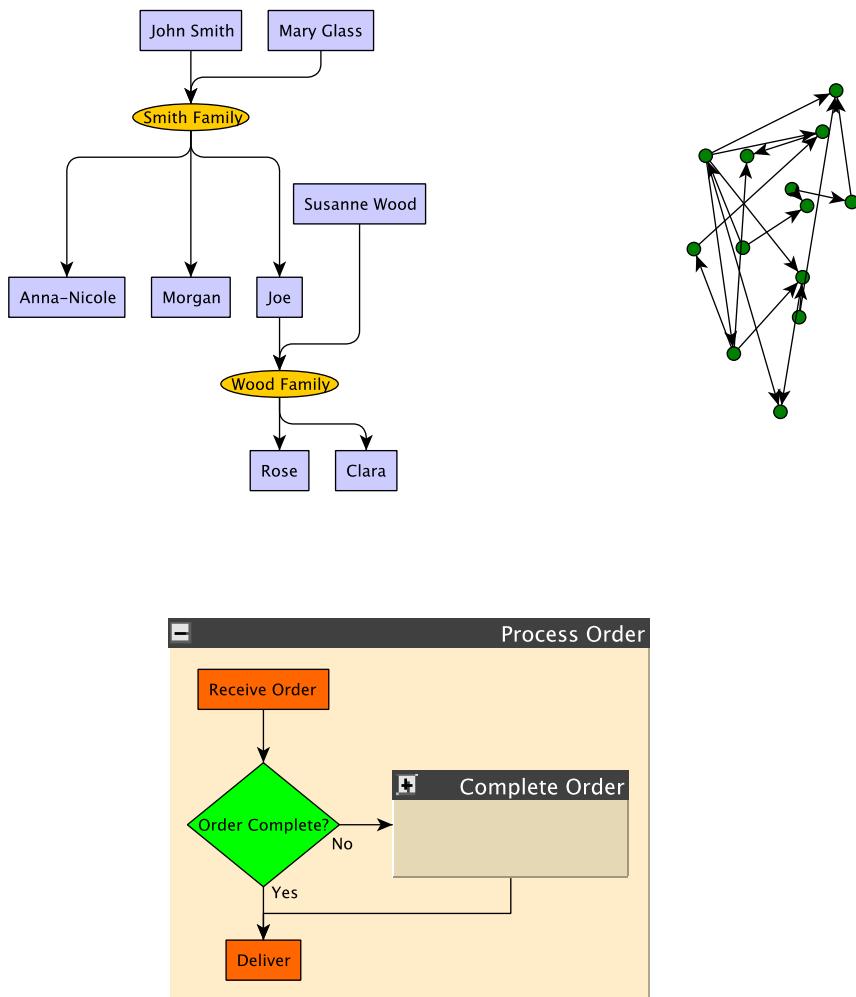
yFiles is an extensive class library that provides algorithms and components for analyzing, viewing, and drawing graphs, diagrams, and networks. It is entirely written in the Java™ programming language and can be used to develop sophisticated applications to be deployed on every platform with a working Java Runtime Environment installation.

This chapter gives an overview what kind of purposes the yFiles library serves, and what kind of applications and application domains are possible.

## Diagrams

Diagrams of any kind naturally arise in a wide variety of application domains. The diagrams encountered commonly comprise entities together with their relationships. Entities are usually represented by some kind of object, e.g., a colored rectangle, and the relationships are usually represented by a connecting line with or without some kind of decoration at its ends. [Figure 1.1, “Different kinds of diagrams” \[1\]](#) shows some examples of diagrams.

**Figure 1.1. Different kinds of diagrams**



## Benefits and Problems

"A diagram is worth a thousand words." is a popular proverb which best expresses the benefits of using diagrams:

- Subjects of arbitrary complexity can be arranged to get a clear picture of hidden structures and dependencies.
- Gordian relationships can be clarified in a matter of seconds.
- Information is communicated to a broad audience easily.

However, these properties strongly depend on the quality of a diagram. And the task of making legible diagrams is often tedious and moreover error-prone when done by hand. Sketching even small data sets can become awkward enough to forsake entirely. For instance, the second diagram from [Figure 1.1, “Different kinds of diagrams” \[1\]](#) shows a rather incomprehensible example.

With large data sets the problem of making high-quality, easy-to-understand diagrams aggravates. Doing it by hand is no longer an alternative, so the need for an automatic generation grows.

## From Diagrams to Graphs

Diagrams in the common sense, i.e., some sort of entities together with connecting lines as described above, are closely related to the mathematical notion of a "graph." Basically, this means that diagrams and graphs serve the same purpose, the two concepts are interchangeable. Hence, a question that arises in the context of a diagram is equally a question concerning a graph.

Formally, a graph is defined as a tuple of two sets, a node set and an edge set. A "node" in this definition is the equivalent of an entity, while an "edge" represents a connecting line, i.e., a relationship between entities.

There is a variety of terms used to refer to the mathematically coined "node" and "edge," however in the course of this document we will stick to the original names as often as possible. See the [glossary](#) for more graph structure terminology widely used in this documentation.

## **Graph-related Problems**

The most common tasks to be performed with graphs boil down to a handful of things:

- visualization
- (manual) creation
- editing
- automatic entity arrangement, i.e., automatic graph layout
- structural analysis

*Visualization* is the task of actually generating a visual representation of an existing graph. Note that this does not necessarily mean the creation of a static picture, but more generally describes forms of representation that could also allow for interaction. The items *(manual) creation* and *editing* both mean interacting with a graph, i.e., creating nodes and connecting them with edges. Or adding further elements to an existing graph, or removing elements from an existing graph. *Automatic entity arrangement* is the creation of high-quality, easy-to-understand drawings of a graph to help in communicating information more quickly. *Structural analysis* means solving advanced questions that relate to a graph structure, for instance, determining a shortest path between two nodes.

The yFiles graph visualization library supports all the above and has even more to offer. yFiles is something like a Swiss Army knife that:

- Helps the professional Java programmer building high-quality, graph structure related applications.
- Helps the serious graph worker realizing demanding projects in the Java programming language.

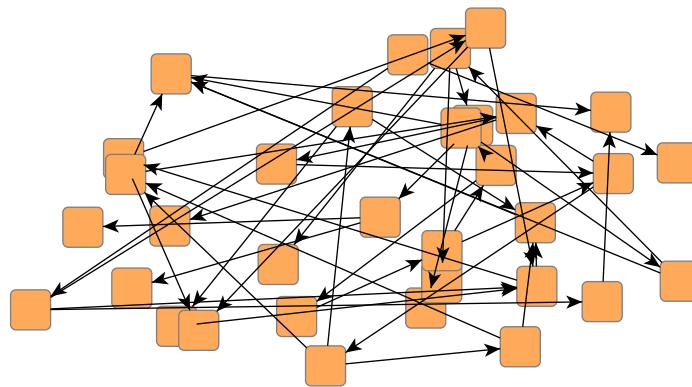
## About yFiles

yFiles is a Java class library that provides algorithms and components for analyzing, viewing, and drawing graphs, diagrams, and networks. Its intended use is for application developers that are working on projects dealing with graph-related structures.

### What it is

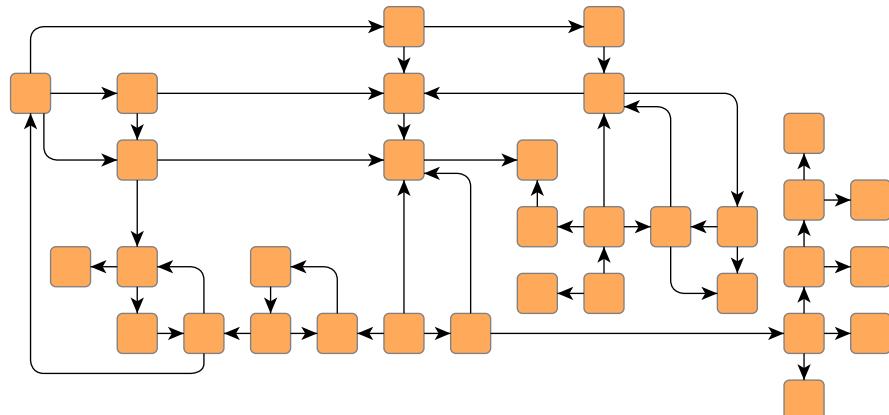
One of the main aspects of yFiles is the provision of sophisticated layout algorithms to support automatic generation of high-quality graph drawings. Take, for instance, the graph depicted in [Figure 1.2, “Initial setup of some data” \[4\]](#). It shows a “raw” version of a graph where it seems that the nodes have been placed randomly on the plane. Suppose this “raw” graph represents an initial setup of some data.

**Figure 1.2. Initial setup of some data**



By using one of yFiles's layout algorithms the “raw” graph will be transformed into the arrangement shown in [Figure 1.3, “Automatically computed “polished” drawing” \[4\]](#). Note that the actual process of computing the “polished” drawing is completely automatic.

**Figure 1.3. Automatically computed “polished” drawing**



### Note

In a real-world scenario the initial setup could be even worse. For instance, when a graph is created from data retrieved from a database, any node created to represent an

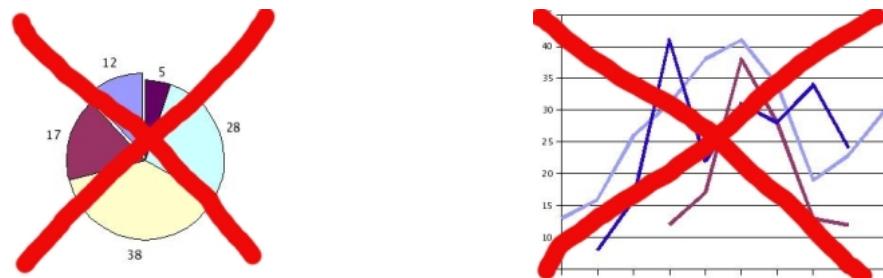
entity from a column would not have *any* positional information associated. Hence, the created nodes would all lie at the origin of the Cartesian coordinate system, i.e., they would "pile up."

### What it's not

Even for mathematically inclined people the term *graph drawing* can be misleading as to what it exactly should mean. Also, the term *visualization* is another popular occasion for a misunderstanding of the purposes the yFiles library serves.

**Figure 1.4, “Other forms of diagrams” [5]** shows some examples of other forms of graphs, or more generally diagrams, that yFiles does not support. In particular, the yFiles graph visualization library does not provide any functionality to draw pie charts, function plots, or anything the like.

**Figure 1.4. Other forms of diagrams**

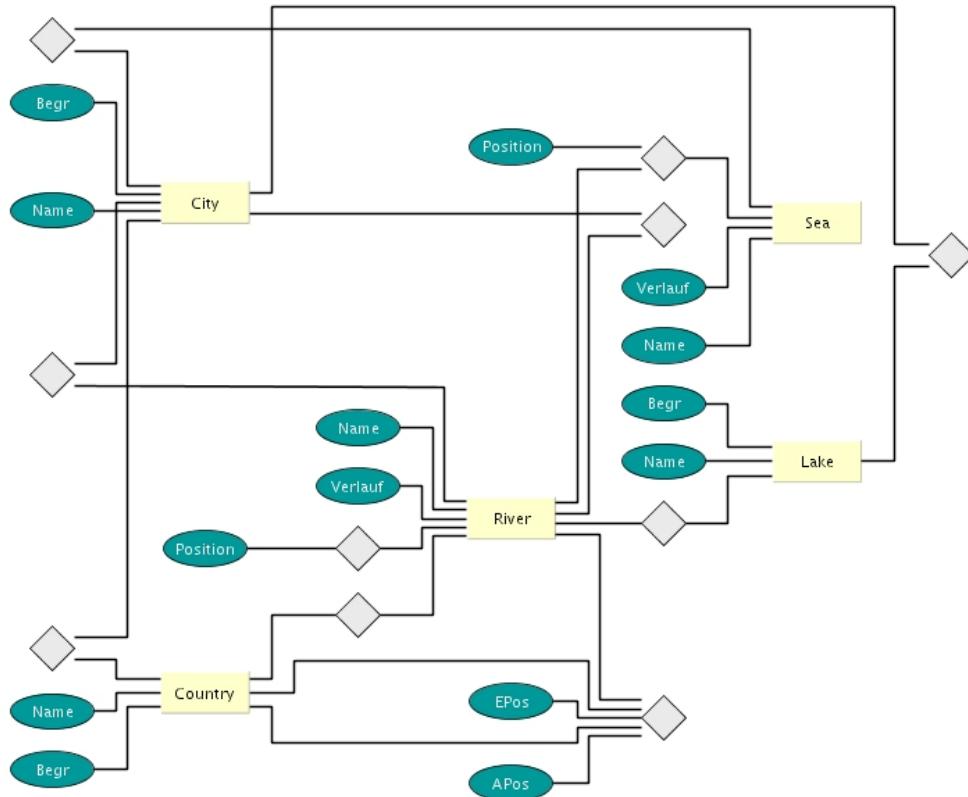


## Application Domains

The yFiles library offers sophisticated functionality that can help in a variety of problem fields. Following are some example drawings of diagrams from different kinds of application domains. All of these drawings have been computed automatically by layout algorithms from the yFiles library.

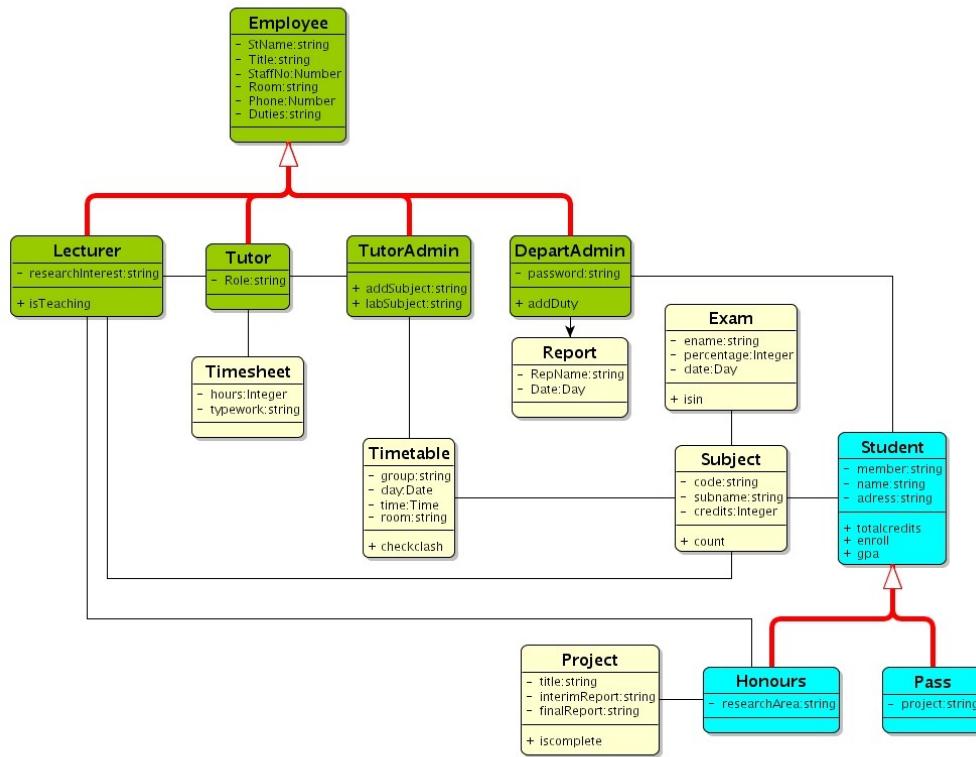
In database design as well as database management and administration Entity-Relationship diagrams (E-R diagrams) are used to sketch and to visualize entities together with their relations. Figure 1.5, “Entity-Relationship diagram” [6] shows an example of an E-R diagram.

**Figure 1.5. Entity-Relationship diagram**



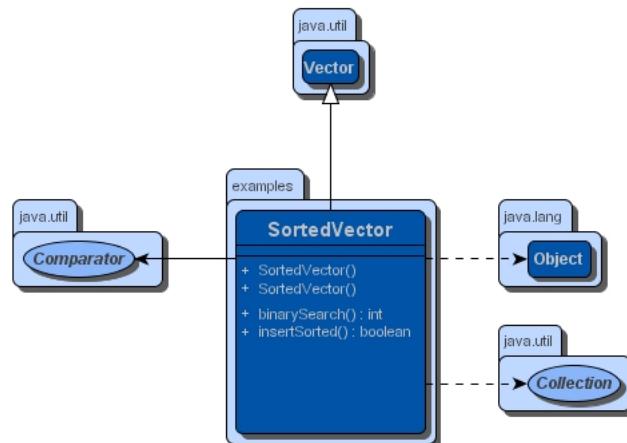
In software engineering the Unified Modeling Language (UML) is widely recognized as the standard to model the relationships between classes. Figure 1.6, “Software engineering” [6] presents a high-quality UML diagram of a mid-sized project.

**Figure 1.6. Software engineering**



The UML notation is also used in software documentation. yDoc, for instance, a Javadoc™ extension that can embed UML diagrams into API documentations, uses the yFiles library to automatically generate drawings that reflect the class and/or package dependencies inherent to a Java project. [Figure 1.7, “Software documentation”](#) [7] shows the UML diagram of a small excerpt of such a project.

**Figure 1.7. Software documentation**



---

---

# Chapter 2. Introducing the yFiles Library

This chapter presents the different yFiles components, briefly discusses possible ways of reading the Developer's Guide depending on a reader's knowledge, and lastly gives an overview on the packages and their provided functionality.

## yFiles Library Components

The functionality that the yFiles library provides is divided into three parts:

- [Basic](#), which serves as the "backbone" for the main part of the library
- [Viewer](#) for everything relating to user interaction
- [Layout](#), which provides highly evolved automatic layout algorithms

### yFiles Basic

yFiles Basic contains essential classes and data types for graph analysis tasks. It provides efficient implementations of advanced data types like graph or priority queue. It furthermore makes a wide variety of graph and network algorithms available which form an indispensable toolkit for any network analysis tasks.

### yFiles Viewer

yFiles Viewer builds upon yFiles Basic. It provides a powerful graph viewer component and other Swing-based GUI elements. The viewer component is showcased in the yEd graph editor application. Other notable features of yFiles Viewer are its support for diverse graph formats (e.g. GML, YGF, JPG, GIF, etc.), and its printing capabilities.

### yFiles Layout

yFiles Layout builds upon yFiles Basic. It provides a perfect suite of graph layout algorithms which offer unequaled opportunities. A multitude of layout styles like hierarchical, orthogonal, or circular are provided as easy-to-integrate components which can be configured programmatically to suit most layout demands. Additionally, yFiles provides edge routing algorithms that make it possible to route edges into existing diagrams easily.

# How to Read This Developer's Guide

## First-Time Users

Clearly, anybody new to graph drawing and/or the yFiles graph visualization library should read the entire Developer's Guide to learn all about yFiles.

Without knowledge of the benefits of (automatic) graph drawing, be sure to not miss out on "[Preface](#)." Then proceed with the chapters on yFiles Basic (namely, "[General Concepts in yFiles](#)" and "[Working With the Graph Structure](#)") to know about the concepts, the graph structure implementation, and how to use it. Together, these are the most important chapters to understand and remember.

The parts on yFiles Layout ("[Automatic Graph Layout](#)") and yFiles Viewer ("[Displaying and Editing Graphs](#)" and "[Input and Output](#)") are independent of each other and therefore can be read in any order. Accordingly, any of these parts can be omitted if the respective library component is not of interest.

The remaining chapters and appendices can equally be skipped the first time. Note nonetheless that they provide essential information for the interested reader. In contrast, the [glossary](#) cannot be consulted often enough. It explains common graph structure related terminology as well as all terms specifically used with yFiles.

## Experienced Users

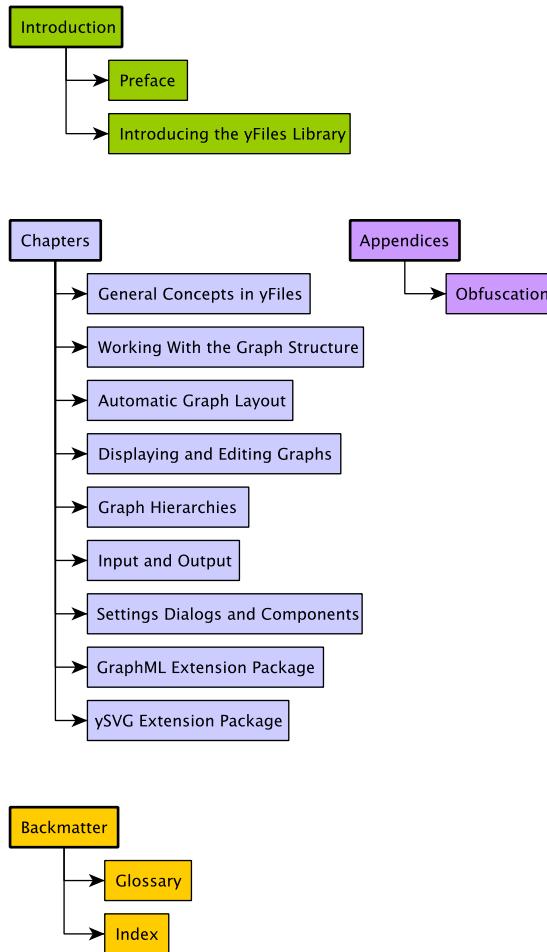
Except maybe the section on graph analysis algorithms at the end of "[Working With the Graph Structure](#)," experienced yFiles users can easily skip these chapters. (However, it may also be true that even experienced users will find some of the sections interesting enough to dwell upon... "[Binding Data to Graph Elements](#)" might be such a candidate.)

The parts on yFiles Layout ("[Automatic Graph Layout](#)") and yFiles Viewer ("[Displaying and Editing Graphs](#)" and "[Input and Output](#)") should be read carefully and often. "Automatic Graph Layout" provides in-depth information concerning all layout algorithms, while "Displaying and Editing Graphs" covers (nearly) every aspect relating to user interaction and visual representations. The chapter on input and output file formats can be read independently of the other Viewer chapter.

"[Graph Hierarchies](#)" is a must-read particularly for everyone doing nested graphs. It explains the mechanism and the preparations necessary for both the layout and the viewer side.

The chapter on the settings framework, namely "[Settings Dialogs and Components](#)," explains the yFiles classes that provide a convenient way to quickly create settings dialogs and components.

**Figure 2.1. Developer's Guide structure**



Independent of experience with the yFiles library, it is strongly advised to read the appendix on "Obfuscation", in order to understand the code protection requirements of the yFiles license terms and how to fulfill them properly.

## Where to Find Things in yFiles

yFiles is an extensive class library that offers many sophisticated features. To help in finding the classes that are responsible for the provision/the implementation of certain features, [Table 2.1, “What Goes Where” \[12\]](#) gives a mapping from functionality to library package(s) containing the actual classes.

**Table 2.1. What Goes Where**

Functionality	yFiles Library Component	Java Package Name
Basic Graph Structure	Basic	y.base
Containers and Iteration	Basic	y.base
Graph Analysis	Basic	y.algo
Edge Routing	Layout	y.layout.router
Graph Structure Augmentation (Layout)	Layout	y.layout
Labeling	Layout	y.layout.labeling
Layout Algorithms	Layout	y.layout
Nested Graphs	Layout, View	y.layout.grouping, y.view.hierarchy
Graph Structure Augmentation (Viewer)	View	y.view
Input and Output Formats	View	y.io, y.io.gml
Viewing and Editing Graphs	View	y.view
Settings Dialogs and Components	View	y.option

# Chapter 3. General Concepts in yFiles

This chapter introduces the general concepts of the yFiles software library. These include purpose-specific, i.e., graph structure related topics as well as practical programming aspects. Note that the concepts described are valid for each of the yFiles distribution packages.

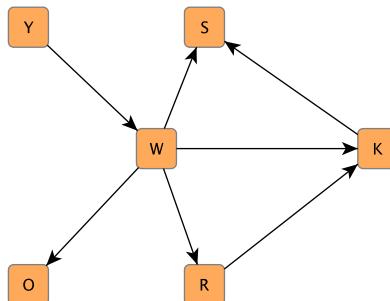
## Basic Graph Structure

All yFiles functionality centers around the mathematical notion of a directed graph. A directed *graph* is defined as consisting of two sets, a node set and an edge set. A *node* represents any kind of entity and an *edge* represents a relation between any two nodes from the node set. *Directed* graph means that all edges from the edge set have direction, i.e., a distinct *source node* and a distinct *target node*. The mathematical notation for an edge is a pair where the first component denotes source and the second component denotes target node: edge  $e = (\text{source}, \text{target})$ .

This directed graph structure is reflected by the classes `Graph`, `Node`, and `Edge` in package `y.base`. Together, these classes form the foundation for all graph functionality.

[Figure 3.1, “A simple graph” \[13\]](#) shows the picture of an example graph with six nodes and seven edges where the nodes are represented by filled circles and the edges are drawn as straight lines between two nodes with an arrowhead at the target end indicating direction. The exact mathematical notation of this example graph would be: Graph  $G = (V, E)$ , with node set  $V = \{Y, W, O, R, K, S\}$  and edge set  $E = \{(Y, W), (W, O), (W, R), (W, K), (W, S), (R, K), (K, S)\}$ .

**Figure 3.1. A simple graph**



Some interesting aspects of the graph structure implementation should be observed. For example, there is no requirement that source and target node of a directed edge have to be different. When they are identical, the special kind of edge is also called “*self-loop*.” (A self-loop at the node with label “Y”, for example, would be written as  $(Y, Y)$ .)

On another note, there is also no restriction on the number of edges connecting the same two nodes from a graph. This means that the yFiles graph structure in fact provides not only a set but a multi-set to store the edges of a graph. We will call such edge sets “*parallel edges*.”

Another consequence with a directed graph is that edges connecting to a node can be distinguished into incoming edges and outgoing edges. *Incoming* edges are those that have the node as target, while *outgoing* edges are those that have the node as source. The set of all edges at a node is simply called its “edges.” [Figure 3.1, “A simple graph” \[13\]](#), for example, shows four outgoing edges and one incoming edge for the node labeled “W.”

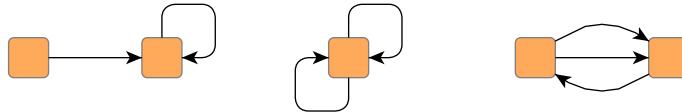
### Note

Despite the strong emphasis on the directedness of the graph structure it is nevertheless possible to mimic undirected behavior easily. Both the graph structure itself and many algorithms (where appropriate) offer methods to ignore edge direction entirely. (Effectively, this means that directedness/undirectedness ultimately is a matter of interpretation on the algorithm’s side.)

[Figure 3.2, “Various configurations” \[14\]](#) shows some graph excerpts that depict (from left to

right) a directed edge and a self-loop, several self-loops at one node, and a set of parallel edges.

**Figure 3.2. Various configurations**



## Important

Keep in mind that the presented figures are only a guidance to demonstrate the various possibilities provided by yFiles. The yFiles *basic* graph structure has neither positional information nor any visual representation associated with the graph elements.

## Clear Responsibilities

The yFiles graph structure implementation adheres to the idea that the graph should be the only instance in charge of all structural changes. (Structural changes most notably means node and edge creation and removal.) This paradigm has some important implications:

- There must exist a valid instance of type Graph to create any graph elements ("inside" this graph instance) at all.
- There is no way to create a node or an edge "outside" a graph.
- All created graph elements instantly belong to a distinct graph.

Additionally, there is a clear cascade of responsibilities for providing access to certain graph element information:

- A graph provides access to its nodes and edges.
- A node provides access to its edges; this can be all edges, or all incoming and outgoing edges, respectively.
- An edge provides access to its source and target node.

## Accessing Graph Elements

A graph provides access to its nodes and edges by means of bidirectional cursors. A cursor is a general concept to iterate over a sequence of objects one after the other. Bidirectional means that it is possible to both move forward and backward on the sequence.

The actual methods from class Graph return either a [NodeCursor](#) or an [EdgeCursor](#) instance to be used for iterating over the respective elements. Both NodeCursor and EdgeCursor present a read-only view on the actual graph elements. That way it is guaranteed that class Graph remains the single authority for any structural changes.

On a side note, observe that NodeCursor and EdgeCursor are typed variants of their common superinterface [YCursor](#). The typing has the advantage of additional safety, also there is no need to explicitly cast the iterated objects (as compared to `java.util.Iterator`, for example).

## Storing Elements

The yFiles library defines a base implementation of a doubly linked list, class [YList](#). Most notably however, are the typed descendants of YList, classes [NodeList](#) and [EdgeList](#). The latter two are used for the respective kinds of graph elements while their common superclass stores objects of arbitrary type. NodeList and EdgeList are frequently used as return types or as parameters in several methods throughout the yFiles library.

## Binding Data to Graph Elements

Most notably graph algorithms expect certain information associated with the graph elements they process. For example, a Shortest Path algorithm from package y.algo needs cost data associated with the edges of a graph.

The yFiles library uses the concept of data accessors to bind this information to graph elements. The general term "data accessor" comprises both storing data and retrieving data, i.e., setter methods (write behavior) and getter methods (read behavior). The actual implementations of these two aspects are interfaces [DataProvider](#) and [DataAcceptor](#). DataProvider defines read-only behavior on the contained data, its counterpart DataAcceptor defines write-only behavior.

The most frequently used data accessors, for example, interfaces [NodeMap](#) and [EdgeMap](#), have read/write behavior, i.e., they inherit from both DataProvider and DataAcceptor. However, there are many cases where read-only behavior is sufficient. This is especially true for situations where the supplemental data is given as a parameter to a method, for example.

In general, using data accessors presents a flexible approach, since the realization of the actual data store is not constrained to given implementations but can be extended in any imaginable way. Also, there is no restriction on the number or the implementation of each type of custom data associated with a graph element.

## A Word on Subclassing Graph Elements

Normally, i.e., in popular object-oriented belief, customizing existent functionality means using inheritance. The yFiles graph implementation, however, disfavors this approach for the sake of the same technique used to provide supplemental data to graph elements. Instead of extending any of class Node or class Edge, arbitrary custom data can be bound to any instance of these classes by using data accessors.

The following topics have been discussed in this chapter:

- The yFiles graph structure implementation basically consists of classes Graph, Node, and Edge.
- The graph structure is directed, but also allows for undirected interpretation.
- It knows about self-loops and parallel edges.
- There is a clear hierarchy of responsibilities; especially, class Graph is the single authority for all structural changes.
- To iterate over graph elements, there are interfaces NodeCursor and EdgeCursor; these are typed variants of interface YCursor which defines a bidirectional cursor.
- To store graph elements, there are classes NodeList and EdgeList; these are typed variants of class YList which defines a doubly linked list.
- To bind arbitrary data to graph elements the yFiles library uses data accessors; data accessors can have read-only, write-only, or read/write behavior; interface DataProvider offers read-only behavior, while interface DataAcceptor offers write-only behavior.
- The most frequently used data accessors are interfaces NodeMap and EdgeMap; both have read/write behavior.
- Instead of extending graph element classes to get customized functionality, yFiles favors the use of data accessors to bind arbitrary data to the instances of graph elements.



---

# Chapter 4. Working With the Graph Structure

This chapter presents first steps to using the yFiles graph structure and outlines general programming aspects. It explains technical consequences of some of the concepts in more detail. Also, it introduces well-known graph analyzing algorithms indispensable to the serious graph worker.

## Creating Graphs and Graph Elements

Class `Graph` offers three constructors, one to create an empty graph, i.e., one where node and edge set are empty, the other two to create new instances which are either partial or complete copies of an already existing graph.

```
// Create a new, empty graph.  
Graph graph = new Graph();  
  
// Create a complete copy of 'graph'.  
Graph graphCopy = new Graph(graph);
```

It is also possible to ask an instance of type `Graph` to create a complete copy of itself or to create a new empty graph instance.

When populating an existing graph with graph elements, it is important to observe certain pre-conditions:

- Only class `Graph` provides methods to create nodes and edges.
- To create an edge in a graph  $G$  it is necessary that both its source node and its target node already exist in  $G$ .

```
// 'graph' is of type y.base.Graph.  
  
// Create 10 nodes.  
Node n[] = new Node[10];  
for (int i = 0; i < 10; i++)  
    n[i] = graph.createNode();  
  
// Create 5 edges. Each edge has "even" source node and "odd" target node.  
Edge e[] = new Edge[5];  
for (int i = 0, j = 0; i < 10; i += 2, j++)  
    e[j] = graph.createEdge(n[i], n[i + 1]);
```

For further code examples see also [GraphDemo.java](#) and [RandomGraphGenerator.java](#) from the tutorial demo applications.

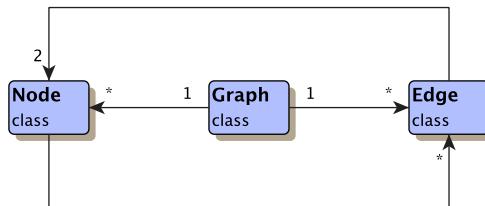
### Important

Note that the Java code snippets presented in the course of this document serve as examples to demonstrate how to use the functionality yFiles provides. Although they are all correct, they are not complete and will most likely not compile when copied directly.

# Graph Structure Functionality

The yFiles graph structure implementation as depicted in [Figure 4.1, “The basic graph structure” \[18\]](#) shows the main classes together with their associations and multiplicity. A graph can have an arbitrary number of nodes and edges, every node and edge, though, has exactly one graph. A node can have an arbitrary number of edges, but an edge always has two nodes (its source node and its target node).

**Figure 4.1. The basic graph structure**



## More About Class Graph

In addition to the functionality presented in the section called “[Creating Graphs and Graph Elements](#)” [17], which covers simple graph element creation, class `Graph` also provides methods that offer more control in specific situations.

For example, in addition to the most frequently used simple edge creation, it is also possible to specify the exact place where a newly created edge should be inserted into both the sequence of outgoing edges at its source node and the sequence of incoming edges at its target node. Furthermore, an already existing edge can be completely remodeled, i.e., both its end nodes as well as the insertion places at either end node can be changed. Reversing an edge, as another variant of changing an edge, is also supported.

[Example 4.1, “Exact edge insertion” \[18\]](#) shows the respective line of code for specifying the exact place of edge insertion.

### Example 4.1. Exact edge insertion

```

// 'graph' is of type y.base.Graph.
// 'source' and 'target' are of type y.base.Node.

// Create a new edge at the second position of all incoming edges at the target
// node.
graph.createEdge(source, source.firstOutEdge(),
                 target, target.firstInEdge(), graph.AFTER, graph.AFTER);
  
```

[Figure 4.2, “Exact edge insertion” \[18\]](#) shows the resulting graph. The newly created edge is inserted after the first incoming edge at the target node. Observe how the sequence of incoming edges (indicated by the numbers near the arrowheads) changes with the insertion.

**Figure 4.2. Exact edge insertion**



(a) Graph before...

(b) ... and after edge insertion.

The analogous operation to graph element creation is their removal. Class [Graph](#) offers methods to remove all graph elements at once, or to remove only a single node or a single edge. When removing either an instance of type [Node](#) or an instance of type [Edge](#) their reference to the containing graph is cleared.

```
// 'graph' is of type y.base.Graph.

// Remove single graph elements from the graph.
// Note that all edges adjacent to the given node are removed prior to the node
// itself.
graph.removeNode(graph.firstNode());
graph.removeEdge(graph.lastEdge());

// Remove all graph elements from the graph at once.
graph.clear();
```

### Note

When removing a single node from a graph, all its connecting edges are also removed. Technically, they are even removed before the node is, since an edge requires both its source node and target node to reside in the same graph.

An alternative to removing graph elements is to hide them. Hiding is a technique to only *temporarily* remove graph elements, i.e., taking them away from the graph structure to perform some task on the remaining graph, and thereafter putting them back into again, unhideing them. Note that both removing and hiding a graph element only takes them away from the graph, the objects themselves still exist.

```
// 'graph' is of type y.base.Graph.

// Hide single graph elements from the graph.
// Note that all edges adjacent to 'exampleNode' are hidden prior to the node
// itself.
Node exampleNode = graph.firstNode();
graph.hide(exampleNode);

// Unhide single graph elements from the graph.
// Note that *only* 'exampleNode' is unhidden, while its formerly adjacent edges
// are not.
graph.unhide(exampleNode);
```

In order to be able to unhide previously hidden graph elements, it is necessary to retain them, for example, in a data structure. Hiding of graph elements can also be performed utilizing specialized classes which automatically keep track of all hidden elements. These classes are described in [the section called “Advanced Topics” \[23\]](#)

### Important

Hiding a graph element only differs from removing a graph element in that there is no graph event fired that signals the structural change. (Please see [the section called “Events and Listeners” \[25\]](#)for a description of graph events.) As a consequence, any listeners to such graph events won't be notified when either node or edge is hidden, which can easily result in data structures getting out of sync with the actual structure of the graph.

The most notable listener that strongly depends on proper notification of any structural changes to a graph is class [HierarchyManager](#). It is responsible to manage essential aspects of hierarchically organized graphs, and uses graph events to synchronize its own data structures. (Please see [Chapter 7, Graph Hierarchies \[23\]](#)for the description of hierarchically organized graph.)

Therefore, when working with graph hierarchies graph elements must not be hidden, but instead should be removed.

In addition to the number of nodes and the number of edges an instance of type [Graph](#) also knows whether a single graph element belongs to it or if there is an edge connecting two nodes. The latter feature is especially useful if there is no reference of an edge at hand, for instance.

If a graph element has been hidden it does not belong to any graph until it is unhidden again.

```
// 'graph' is of type y.base.Graph

// Get the number of nodes in the graph.
// (Both methods are equivalent.)
int nodeCount = graph.nodeCount();
int N = graph.N();

// Get the number of edges in the graph.
// (Both methods are equivalent.)
int edgeCount = graph.edgeCount();
int E = graph.E();

// Check if the graph is empty.
boolean isEmpty = graph.isEmpty();

// Check if the first node belongs to the graph.
boolean containsNode = graph.contains(graph.firstNode());

// Check if there is an edge between first and last node of the graph.
boolean containsEdge = graph.containsEdge(graph.firstNode(), graph.lastNode());
```

Class Graph offers access to its graph elements in several different ways. It returns either cursors to iterate over the respective sets of graph elements, or creates arrays containing references to all nodes or all edges from the graph. As an additional convenience, there are methods to get the first, respectively last element of any of the two sets. While the returned cursors by definition have read-only behavior on the underlying container, the returned arrays actually are copies of the respective sets of elements from the graph at a certain point in time. In effect, this means that, for example, a returned node array can be modified in any way, i.e., nodes might be removed from the array or the sequence of nodes might be changed, without affecting the node set from the graph.

To change the sequence of any of the graph element sets class Graph has methods to move an element to the first or last position, respectively.

```
// 'graph' is of type y.base.Graph.

// Get the first and last node of the node set from the graph.
Node firstNode = graph.getFirstNode();
Node lastNode = graph.getLastNode();

// Exchange first and last node of the node set.
graph.moveToLast(firstNode);
graph.moveToFirst(lastNode);

// Get the first and last edge of the edge set from the graph.
Edge firstEdge = graph.getFirstEdge();
Edge lastEdge = graph.getLastEdge();

// Exchange first and last edge of the edge set.
graph.moveToLast(firstEdge);
graph.moveToFirst(lastEdge);

// Get a cursor of all nodes from the graph.
NodeCursor nc = graph.nodes();

// Get a cursor of all edges from the graph.
EdgeCursor ec = graph.edges();

// Get an array of all nodes from the graph.
Node nodes[] = graph.getNodeArray();

// Get an array of all edges from the graph.
Edge edges[] = graph.getEdgeArray();
```

## Graph Elements

Every instance of a graph element, i.e., every instance of type `Node` and type `Edge` holds a reference to the instance of type `Graph` it belongs to. This reference is cleared when the graph element has been removed or hidden from its graph.

```
// 'node' is of type y.base.Node.

// Get the graph the node belongs to.
Graph graph = node.getGraph();
```

Furthermore, every graph element knows the position it has in the respective set of elements within its graph. More precisely, when an instance of type `Graph` is asked for an array of its

nodes, for example, this array reflects the node iteration sequence exactly, i.e., a node at position 6 in the array will return the value 6 when asked for its position.

### Important

The position of a removed graph element is undefined. The same holds for a hidden graph element.

```
// 'graph' is of type y.base.Graph.
// 'node' is of type y.base.Node.

// Get the node's position.
int index = node.index();

// Get an array of all nodes from the graph.
Node nodes[] = graph.getNodeArray();
// Check the positions of all nodes.
for (int i = 0; i < graph.N(); i++)
{
    if (nodes[i].index() != i)
        throw new RuntimeException("Mismatch at position " + i + ".");
    else
        System.out.println("All is well at position " + i + ".");
}
```

### Class Node

Class **Node** is responsible for everything that is related to its connecting edges. More specifically, an instance of type **Node** knows the overall number of connecting edges (the so-called degree of the node) as well as the division into the number of incoming edges (in-degree) and the number outgoing edges (out-degree).

```
// 'node' is of type y.base.Node.

// Get the number of edges at a node.
int degree = node.degree();

// Get the number of incoming edges at a node.
int inDegree = node.inDegree();

// Get the number of outgoing edges at a node.
int outDegree = node.outDegree();
```

A node also provides methods to easily test whether there is a connecting edge to another node.

```
// 'graph' is of type y.base.Graph.

Node firstNode = graph.firstNode();
Node lastNode = graph.lastNode();

// Check whether there is an edge between first and last node of the graph.
// First check if there is a connecting edge outgoing to 'lastNode'.
Edge e = firstNode.getEdgeTo(lastNode);
// If not, then check if there is a connecting edge incoming from 'lastNode'.
if (e == null)
    e = firstNode.getEdgeFrom(lastNode);
```

Furthermore, it can give cursors to iterate over all connecting edges, iterate over all incoming edges, and iterate over all outgoing edges. When iterating over all edges incoming edges appear first, outgoing edges last.

```
// 'node' is of type y.base.Node.

// Get a cursor to iterate over all edges at a node.
EdgeCursor edges = node.edges();

// Get a cursor to iterate over all incoming edges at a node.
EdgeCursor inEdges = node.inEdges();

// Get a cursor to iterate over all outgoing edges at a node.
EdgeCursor outEdges = node.outEdges();
```

### Important

Since a self-loop is both an incoming and an outgoing edge at its node, it appears twice when iterating over all edges of the node. Also, a self-loop counts twice when asking

for the degree of a node.

As a convenience, class `Node` furthermore provides cursors to iterate over all its neighbors, to iterate over all its predecessors, or to iterate over all its successors. Predecessors and successors mean the nodes at the opposite of an incoming edge, and an outgoing edge, respectively.

```
// 'node' is of type y.base.Node.

// Get a cursor to iterate over all neighbor nodes.
NodeCursor neighbors = node.neighbors();

// Get a cursor to iterate over the source nodes of all incoming edges.
// These nodes are called predecessors.
NodeCursor predecessors = node.predecessors();

// Get a cursor to iterate over the target nodes of all outgoing edges.
// These nodes are called successors.
NodeCursor successors = node.successors();
```

## Class Edge

The most important information an instance of type `Edge` provides is its source node and its target node. Closely related is the method to get the opposite when holding one of the edge's end nodes.

```
// 'edge' is of type y.base.Edge.

// Get the two end nodes of an edge.
Node source = edge.source();
Node target = edge.target();

// Getting the opposite when holding one of either source or target node.
Node opposite = edge.opposite(source);
```

### Note

Although a hidden edge holds no reference to a graph, it can still provide access to its source and target node, independent of their status.

As a convenience, class `Edge` additionally offers information whether it is a self-loop.

```
// 'edge' is of type y.base.Edge.

// Ask the edge whether it is a self-loop.
boolean isSelfloop = edge.isSelfloop();
```

### Note

Ignoring the directedness of the yFiles graph structure implementation and instead interpreting it undirected would be done using method `neighbors` of class `Node`, or a combination of methods `edges` of class `Node` and `opposite` of class `Edge`.

## Complexity

Table 4.1, “Time complexities in package `y.base`” [22] shows time *complexities* for several of the most frequent tasks when working with the yFiles graph structure implementation.

Creation of graph elements takes constant time as does the removal of an edge. Removing a node, though, implies the removal of all its connecting edges first, so this task takes linear time, i.e., time proportional to the number of connecting edges. Also, all iteration is done in linear time. Checking whether a given node or edge belong to a graph is done in constant time, while testing if there is an edge connecting two nodes takes linear time.

**Table 4.1. Time complexities in package `y.base`**

Task	Involved Class(es)	Complexity
Creating a node	Graph	O(1)
Creating an edge	Graph	O(1)
Removing/hiding a node	Graph	O(Node.degree())

Task	Involved Class(es)	Complexity
Removing/hiding an edge	Graph	O(1)
Clearing the graph	Graph	O(Graph.N() + Graph.E())
Iterating over the node set	Graph, NodeCursor	O(Graph.N())
Iterating over the edge set	Graph, EdgeCursor	O(Graph.E())
Iterating over all edges connecting to a node	Node, EdgeCursor	O(Node.degree())
Iterating over all incoming edges	Node, EdgeCursor	O(Node.inDegree())
Iterating over all outgoing edges	Node, EdgeCursor	O(Node.outDegree())
Checking if a node belongs to a graph	Graph	O(1)
Checking if an edge belongs to a graph	Graph	O(1)
Checking if there is an edge connecting two nodes	Graph	O(min(source.outDegree(), target.inDegree()))

## Advanced Topics

### Hiding Graph Elements

Support for hiding graph elements is further completed by classes [GraphHider](#) and [GraphPartitionManager](#) from package `y.util`. In addition to the methods from class `Graph`, `GraphHider` offers various methods to hide arbitrary collections of graph elements or even all elements from the graph at once. Moreover, it automatically keeps track of all elements hidden as a result of such method calls.

To unhide graph elements, class `GraphHider` provides a small set of methods that collectively employ all-at-once behavior when unhiding, i.e., all elements previously hidden are unhidden at once.

Class `GraphPartitionManager` extends the idea from class `GraphHider` in a way so that it is possible to mark parts of the graph as so-called "partitions" and hide/unhide these. [Example 4.18, "Making a graph connected"](#) [40] is a rather extensive code snippet where an instance of `GraphPartitionManager` is used to hide/unhide connected components.

### Important

When working with graph hierarchies graph elements must not be hidden, but instead should be removed.

### Copying a Graph

Class [GraphCopier](#) provides convenient support for copying an object of type `y.base.Graph`. It enables the complete range from copying the entire `Graph` to copying only sets of graph elements. Optionally, any data associated to either nodes or edges by means of data accessors, i.e., data providers or node maps and edge maps, respectively, can also be copied.

`GraphCopier` relies on implementations of static inner interface [GraphCopier.CopyFactory](#) to delegate the actual work of copying nodes and edges from one graph to another to. A simple default implementation for a copy factory that is capable of handling a `Graph` object is provided by class [GraphCopyFactory](#). Initially, i.e., when no other copy factory has been set, class `Graph` returns a similar implementation when its `getGraphCopyFactory` method is invoked.

### Example 4.2. Creating a GraphCopier

```
// 'graph' is of type y.base.Graph.
// Create a new GraphCopier that uses the copy factory which is registered with
// the graph.
GraphCopier gc = new GraphCopier(graph.getGraphCopyFactory());
```

[API Excerpt 4.1, "Copy methods in class GraphCopier"](#) [24] lists the methods from class `GraphCopier` that can be used to copy either an entire `Graph` object or the subgraph that is induced by a

given set of nodes from a given source graph.

### API Excerpt 4.1. Copy methods in class GraphCopier

```
// Copies an entire graph.  
Graph copy(Graph sourceGraph)  
NodeList copy(Graph sourceGraph, Graph targetGraph)  
  
// Copies the subgraph induced by the given nodes.  
Graph copy(Graph sourceGraph, NodeCursor sourceNodes)  
NodeList copy(Graph sourceGraph, NodeCursor sourceNodes, Graph targetGraph)
```

By default, copying a Graph object or a set of nodes does not include any data associated with either the nodes or edges. The methods shown in [API Excerpt 4.2, “Data accessor-related setter methods” \[24\]](#) can be used to control whether the contents of data accessors, i.e., both data providers and node maps and edge maps, respectively, that are registered with the source graph are copied over to the target graph, too.

Data accessors are presented in the section called “Binding Data to Graph Elements” [32]

### API Excerpt 4.2. Data accessor-related setter methods

```
// Controls whether to copy data provider contents.  
void setDataProviderContentCopying(boolean dataProviderContentCopying)  
  
// Determines whether to copy the contents of node maps and edge maps  
// registered with the source graph.  
void setNodeMapCopying(boolean nodeMapCopying)  
void setEdgeMapCopying(boolean edgeMapCopying)
```

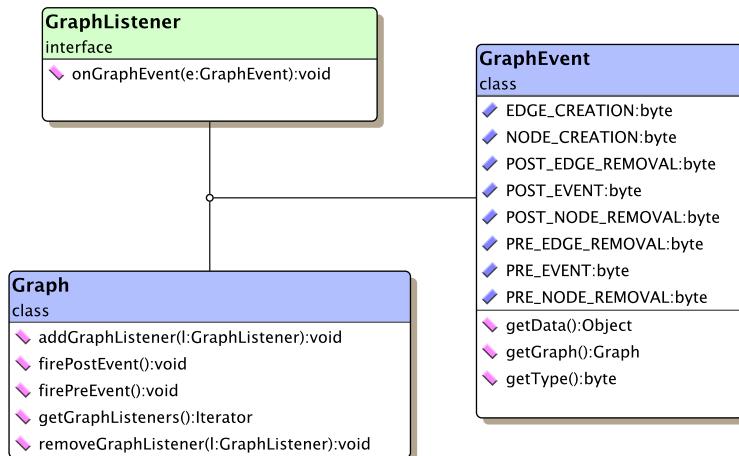
# Events and Listeners

## Structural Changes

Interface `GraphListener` notifies interested parties of structural graph changes. The information is conveyed using objects of type `GraphEvent`, and covers element insertion and removal as well as element modifications like, e.g., changing an edge's end nodes.

To receive notifications, `GraphListener` implementations have to be properly registered with a `Graph` object. [Figure 4.3, “Context for using GraphListener” \[25\]](#) gives an overview on interface `GraphListener`'s role.

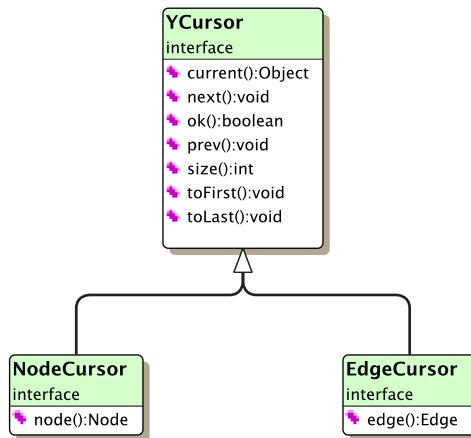
**Figure 4.3. Context for using GraphListener**



# Iteration Mechanisms

The yFiles cursor hierarchy with interface **YCursor** and its typed sub-interfaces **NodeCursor** and **EdgeCursor** provides bidirectional iteration over a sequence of objects in a uniform fashion. Figure 4.4, “The yFiles cursor hierarchy” [26] depicts the inheritance relationship along with all relevant methods.

**Figure 4.4. The yFiles cursor hierarchy**



A cursor offers access to the object it currently points to, it tells whether the current position is valid or “out of range,” and finally a cursor gives the size of the sequence it iterates over. Upon creation, a cursor by default points to the first object of the underlying sequence. Moving the cursor position is possible both absolutely, i.e., to the first or last object of the sequence and relatively, i.e., to the next or the previous object. Note that the typing of the sub-interfaces takes place at the access methods, i.e., these return references of type *Node* and *Edge*, respectively.

```

// 'graph' is of type y.base.Graph.

// Get a cursor to iterate over all edges from the edge set of the graph.
EdgeCursor ec = graph.edges();

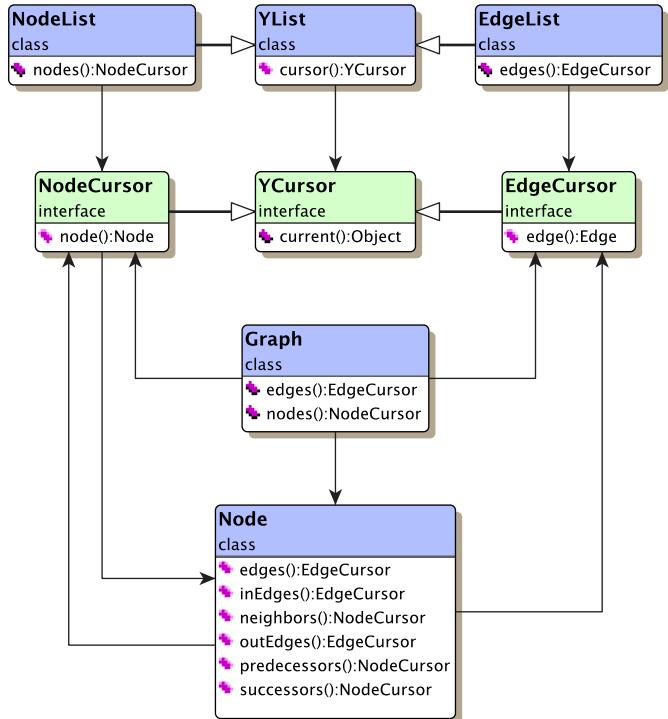
// Ask for the size of the underlying sequence of edges.
int size = ec.size();

// Move the cursor to the last position.
ec.toLast();
  
```

A cursor cannot change the sequence it iterates over, particularly, it cannot remove elements from the underlying container. So effectively, a cursor constitutes a read-only view on the sequence it iterates over. To actually remove an element from the underlying container, the container itself must be accessible. See the section called “Iteration and Element Removal” [28] for more details.

Figure 4.5, “Classes that provide cursors” [26] shows the most prominent yFiles classes that provide cursors to iterate over object, node, or edge sequences, respectively. These classes are **YList**, **NodeList**, **EdgeList**, **GraphInterface**, **Graph**, and **Node**.

**Figure 4.5. Classes that provide cursors**



Example 4.3, “Forward iteration with various cursor types” [27]and Example 4.4, “Backward iteration” [27]show the usage of cursors in conjunction with loops to iterate over a sequence of either typed or untyped objects. Note the additional cursor movement to the last position of the sequence with backward iteration.

### Example 4.3. Forward iteration with various cursor types

```

// 'graph' is of type y.base.Graph.
// 'list' is of type y.base.YList.

// Forward iterate over all objects from the list.
for (YCursor c = list.cursor(); c.ok(); c.next()) {
    Object obj = c.current();
}

// Forward iterate over all nodes of the node set from the graph.
for (NodeCursor nc = graph.nodes(); nc.ok(); nc.next()) {
    Node n = nc.node();
}

// Forward iterate over all edges from the edge set of the graph.
for (EdgeCursor ec = graph.edges(); ec.ok(); ec.next()) {
    Edge e = ec.edge();
}
  
```

### Example 4.4. Backward iteration

```

// Get a cursor to iterate over the list.
YCursor c = list.cursor();

// Move the cursor to the last position of the sequence.
// Backward iterate over all objects.
for (c.toLast(); c.ok(); c.prev()) {
    Object obj = c.current();
}
  
```

For further code examples see also [ListDemo.java](#) from the tutorial demo applications.

### Iteration and Element Removal

Interface `YCursor` does not provide a method to remove an element from an underlying container. Instead, any elements must be removed from the container itself.

#### Example 4.5. Removing elements from an underlying container

```
// 'graph' is of type y.base.Graph.

// Remove unwanted nodes from the node set of the graph.
for (NodeCursor nc = graph.nodes(); nc.ok(); nc.next())
{
    if (isUnwanted(nc.node()))
        // The graph is asked to remove the specified node.
        graph.removeNode(nc.node());
}
```

However, when an element that is pointed to by a cursor gets removed from the underlying container, the cursor does not change state, i.e., it still points to the same, now removed, element. Moving the cursor thereafter either forward or backward has the same effect as if the element was in the container, i.e., the cursor subsequently points to the next, respectively previous element from the foregoing sequence.

Once the cursor has been moved away from the removed element's position, though, there is no way to return to that position again. In conclusion, a simple call sequence as in [Example 4.6](#), “Call sequence to move a cursor” [28] does not necessarily return the cursor to the originating position in the sequence.

#### Example 4.6. Call sequence to move a cursor

```
// 'graph' is of type y.base.Graph.

// Get a cursor to iterate over all nodes of the node set from the graph.
NodeCursor nc = graph.nodes();

// The cursor points to the first position of the sequence, i.e., to the first
// node of the node set.
Node firstN = nc.node();
nc.next();                                // Cursor now points to the second node.
Node nextN = nc.node();
// The simple call sequence: move forward and immediately backward.
nc.next();                                // Cursor now points to the third node.
nc.prev();                                 // Cursor now points to the second node.
// Test for consistency.
if (nextN != nc.node())
    throw new RuntimeException("Inconsistent cursor.");

// Move the cursor back to the first node.
nc.toFirst();

Node firstN = nc.node();                  // Cursor now points to the second node.
Node nextN = nc.node();
// Now, remove the second node from the node set.
graph.removeNode(nc.node());
// The simple call sequence: move forward and immediately backward.
nc.next();                                // Cursor now points to the third node.
nc.prev();                                 // Cursor now points to the first node.
// Test for consistency.
if (nextN != nc.node())
    throw new RuntimeException("Inconsistent cursor.");
// This time an exception will be thrown that the cursor has become
// inconsistent.
```

### Alternative Iteration Techniques

Additionally to iteration using cursors, there is another possibility to specifically iterate over the edges adjacent to a node. [Example 4.7](#), “Non-cursor iteration over a node's edges” [28] demonstrates this technique.

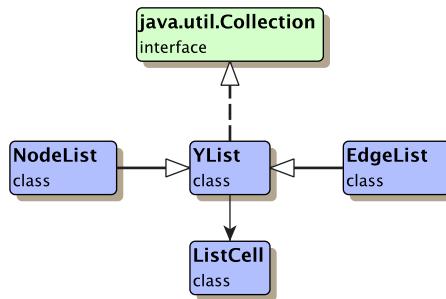
#### Example 4.7. Non-cursor iteration over a node's edges

```
// 'node' is of type y.base.Node.  
  
// Retrieve the initial incoming edge from the node.  
Edge e = node.firstInEdge();  
while (e != null)  
    // Successively retrieve the next incoming edge.  
    e = e.nextInEdge();  
  
// Retrieve the initial outgoing edge from the node.  
e = node.firstOutEdge();  
while (e != null)  
    // Successively retrieve the next outgoing edge.  
    e = e.nextOutEdge();
```

# Containers

Classes `NodeList` and `EdgeList`, the typed descendants of class `YList`, are the most frequently encountered yFiles data structures to store graph elements. Figure 4.6, “yFiles graph element containers” [30] shows their relationship.

**Figure 4.6. yFiles graph element containers**



Common to all lists is their support for the methods from interface `java.util.Collection` via the implementation in class `YList`. Beyond this implementation, class `YList` provides advanced list access and manipulation services which strongly rely on the possibilities offered by class `ListCell`.

## High-Level Features

In addition to the methods from interface `java.util.Collection` there are several further methods offered. For instance, it is possible to add elements explicitly to the front or the back of a list. Also, removing elements from the list is possible from both the front and the back. Note that the method names for removing elements from either end of the list resemble those of abstract data type stack.

Other methods offer list sorting, reversing a list, or the splicing of two lists into one.

### Example 4.8. Adding elements to and removing elements from a list

```

// 'list' is of type y.base.YList.

// Add new elements to either end of the list.
list.addFirst("I am the first node!");
list.addLast("I am the last node!");

// Remove elements from either end of the list.
list.pop();
list.popLast();
  
```

To iterate over the elements of the list it provides a `YCursor` (respectively `NodeCursor` and `EdgeCursor` with classes `NodeList` and `EdgeList`). Even though `java.util.Iterator` is also available as an alternative, the yFiles cursors are the commonly used and most qualified means for iteration. See also the discussion in the section called “Iteration Mechanisms” [26].

Example 4.9, “Efficiently removing elements from a list” [30] shows how to efficiently remove an element from a list. The cursor directly points to the element, there is no need to search the list.

### Example 4.9. Efficiently removing elements from a list

```

// 'list' is of type y.base.YList.

// Remove unwanted objects from the list.
for (YCursor c = list.cursor(); c.ok(); c.next())
{
    if (isUnwanted(c.current()))
  
```

```
    list.removeAt(c);
}
```

## Low-Level Features

Class `ListCell` is the building block of the doubly linked list structure. It knows both its predecessor and its successor, and provides read/write behavior for the actual data it is holding.

### Example 4.10. Using class ListCell

```
// 'list' is of type y.base.YList.  
  
// Get the first cell of the list.  
ListCell firstCell = list.firstCell();  
  
// Get the actual data the cell is holding.  
Object obj = lastCell.getInfo();  
  
// Change the actual data.  
lastCell.setInfo("Updated Data.");
```

Using an instance of type `ListCell`, for example, it is possible to have fast access to the preceding, respectively succeeding cell from the containing list, or to insert new cells relatively to already contained ones. By way of the successors (predecessors) of a `ListCell` instance, direct iteration over the elements of a list is possible.

### Example 4.11. Directly iterating over the cells of a YList

```
// 'list' is of type y.base.YList.  
  
// Directly (backward) iterate over the list.  
for (ListCell lc = list.lastCell(); lc != null; lc = list.predCell(lc))  
{  
    if (conditionIsSatisfied(lc.getInfo()))  
    {  
        // Insert a new cell with given data relatively to the held reference cell.  
        // The newly allocated cell will be returned.  
        ListCell newCell = list.insertCellBefore("Prepended Data.", lc);  
    }  
}
```

For further code examples see also [ListDemo.java](#) from the tutorial demo applications.

# Binding Data to Graph Elements

The concept of data accessors comprises two aspects which are commonly used in different scenarios. To bind supplemental data to graph elements that should be read-only, an implementation of interface `DataProvider` suffices. We will call these implementations "data providers" subsequently. A data accessor with full read/write behavior, though, additionally implements interface `DataAcceptor`. The yFiles library knows these implementations as "maps," and has two dedicated interfaces already defined, `NodeMap` and `EdgeMap`. Both extend interface `DataMap` which is a combination of interfaces `DataProvider` and `DataAcceptor`.

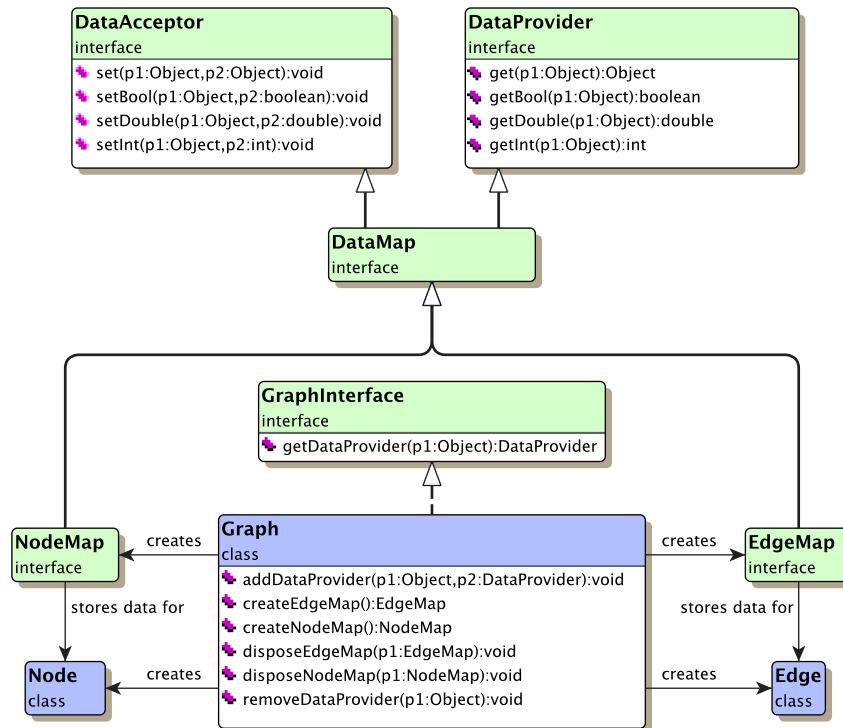
Applying the two semantics it is, e.g., possible to restrict certain callees to "immutable" data, while others are allowed to make changes.

## Note

Observe that both interface `NodeMap` and `EdgeMap` show identical signatures on their respective methods, using `java.lang.Object` instead of either `y.base.Node` or `y.base.Edge` as the parameter type for their key. Actual implementations should nevertheless ensure that the keys provided have correct type.

**Figure 4.7, "The concept of data accessors"** [32] gives a brief overview of the classes involved in the basic concepts of maps and data providers.

**Figure 4.7. The concept of data accessors**



## Maps and Data Providers

Common to all data accessor implementations which are offered by the yFiles library is that they cover all elements of a set, i.e., a node map provides values for all nodes from a graph (however, these may all be default values when there hasn't been anything stored yet). [Example 4.12, "Storing and retrieving data associated with a node"](#) [32] shows how node maps are used to store and retrieve arbitrary data.

### Example 4.12. Storing and retrieving data associated with a node

```
// 'graph' is of type y.base.Graph.
// 'labelNodeMap' is of type y.base.NodeMap.
// 'counterNodeMap' is of type y.base.NodeMap.

// Bind a label to the first node of the node set.
// The bound data actually is of type java.lang.String.
labelNodeMap.set(graph.firstNode(), "I am the first node!");

// Increase the value stored in 'counterNodeMap' for the last node.
// The bound data is an int.
counterNodeMap.setInt(graph.lastNode(),
                      counterNodeMap.getInt(graph.lastNode()) + 1);

// Print out the label of the first node.
System.out.print("The name of the first node is: ");
System.out.println(labelNodeMap.get(graph.firstNode()));
```

**Table 4.2, “Comparing map implementations”** [33] lists the differences of some data accessor implementations.

**Table 4.2. Comparing map implementations**

	Domain	Memory	Perform-ance	Note
Default maps	Multi-purpose	o	+	Need cleanup.
Index-based maps	Single-purpose	++	++	Require the underlying container to remain unaltered.
HashMap backed maps	Multi-purpose	+	o	Work well for sparse data.

For a code example comparing the different map implementations see also [NodeMapTest.java](#) from the tutorial demo applications.

## Default Map Implementations

Although writing a customized implementation of interfaces NodeMap or EdgeMap is easy, the most frequently used way to get these is conveniently provided by class Graph. [Example 4.13, “Creating default node maps”](#) [33] shows one of the two methods that both return default implementations of these interfaces which can be used for most purpose and data types.

The maps returned by these methods hold exactly one value for a given key, i.e., no matter how many calls to any of the setter methods are issued for a given key, only the last value set will be held. Also, the type of the key given with a setter method is restricted to the respective type of graph elements, i.e., restricted to Node or Edge. The type of the value though, is not restricted to be same over the range of all nodes, for example. In fact, it would be perfectly legal to set a value of type double with one node, and values of type boolean with every other. This however, is strongly discouraged, since it definitely leads to problems when the values will be retrieved.

Default map implementations can be created at any time, even when the graph is empty. From the moment of creation on, they will cover all graph elements from the respective set, as well as all respective elements created by the graph thereafter. However, these maps cannot cover elements that are hidden at the time of creation.

Note that these default implementations have to be properly disposed of after usage to release the allocated resources. To this end, class Graph has appropriate methods for either kind of default implementations.

### Example 4.13. Creating default node maps

```
// 'graph' is of type y.base.Graph.

// Obtain a new y.base.NodeMap default implementation from the graph.
NodeMap nodeMap = graph.createNodeMap();

// Set values for some of the nodes.
nodeMap.setDouble(graph.firstNode(), 3.14);
nodeMap.setDouble(graph.lastNode(), 42.0);
```

```
// Print the values stored in the node map.
for (NodeCursor nc = graph.nodes(); nc.ok(); nc.next())
{
    System.out.print("Node " + nc.node() + ": ");
    System.out.println(nodeMap.getDouble(nc.node()));
}

// Finally release the resources previously allocated by the createNodeMap()
// method.
graph.disposeNodeMap(nodeMap);
```

### Creating Customized Data Accessors

In addition to using the default map implementations provided by class Graph, there are further ways to create either maps or data providers. For example, data providers can be implemented so that the actual data is only implicitly defined, i.e., it is calculated on the fly when the value is asked for. This way, it is possible to "store" large amounts of data without having any memory be allocated. [Example 4.14, "Using class DataProviderAdapter to create customized data providers"](#) [34] demonstrates how to use class `DataProviderAdapter` to elegantly define a custom `DataProvider` implementation.

#### Example 4.14. Using class `DataProviderAdapter` to create customized data providers

```
// Define a DataProvider implementation that for each node in the graph returns
// an int that is the square of the node's index.
final DataProvider implicitDataProvider = new DataProviderAdapter() {

    // This implementation provides only ints.
    public int getInt(Object dataHolder)
    {
        if (!dataHolder instanceof Node)
            throw new UnsupportedOperationException("Key has wrong type.");
        final Node node = (Node) dataHolder;

        return (node.index() * node.index());
    }

    // Define a DataProvider implementation that for each edge in the graph returns
    // a distance value that is the difference of the values returned by
    // implicitDataProvider for the source and target node.
    final DataProvider edgeLengthProvider = new DataProviderAdapter() {

        // This implementation provides only ints.
        public int getInt(Object dataHolder)
        {
            if (!dataHolder instanceof Edge)
                throw new UnsupportedOperationException("Key has wrong type.");
            final Edge edge = (Edge) dataHolder;

            return (implicitDataProvider.getInt(edge.target()) -
                    implicitDataProvider.getInt(edge.source()));
        }
    }

    // Display the values "stored" (i.e., calculated on the fly) for each edge.
    for (EdgeCursor ec = graph.edges(); ec.ok(); ec.next())
    {
        System.out.print("Edge " + ec.edge() + ": ");
        System.out.println(edgeLengthProvider.getInt(ec.edge()));
    }
}
```

Class `DataProviders` from package `y.util` offers a set of static methods to conveniently create several specialized data provider implementations for either nodes or edges. For instance, there are methods to create data providers from constant values, from given arrays, or from existing data providers.

Class `Maps` from package `y.util` provides a set of static methods to conveniently create several specialized map implementations:

- The `createNodeMap` and `createEdgeMap` methods that take `java.util.Map` implementations and return either a `NodeMap` or an `EdgeMap` view of the given instances. Basically, this allows for any valid `java.util.Map` implementation, e.g., `java.util.HashMap` or

`java.util.TreeMap`, to be used in conjunction with the yFiles graph implementation.

- The various `createIndexNodeMap` and `createIndexEdgeMap` methods return map implementations that are fast and at the same time use little memory.

### HashMap-Backed Map Implementations

HashMap implementations are true multi-purpose data holders, there is no restriction to the type of the keys nor the type of the values. Particularly, the keys are not restricted to graph elements. Compared to the default implementations of interface NodeMap and EdgeMap provided by class Graph they are generally a bit slower. However, their memory usage is proportional to the amount of the data that is actually associated with the entities.

### Tip

Map implementations backed by an instance of type `java.util.HashMap` are especially suited for sparsely distributed data, i.e., only few entities of a domain have non-null data associated with them.

### Index-Based Map Implementations

Index-based map implementations are very fast and use only little memory, their drawback, however, is that once instantiated their values are restricted to the type given at creation time. More important though, all index-based containers require the respective set of graph elements to remain unaltered, i.e., there are no operations allowed that change the sequence of the graph elements in any way.

### Example 4.15. Using yFiles convenience classes to create edge maps

```
// 'graph' is of type y.base.Graph.  
  
// Create an edge map that holds a boolean value for each edge from  
// the edge set of the graph.  
EdgeMap map = Maps.createIndexEdgeMap(new boolean[graph.E()]);  
  
// Store some data into the edge map.  
// (For each edge the map will contain the boolean value indicating whether  
// the edge points from a node with a lower index to a node with  
// a higher index.)  
for (NodeCursor nc = graph.nodes(); nc.ok(); nc.next())  
{  
    for (EdgeCursor ec = nc.node().edges(); ec.ok(); ec.next())  
    {  
        Edge e = ec.edge();  
        map.setBool(e, e.source().index() < e.target().index());  
    }  
}
```

## Notes

Successfully using a data accessor requires a kind of "agreement" on the type the data accessor holds. More precisely, storing the value for a key and retrieving the value thereafter has to be done using setter and getter methods of matching type. The result when using setter/getter methods with non-matching types highly depends on the specific data accessor implementation. [Example 4.16, "Successfully using data accessors"](#) [35] demonstrates the proper usage of a data accessor.

This rule furthermore implies that a self-made data accessor has to provide the proper getter (and/or setter) methods when it is used as a parameter to an algorithm.

### Example 4.16. Successfully using data accessors

```
// 'graph' is of type y.base.Graph.  
  
// Get a default NodeMap implementation from the graph.  
NodeMap nm = graph.createNodeMap();  
  
// Store values for some chosen nodes.  
nm.setBool(graph.firstNode(), true);  
nm.setBool(graph.lastNode(), true);
```

## Notes

---

```
// Retrieve the stored values.  
// WRONG. WRONG. WRONG.  
// Boolean values cannot be retrieved as ints.  
int firstValue = nm.getInt(graph.firstNode());  
int lastValue = nm.getInt(graph.lastNode());  
  
// Retrieve the stored values.  
// RIGHT.  
boolean first = nm.getBool(graph.firstNode());  
boolean last = nm.getBool(graph.lastNode());
```

# Analyzing Graphs

One of the key aspects of the yFiles library is its emphasis on sophisticated algorithms for analyzing graph structures. Package `y.algo` provides a variety of services ranging from simple graph attribute queries to advanced graph characteristic computations. [Table 4.3, “Services for analyzing graphs” \[37\]](#) gives an overview of the classes and their services.

In general, most of the services from package `y.algo` are offered by means of static methods, i.e., there is no need to create an instance of the respective specialized classes. Class `Dfs`, however, is an exception to this rule. It provides a framework to build customized depth-first search behavior, and has to be extended appropriately.

**Table 4.3. Services for analyzing graphs**

Class Name	Description
<code>Bfs</code>	This class provides services that center around breadth-first search (BFS).
<code>Bipartitions</code>	Responsible for graph bipartition problems.
<code>Centrality</code>	Provides several so-called "centrality" measures to denote importance of graph elements within their graph.
<code>Cycles</code>	Responsible for finding cycles within a graph that have certain properties.
<code>Dfs</code>	Framework class for depth-first search (DFS) based algorithms.
<code>GraphChecker</code>	Provides methods that check structural properties of a given graph.
<code>GraphConnectivity</code>	Provides algorithms for determining certain connectivity components within a graph.
<code>Groups</code>	Provides support for automatic graph clustering.
<code>NetworkFlows</code>	Provides sophisticated algorithms for solving classical network flow problems like <code>MinCostFlow</code> or <code>MaxFlow</code> .
<code>Paths</code>	Responsible for finding paths within a graph that have certain properties.
<code>RankAssignments</code>	Provides algorithms for solving the rank assignment problem.
<code>ShortestPaths</code>	Provides diverse algorithms and helper methods for solving the Shortest Path problem on weighted graphs, i.e., graphs with a cost function on the edges.
<code>SpanningTrees</code>	Provides (minimum) spanning tree algorithms for graphs.
<code>Transitivity</code>	Provides algorithms to compute reachability information for directed, acyclic graphs: transitive closure, transitive reduction.
<code>Trees</code>	Provides diverse algorithms and services for tree-structured graphs or subgraphs.

## Quickly Checking for Graph Characteristics

To conveniently check for the most common elementary graph characteristics class `GraphChecker` offers a set of static methods. These cover for example whether a graph is cyclic/acyclic or contains multiple components. For the more advanced, in-depth questions there are specialized classes, like, e.g., class `Cycles`, or `GraphConnectivity`. (Class `GraphChecker` provides only "meta"-services, it asks in turn these specialized classes.)

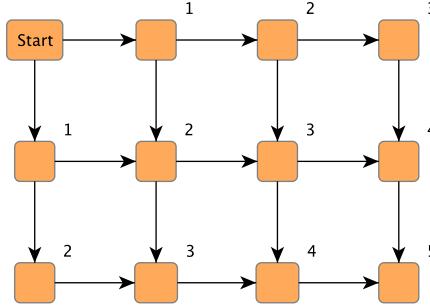
## Advanced Graph Characteristics

### Breadth-First Search

Beginning at a starting node (which is thought of as having distance 0 to itself) breadth-first search visits all of its direct neighbors (these having distance 1) before it in turn uses these already visited nodes as new starting nodes to continue with their direct neighbors. That way, it visits all reachable nodes at distance  $k$  to the original starting node before those at distance  $k+1$ .

Look at the example graph in [Figure 4.8, “Breadth-first search distances”](#) [38] The numbers at the upper-right corners of the nodes denote the distance the respective node has to the starting node.

**Figure 4.8. Breadth-first search distances**



Class `Bfs` returns all nodes with equal distance in a so-called "layer." The methods it provides all allow for multiple start nodes which are called "core" nodes. By definition, all core nodes are returned in the first layer.

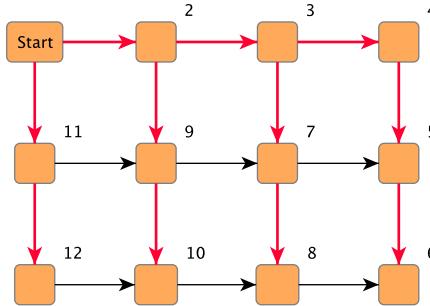
As an option, the BFS run can either be directed or undirected. By default, class `Bfs` ignores edge directions, i.e., it interprets the edges undirected.

## Depth-First Search

Beginning at a starting node depth-first search (DFS) recursively visits the first yet undiscovered direct neighbor of every reached node before continuing in the same manner with the remaining neighbors at this node.

Look at the example graph in [Figure 4.9, “Depth-first search graph traversal”](#) [38] The numbers at the upper-right corners of the nodes denote the visiting sequence with directed DFS. Additionally, the emphasized edges show the path of the DFS run. Observe that the "Start" node is visited first and accordingly is assigned the first index (not shown).

**Figure 4.9. Depth-first search graph traversal**



Class `Dfs` is used as a framework to build customized depth-first search services. It offers a collection of callback methods for all important events during a DFS run. These callbacks are empty by default, and have to be extended appropriately by an inheriting class. [Example 4.17, “Using class Dfs”](#) [39] demonstrates how to use the callback methods provided by class `Dfs`.

Events during a DFS run are pre- and post-visit events with nodes and pre- and post-traversal events with edges. Another callback is provided to handle unconnected graphs when all nodes that are reachable from the starting node have been visited and there are still undiscovered nodes

in the graph. Note that the invocation of this callback can optionally be disabled.

Class Dfs either ignores or respects edge directions, i.e., it interprets the graph structure either undirected or directed.

### Example 4.17. Using class Dfs

```
// Extend class Dfs.
class MyDFS extends Dfs
{
    // Print the so-called "DFS number."
    // This number denotes when a node is visited.
    void previsit(Node node, int dfsNumber) {
        System.out.println("Node: " + node +
                           " has DFS number: " + dfsNumber);
    }

    // Print the so-called "completion number."
    // This number denotes when a node has been completed.
    void postVisit(Node node, int dfsNumber, int compNumber) {
        System.out.println("Node: " + node +
                           " has completion numbers: " + compNumber);
    }

    // The graph is unconnected, i.e., it has multiple components.
    void lookFurther(Node v) {
        System.out.println("New starting node: " + v);
    }
}

// Instantiate the customized class.
public runDFS(Graph graph)
{
    MyDFS myDFS = new MyDFS();
    // Set directed interpretation of the graph structure.
    myDFS.setDirectedMode(true);

    // Run the depth-first search algorithm on the given graph.
    myDFS.start(graph);
}
```

### Tip

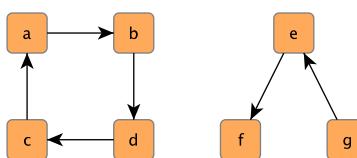
To simply compute the DFS completion numbers for the node set of a graph there are convenience methods provided in class [NodeOrders](#).

### Graph Connectivity

To answer the question which parts of a graph are reachable from a given node by following the edges irrespective of their direction, class [GraphConnectivity](#) is of good help. It offers services to check if a graph is connected or even biconnected, it has methods to find all connected/biconnected components of a graph, and methods to insert additional edges into the graph thus making it connected/biconnected.

[Figure 4.10, “A graph with two components”](#) [39] for instance, shows a graph with two connected components. In fact, the left one is even biconnected, i.e., removing any one of {a, b, c, d} leaves the remaining nodes still connected.

**Figure 4.10. A graph with two components**



[Example 4.18, “Making a graph connected”](#) [40] shows code to make a graph a connected compilation of biconnected components. Using the graph from [Figure 4.10, “A graph with two components”](#), we can make it connected by adding the edge (e, f).

ponents” [39] as input would lead to [Figure 4.11, “A connected graph”](#) [40] where the dotted edges represent the additionally inserted.

### Example 4.18. Making a graph connected

```
// 'graph' is of type y.base.Graph.

// Check if the graph is connected.
if (!GraphConnectivity.isConnected(graph))
{
    // Create a node map that will be used with class GraphPartitionManager.
    NodeMap components = graph.createNodeMap();

    // Get all connected components of the graph.
    NodeList nl[] = GraphConnectivity.connectedComponents(graph);

    // Create an array that serves as a mapping from an int to an Object.
    String i2o[] = new String[nl.length];

    for (int i = 0; i < nl.length; i++)
    {
        i2o[i] = Integer.toString(i);
        // Fill the node map with the respective marker object of each component.
        for (NodeCursor nc = nl[i].nodes(); nc.ok(); nc.next())
            components.set(nc.node(), i2o[i]);
    }

    // Create a PartitionManager to manage the components of the graph.
    GraphPartitionManager gpm = new GraphPartitionManager(graph, components);

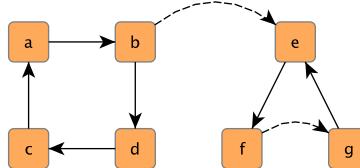
    // Make the PartitionManager hide all connected components of the graph.
    gpm.hideAll();
    for (int i = 0; i < nl.length; i++)
    {
        // Make the PartitionManager unhide all nodes and edges from a specific
        // component.
        gpm.unhidePartition(i2o[i]);

        // Check if this connected component from the graph is biconnected.
        if (!GraphConnectivity.isBiconnected(graph));
        {
            // Make the connected component biconnected, i.e., add some edges.
            EdgeList el = GraphConnectivity.makeBiconnected(graph);
        }
        // Make the PartitionManager hide the biconnected component.
        gpm.hidePartition(i2o[i]);
    }
    // Make the PartitionManager unhide all nodes and edges from the graph.
    gpm.unhideAll();

    // Make the entire graph connected.
    EdgeList el = GraphConnectivity.makeConnected(graph);

    // Release the resources previously allocated by the createNodeMap() method.
    graph.disposeNodeMap(components);
}
```

**Figure 4.11. A connected graph**

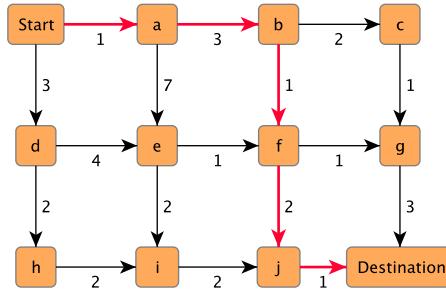


### Shortest Paths

Solving the problem of shortest paths is a commonly encountered question with graphs. Consider [Figure 4.12, “Finding the shortest path”](#) [41] for instance. The numbers at the edges denote “costs” that are associated with the traversal of an edge. To get from “Start” to “Destination” with minimum costs results in the path of emphasized edges. These edges define the shortest

path between the two nodes with an accumulated cost of 8 [units].

**Figure 4.12. Finding the shortest path**



Example 4.19, “Running single-source single-sink Shortest Path” [41] shows the code to prepare and run a single-source single-sink Shortest Path algorithm on the graph depicted in Figure 4.12, “Finding the shortest path” [41].

### Example 4.19. Running single-source single-sink Shortest Path

```

// 'graph' is of type y.base.Graph.

// Create an array the size of the edge set with costs for each edge.
double cost[] = {1, 3, 2, 3, 7, 1, 1, 4, 1, 1, 2, 2, 2, 3, 2, 2, 1};
// Create a data provider from the array.
// This data provider will be used as input to the Shortest Path algorithm.
DataProvider dp = (DataProvider)Maps.createIndexEdgeMap(cost);

// Create a node map to be filled by the algorithm.
NodeMap pred = graph.createNodeMap();

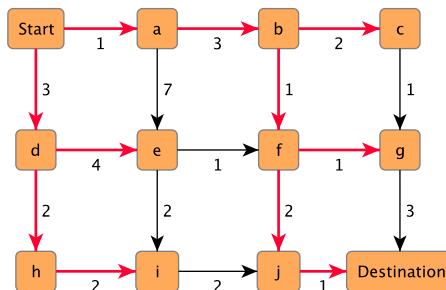
// Suppose the first node from the graph is the node named "Start."
Node startNode = graph.firstNode();
// Suppose the last node from the graph is the node named "Destination."
Node destinationNode = graph.lastNode();

// Run the single-source single-sink algorithm on the graph.
double result = ShortestPaths.singleSourceSingleSink(graph, startNode,
                                                      destinationNode, true, dp, pred);

// Release the resources previously allocated by the createNodeMap() method.
graph.disposeNodeMap(pred);
    
```

Figure 4.13, “Finding the shortest paths from a starting node” [41] shows the shortest paths from “Start” to all other nodes from the graph. Beginning at the destination node of each of these paths, “Start” can be reached unambiguously by following the emphasized edges.

**Figure 4.13. Finding the shortest paths from a starting node**



The class [ShortestPaths](#) offers a multitude of methods using several well-known algorithms to solve shortest path problems in most of its variants. Paths can be computed between two nodes (single-source single-sink), originating from one node (single-source), or between all pairs of nodes (all pairs). Included are:

- Methods for acyclic graphs with arbitrary costs on the edges, i.e., negative values are allowed.
- Dijkstra algorithm for arbitrary graphs, i.e., even cyclic graphs, with non-negative costs on the edges.
- Bellman-Ford algorithm for arbitrary graphs and arbitrary cost values on the edges. (Graphs that contain cycles with negative costs, though, yield no reasonable result, but instead give a boolean return value `false`.)

In addition to customized costs on the edges, it is possible to compute shortest paths with uniform costs, too. There are several convenience methods defined to this end.

Note that many of the methods from [ShortestPaths](#) use maps and/or data providers to store either its result or to receive its parameters. Furthermore, all appropriate methods offer the option of interpreting the graph structure either directed or undirected.

Class [ShortestPaths](#) also has a set of convenience methods for result conversion, i.e., constructing a node or an edge path from either a given data provider or an array. [Example 4.20](#), “[Running single-source Shortest Path](#)” [42] demonstrates how to construct one specific path from the returned array after a single-source Shortest Path algorithm is run on the already prepared graph from [Figure 4.12](#), “[Finding the shortest path](#)” [41].

### Example 4.20. Running single-source Shortest Path

```
// Run the single-source Dijkstra algorithm on the graph.
ShortestPaths.dijkstra(graph, startNode, true, cost, new double[graph.N()],
pred);

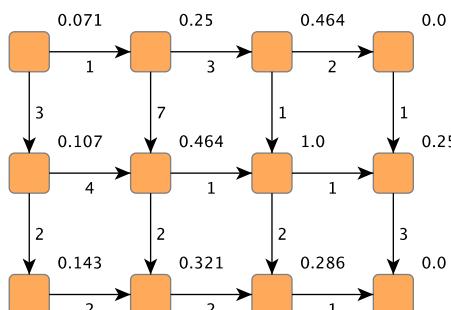
// Suppose 'iNode' of type y.base.Node holds a reference to the node named "i."
// Construct the proper path from "Start" to "i" as a list of nodes.
NodeList nl = ShortestPaths.constructNodePath(startNode, iNode, pred);
```

## Centrality Measures

For the elements of a graph the notion of importance of a single node or edge within the graph structure can be expressed using so-called “centrality” measures. A centrality measure is a function that yields a double value for each node or each edge. This value directly denotes the graph element's importance, i.e., the higher the value, the more important the element.

Class [Centrality](#) provides a variety of such centrality measures for both node set and edge set. Most of these measures also take edge costs into account. [Figure 4.14](#), “[Node betweenness centrality](#)” [42] depicts a graph with node centrality values at the upper-right corner of the nodes. The numbers at the edges denote “costs” that are associated with an edge.

**Figure 4.14. Node betweenness centrality**



The methods from class Centrality optionally either ignore or respect edge direction when computing centrality indices. Also, the class offers convenience methods to normalize all values computed for either node set or edge set to lie in the interval [0..1].

[Example 4.21, “Computing centrality indices for nodes”](#) [43] shows the code to prepare and run the node betweenness centrality algorithm on the graph depicted in [Figure 4.14, “Node betweenness centrality”](#) [42] ignoring edge directions. Subsequent to the computation, the resulting double values are also normalized.

### Example 4.21. Computing centrality indices for nodes

```
// 'graph' is of type y.base.Graph.

// Create an array with the costs for each edge.
double cost[] = {1, 3, 2, 3, 7, 1, 1, 4, 1, 1, 2, 2, 2, 3, 2, 2, 2, 1};
// Create the data provider to be used as parameter from this array.
DataProvider dp = (DataProvider)Maps.createIndexEdgeMap(cost);

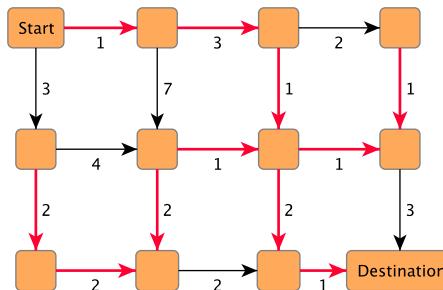
// Create the node map to hold the centrality indices for all nodes.
NodeMap myCentrality = graph.createNodeMap();
// Compute node betweenness centrality.
Centrality.nodeBetweenness(graph, myCentrality, false, dp);
// Normalize the double values so that they lie in the interval [0..1].
Centrality.normalize(graph, myCentrality);
```

## Spanning Trees

A spanning tree of a connected graph is the smallest set of edges such that all nodes of the graph are connected. Finding these and also finding the *minimum* spanning tree of a graph is the duty of class `SpanningTrees`.

Consider [Figure 4.15, “Finding the minimum spanning tree”](#) [43] for instance. The numbers at the edges denote "costs" that are associated with an edge. The smallest set of edges such that all nodes from the graph are connected and the accumulated costs of the set is at a minimum results in the emphasized edges. These edges define the minimum spanning tree of the graph with an overall cost of 17 [units].

**Figure 4.15. Finding the minimum spanning tree**



Class `SpanningTrees` provides different algorithms to find a minimum spanning tree of a given graph. Furthermore, it also offers convenience methods to get the overall costs of a spanning tree, and to compute a spanning tree with uniform costs at the edges.

[Example 4.22, “Running the Spanning Tree algorithm”](#) [43] shows the code to prepare and run a Spanning Tree algorithm on the graph depicted in [Figure 4.15, “Finding the minimum spanning tree”](#) [43].

### Example 4.22. Running the Spanning Tree algorithm

```
// 'graph' is of type y.base.Graph.

// Create an array with the costs for each edge.
```

```

double cost[] = {1, 3, 2, 3, 7, 1, 1, 4, 1, 1, 2, 2, 2, 3, 2, 2, 1};
// Create the data provider to be used as parameter from this array.
DataProvider dp = (DataProvider)Maps.createIndexEdgeMap(cost);

// Compute the minimum spanning tree of the graph.
EdgeList el = SpanningTrees.kruskal(graph, dp);
// Get the overall costs of the minimum spanning tree.
double treeCosts = SpanningTrees.cost(el, dp);

```

### Transitivity

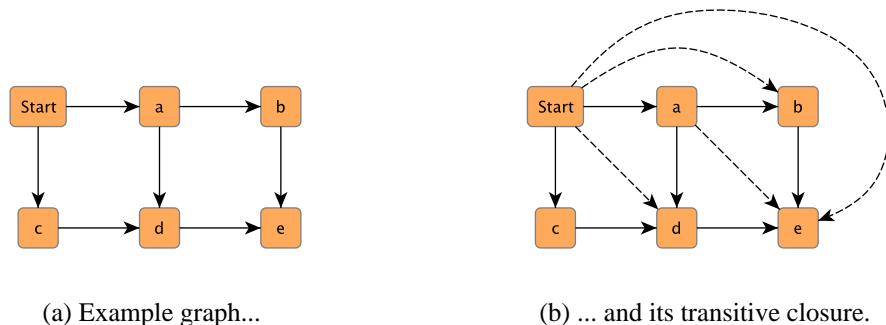
Class [Transitivity](#) offers sophisticated computational services relating to reachability questions in directed acyclic graphs. The two operations provided are transitive closure and transitive reduction.

Transitive closure means the augmentation of a graph's edge set such that an existing path between any two nodes, which are not direct neighbors to each other, leads to an additional edge connecting these nodes. In effect, this means that any two nodes that are reachable by a path of arbitrary length before the augmentation, are instantly reachable thereafter, i.e., they are direct neighbors.

In [Figure 4.16, “Transitive closure of a graph”](#) [44] a simple graph together with its transitive closure is depicted. The additionally inserted edges are dotted. For example, the dotted edge between "Start" and "e" has been inserted to reflect even multiple possible paths from "Start" to "e": [a, b, e], [a, d, e], or [c, d, e] are all existing paths.

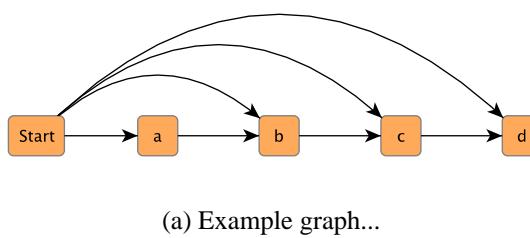
Note that class Transitivity computes only the transitive, but not the reflexive, transitive closure. However, the latter one can easily be achieved by adding self-loops to each node.

**Figure 4.16. Transitive closure of a graph**



Transitive reduction is the reverse operation to transitive closure. It means the removal of all directly connecting edges from a graph as long as there remains another path of at least two edges length between the two considered nodes. [Figure 4.17, “Transitive reduction of a graph”](#) [44] shows a simple graph and its transitive reduction.

**Figure 4.17. Transitive reduction of a graph**



(b) ... and its transitive reduction.

## Trees

Class `Trees` provides a variety of services centering around tree structures. It offers methods to modify existing trees and methods to check if a graph fulfills certain tree characteristics. Moreover, it has methods to "extract" tree structures contained in arbitrary graphs. Note that there is a division in the methods of class `Trees`. On the one hand there are methods to deal with general graphs, and on the other hand there are methods that expect the given parameter of type `Graph` to be a proper tree.

To properly use the methods from class `Trees`, it is necessary to understand the notion of a rooted tree and what directed tree means. Directed means that all edges uniformly point towards the leaf nodes. A rooted tree, as it is usually defined, meets the following conditions:

- It is a directed tree, i.e., every node has at most one incoming edge and all edges point towards the leaves of the tree.
- There is exactly one node that has no incoming edge, but only outgoing edges. This is the root node.

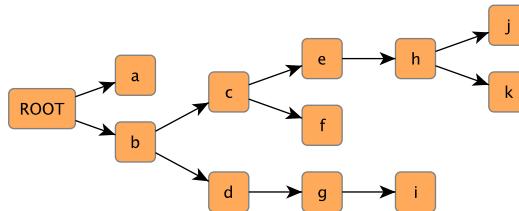
A reversed rooted tree means that all edges point towards the root node, instead of the leaf nodes. In particular, this means that the root node has only incoming, but no outgoing edges.

Querying the tree depicted in [Figure 4.18, “A simple tree” \[45\]](#) for its root node, e.g., would lead to node "ROOT", and the list of all leaf nodes would be {a, f, i, j, k}.

```
// 'graph' is of type y.base.Graph.
// Test if the graph is a tree.
if (Trees.isTree(graph))
{
    // Explicitly set the first node from the node set as the root node of the
    // tree.
    // Direct the tree; i.e., revert those edges that point towards the root
    // node.
    EdgeList el = Trees.directTree(graph, graph.firstNode());

    // Get the leaf nodes of the tree.
    NodeList nl = Trees.getLeafNodes(graph, true);
}
```

**Figure 4.18. A simple tree**



Finding the common ancestor of nodes from a tree is another query supported by class `Trees`. For instance, the common ancestor of nodes {f, j, k} from [Figure 4.18, “A simple tree” \[45\]](#) would be node "c."

### Example 4.23. Finding the common ancestor of a set of nodes

```
// 'graph' is of type y.base.Graph.
// Get the root node of the tree.
Node root = Trees.getRoot(graph);

// Get the leaf nodes of the tree.
```

```
NodeList nl = Trees.getLeafNodes(graph, true);
// Remove the first and last leaf node from the list.
nl.pop();
nl.popLast();

// Find the nearest common ancestor for all remaining leaves.
Node ancestor = Trees.getNearestCommonAncestor(graph, root, true, nl);
```

The following topics have been discussed in this chapter:

- How to create graphs and graph elements, and how to use the functionality provided by the basic graph structure classes to work with a graph.
- What different kind of cursors and lists are provided and how they are used.
- What maps and data providers are and how to get default map implementations from a graph.
- How to implement customized maps and data providers using appropriate convenience and utility classes.
- Sophisticated graph algorithms to analyze graphs, or to compute various graph characteristics.

---

# Chapter 5. Automatic Graph Layout

This chapter discusses the yFiles layout algorithms, and explains their options and customization possibilities. It shows how to create and invoke the algorithms, and how to achieve aesthetic and clear graph layouts. Also, the general layout architecture is introduced, and common concepts for the layout algorithms are described.

## Layout Algorithms

The yFiles package [y.layout](#) together with its sub-packages provides a multitude of different layout algorithms and styles. The algorithms divide into three main categories:

- [Graph layout algorithms](#) are responsible for assigning coordinates to all graph elements. More precisely, the nodes of a graph are placed according to some optimization criteria, and edge paths are also generated.
- [Edge routing algorithms](#) process only edges, i.e., they compute edge paths. The nodes of a graph are left unchanged, neither their position nor their size is altered in any way.
- [Label placement algorithms](#) leave both a graph's nodes and its edge paths unaltered, but compute suitable positions for labels. Their criteria for the arrangement are such that the labels do not overlap with each other or any of the graph elements.

Optimization criteria for graph layout algorithms are numerous. They vary from minimizing the length of edge paths to minimizing the number of bends, from placing highly connected components tightly together to placing nodes in a hierarchical top-to-bottom manner. Besides the rather unemotional nature of such optimization, a graph layout algorithm at the same time also aims at making its outcome an aesthetically pleasing experience for a user.

## Advanced Layout Features

The yFiles graph layout algorithms support a number of sophisticated features when calculating a graph's layout. Most notably, [hierarchically organized graphs](#) and arranging their grouped nodes is supported. [Bus-like edge paths](#) can be generated for edges connecting to a common node, and both starting point and end point of an edge path [can be constrained to a specific side of a node or even explicit coordinates](#). Additionally, some algorithms provide labeling as an integrated part of layout generation.

# Layout Architecture

To enable automatic graph layout, the basic graph structure is enriched with information for two-dimensional placement of its elements. Furthermore, the basis for layout algorithms to access this information is laid, too.

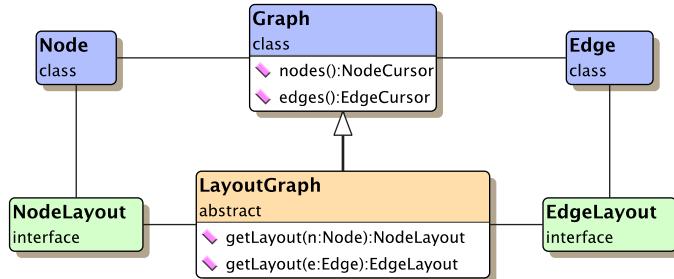
## Graph Structure Enhancements

The graph implementation in package `y.layout` is provided by abstract class `LayoutGraph`. It enhances the graph structure with support for positional and dimensional information for all graph elements.

In particular, this means width and height for a node and also coordinates for its upper left corner. For edges, more precisely edge paths, positional information means coordinates for both starting point and end point, as well as the control points in-between these two.

Both positional and dimensional information are indispensable for calculating the layout for a graph. Accordingly, these two are also called the "layout information" for a graph and its elements. [Figure 5.1, “Graph structure classes” \[48\]](#) depicts the hierarchy of the yFiles graph structure classes as present in package `y.layout`.

**Figure 5.1. Graph structure classes**

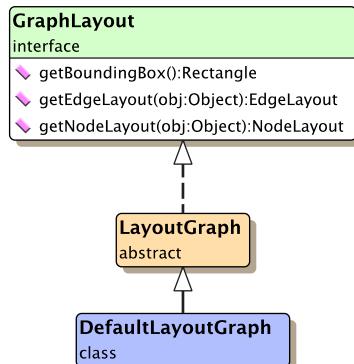


## Positional and Dimensional Information

Abstract class `LayoutGraph` implements interface `GraphLayout`, which defines the basis for layout information on graph elements. Class `DefaultLayoutGraph`, the default implementation for `LayoutGraph`, presents the main graph class of package `y.layout` that can be used for all graph layout tasks.

[Figure 5.2, “Class hierarchy for class DefaultLayoutGraph” \[48\]](#) shows the class hierarchy for class `DefaultLayoutGraph`.

**Figure 5.2. Class hierarchy for class DefaultLayoutGraph**



[API Excerpt 5.1, “LayoutGraph” \[49\]](#) shows the one method from abstract class LayoutGraph that returns graph-level layout information.

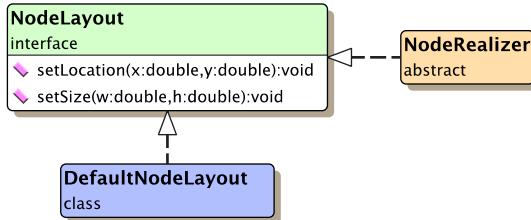
### API Excerpt 5.1. LayoutGraph

```
// Getter method for the graph's dimension.  
Rectangle getBoundingBox()
```

Interface [NodeLayout](#) and its default implementation, class [DefaultNodeLayout](#), are used to add the layout information to a node. This information consists of the coordinates for the upper left corner of the node, and its width and height.

[Figure 5.3, “NodeLayout” \[49\]](#) shows the class hierarchy for interface NodeLayout. Note that abstract class [NodeRealizer](#) from package y.view also implements this interface to hold a node's layout information.

**Figure 5.3. NodeLayout**



[API Excerpt 5.2, “Node-related methods from LayoutGraph” \[49\]](#) lists some of the convenience methods defined by abstract class LayoutGraph that can be used to control the layout information for nodes.

### API Excerpt 5.2. Node-related methods from LayoutGraph

```
// Getter methods for nodes.  
YPoint getCenter(Node v)  
YPoint getLocation(Node v)  
double getHeight(Node v)  
double getWidth(Node v)  
  
// Setter methods for nodes.  
void setCenter(Node v, YPoint position)  
void setLocation(Node v, YPoint position)  
void setSize(Node v, double w, double h)
```

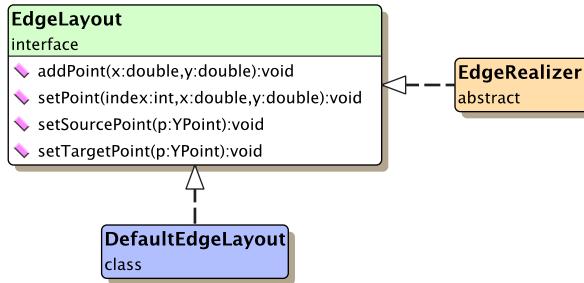
Interface [EdgeLayout](#) and its default implementation, class [DefaultEdgeLayout](#), are used to add the layout information to an edge. This information consists of the coordinates for both starting point and end point of the edge path, as well as the coordinates for the control points in-between these two.

### Important

The coordinates for the edge's end points are relative to the center coordinates of the edge's source node and target node, respectively. The coordinates for the control points, in contrast, are absolute.

[Figure 5.4, “EdgeLayout” \[49\]](#) shows the class hierarchy for interface EdgeLayout. Note that abstract class [EdgeRealizer](#) from package y.view also implements this interface to hold an edge's layout information.

**Figure 5.4. EdgeLayout**



**API Excerpt 5.3, “Edge-related methods from LayoutGraph” [50]** lists some of the convenience methods defined by abstract class LayoutGraph that can be used to control the layout information for edges. Note that the path list for an edge includes its source port, all control points, and also its target port. The point list, though, holds only the edge's control points.

### Note

Control points are also known as "bends." This other term is especially used with the visual representation of an edge, i.e., most notably in relation with package [y.view](#).

### API Excerpt 5.3. Edge-related methods from LayoutGraph

```

// Getter methods for edges.
YList getPathList(Edge edge)
YList getPointList(Edge edge)

YPoint getSourcePointAbs(Edge edge)
YPoint getTargetPointAbs(Edge edge)

// Setter methods for edges.
void setPath(Edge edge, YList path)
void setPoints(Edge edge, YList points)

void setSourcePointAbs(Edge edge, YPoint point)
void setTargetPointAbs(Edge edge, YPoint point)
  
```

## Advanced Topics

### Copying a LayoutGraph

The generic mechanism for copying a graph structure as described in the section called “[Copying a Graph](#)” [23] can also be used to copy implementations of abstract class LayoutGraph. Instead of a GraphCopyFactory instance, however, other copy factory implementations, which are capable of copying the information associated with the elements of a LayoutGraph, need to be used. Specifically, this means that node layout and edge layout data as well as node labels and edge labels need to be handled properly by such implementations.

Classes [LayoutGraphCopyFactory](#) and [LayoutGraphCopyFactory.HierarhicGraphCopyFactory](#) are predefined copy factories that handle both layout data as well as labels. The latter class is returned when calling the `getGraphCopyFactory` method on a `DefaultLayoutGraph`, the default LayoutGraph implementation. It is a hierarchy-aware graph copy factory that wraps a `LayoutGraphCopyFactory` instance and in addition to the services provided by its delegate also automatically copies the contents of all [GroupingKeys](#) data providers registered with the LayoutGraph.

**Example 5.1, “Creating a GraphCopier for a “flat” LayoutGraph” [50]** shows how to create a GraphCopier that uses the graph copy factory for “flat” graphs instead of the default hierarchy-aware one.

### Example 5.1. Creating a GraphCopier for a “flat” LayoutGraph

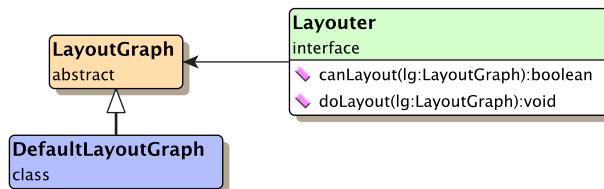
```
// 'graph' is of type y.layout.DefaultLayoutGraph.
// Create a new GraphCopier that uses a graph copy factory for "flat" graphs.
GraphCopier gc = new GraphCopier(new LayoutGraphCopyFactory());
```

## Layout Infrastructure

All graph layout algorithms in yFiles implement interface **Layouter**. This interface declares methods for testing if a given input graph of type **LayoutGraph** can be handled, and also for actually assigning a layout to such a graph.

The calculated new layout information that results from a Layouter invocation supersedes that of the given LayoutGraph object. [Figure 5.5, “Layouter dependencies” \[51\]](#) shows the dependencies for interface Layouter.

**Figure 5.5. Layouter dependencies**



[Example 5.2, “Invoking a layout algorithm” \[51\]](#) demonstrates how to invoke a graph layout algorithm on a graph of type **DefaultLayoutGraph**.

### Example 5.2. Invoking a layout algorithm

```
// 'graph' is of type y.layout.DefaultLayoutGraph.
// Run a hierarchical layout on the given graph.
Layouter hl = new HierachicLayouter();
hl.doLayout(graph);
```

# Concepts

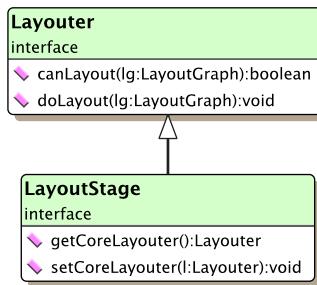
The task of a layout algorithm is a major undertaking that involves arbitrarily complex logic. However, there can be identified a number of well-defined (sub-)tasks that even completely different algorithms do in a similar manner. Factoring out such tasks so that they can be reused in varying contexts, greatly reduces the complexity of any layout algorithm.

The yFiles library allows to formulate complex layout processes by plugging together so-called "layout stages" that, among other things, can be used to handle such well-defined (sub-)tasks.

## The Layout Stages Concept

A layout stage serves as a kind of container that encapsulates arbitrary layout functionality, and provides a general means to string together multiple stages into a compound layout process. Interface `LayoutStage` defines the basis for a layout stage. It is a specialization of interface `Layouter`, and thus can be used as a stand-alone layout provider as well as a part of a larger layout process.

**Figure 5.6. LayoutStage**



The methods of interface `LayoutStage` are used to establish a relationship to another `Layouter` implementation, the so-called "core layouter." The core layouter's invocation is entirely bracketed by the layout stage's logic.

When used in the context of a larger layout process, the layout stage can easily be used to simplify the core layouter's task. It performs preprocessing steps on the input graph before the core layouter's invocation, and postprocessing steps thereafter.

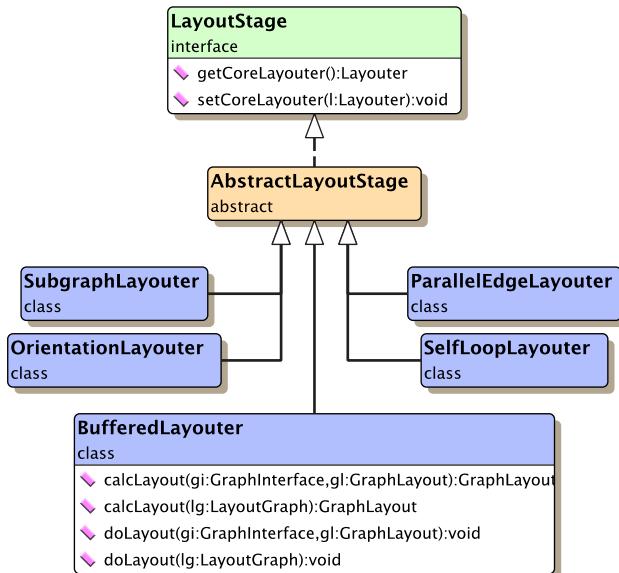
Table 5.1, "Predefined layout stages" [52] lists some of the predefined `LayoutStage` implementations, most of them being part of the default compound layout process as described below. Further layout stages are described in the section called "Layout Stages" [65]

**Table 5.1. Predefined layout stages**

Classname	Description
<code>BufferedLayouter</code>	Duplicates the graph to be processed, so that the layout cannot destroy the original data. See also the <a href="#">description of buffered layout calculation</a> below.
<code>SelfLoopLayouter</code>	Calculates orthogonal edge paths for a graph's self-loops (reflexive edges).
<code>ParallelEdgeLayouter</code>	Calculates edge paths for all edges with identical source node and target node. (Self-loops are not processed.)
<code>OrientationLayouter</code>	Changes the orientation of a computed layout.
<code>SubgraphLayouter</code>	Reduces the original graph to the subgraph that is induced by selected nodes.

Figure 5.7, "Layout stages complex" [53] shows the class hierarchy for the layout stages.

**Figure 5.7. Layout stages complex**



## Default Compound Layout Process

The most prominent example for a complex layout process can be observed with abstract class **CanonicMultiStageLayouter**. It is the superclass for the yFiles major layout algorithms, and provides a conveniently configurable layout pipeline for performing preprocessing steps on the input graph before the core layouter's invocation, and postprocessing steps thereafter. The central role of CanonicMultiStageLayouter can also be seen in [Figure 5.9, “The yFiles layout algorithms” \[72\]](#)

Except for class **BufferedLayouter**, all layout stages from [Table 5.1, “Predefined layout stages” \[52\]](#) are part of the compound layout process defined by **CanonicMultiStageLayouter**. [API Excerpt 5.4, “Configuration methods of class CanonicMultiStageLayouter” \[53\]](#) lists some of the methods that can be used for configuring the layout process.

### API Excerpt 5.4. Configuration methods of class CanonicMultiStageLayouter

```

// Methods to add and remove individual layout stages.
void appendStage(LayoutStage stage)
void prependStage(LayoutStage stage)
void removeStage(LayoutStage stage)

// Methods for enabling and disabling the predefined layout stages, and also
// for controlling their enabled state.
// (Excerpt.)
void enableOnlyCore()
boolean isComponentLayouterEnabled()
boolean isSelfLoopLayouterEnabled()
void setComponentLayouterEnabled(boolean enabled)
void setSelfLoopLayouterEnabled(boolean enabled)

// Getter and setter methods for predefined layout stages..
// (Excerpt.)
LayoutStage getComponentLayouter()
LayoutStage getSelfLoopLayouter()
void setComponentLayouter(LayoutStage layouter)
void setSelfLoopLayouter(LayoutStage layouter)

```

## Using Buffered Layout

With the yFiles layout algorithms it is possible to have a graph layout calculated using two different approaches, namely "unbuffered" layout or "buffered" layout. Unbuffered layout means to directly invoke a layout algorithm's `doLayout` method. Choosing this approach, the layout calculation is performed on the given graph, and is also immediately assigned. Buffered layout, in contrast, utilizes class `BufferedLayouter`, which creates a copy of the original graph that is then used for layout calculation.

Unbuffered layout has some severe drawbacks that should be observed:

- A layout algorithm might perform badly in terms of memory consumption and execution time, due to the implementation of the graph structure. Crucial graph methods might not be optimized for layout tasks. For instance, invoking a layout algorithm directly on an instance of `Graph2D` is almost never a good idea, since its nodes and edges are rather "heavy."
- A layout algorithm might perform badly in terms of memory consumption and execution time, due to the implementation of the graph structure. Crucial graph methods might not be optimized for layout tasks.
- Some layout algorithms need to temporarily add/remove nodes or edges to/from the given graph. Any registered graph listeners will be notified about such structural changes, which subsequently might result in unnecessary or even harmful action on a listener's behalf.
- Even though it is guaranteed that a layout algorithm will not change a graph's node set and edge set, it is not unusual that the ordering of nodes and/or edges is modified. Consequently, it is not safe to rely on the `index()` feature of nodes or edges.
- In rare cases it might happen that a layout algorithm will crash during a calculation (due to a bug, for example). It will then return immediately and generate an exception. The input graph will be left in an intermediate, often broken state and no recovery will be possible for it.
- Directly invoking a layout algorithm's `doLayout()` method will not return the calculated coordinates, but instead assign them right away to the given graph. Consequently, any other way of coordinate assignment, e.g., in an animated fashion using coordinate interpolation, is defeated.

With these drawbacks in mind, it is almost always a good idea to choose buffered layout instead. It facilitates many sophisticated features, like, e.g., layout morphing, and at the same time increases an application's robustness.

### Class `BufferedLayouter`

The main purpose of class `BufferedLayouter` is to create a copy of the input graph before calling its core layouter. The graph structure that is used for the copied graph is optimized for layout calculation.

The core layouter subsequently executes on the copy and calculates a new layout, which is then transferred to the original graph. There are several beneficial aspects of this functionality:

- The structure of the input graph is guaranteed to not change at all. Usually, layout providers (i.e., Layouter implementations) make no guarantees on leaving the sequence of nodes or edges unchanged, which may result in unexpected side effects. One such side effect is, for example, that a layouter may assign completely different layouts to a graph when being invoked twice on the same graph. The reason for such behavior is that a layout provider's output in general depends on the sequence of elements in the graph, but this sequence has changed with the first layout invocation.
- Instead of immediately writing back the calculated layout to the given input graph, class `BufferedLayouter` provides the possibility to return the result as a `GraphLayout` object, leaving the original graph's layout unmodified. The `GraphLayout` object then allows to defer coordinate assignment to a later point in time, for example, using class `LayoutTool`'s `applyGraphLayout` method. By means of class `LayoutMorpher`, it can also be used to nicely "morph" from one layout to another in an animated fashion. (See the section called "[Layout Morphing](#)" [232])
- Calculating a layout on a copy instead of the original graph proves to be more robust. Even if there should occur an unrecoverable error in the layout process, class `BufferedLayouter` guarantees that the structure of the input graph remains consistent.

Wrapping a layout algorithm with a BufferedLayouter layout stage is as easy as shown in [Example 5.3, “Using buffered layout” \[55\]](#)

### Example 5.3. Using buffered layout

```
// 'graph' is of type y.layout.LayoutGraph.  
// Run organic layout by implicitly wrapping its invocation using the services  
// of class BufferedLayouter.  
new BufferedLayouter(new OrganicLayouter()).doLayout(graph);
```

Alternatively, class BufferedLayouter allows to get the calculated graph layout as a separate object. This is demonstrated in [Example 5.4, “Buffered layout with deferred coordinate assignment” \[55\]](#)

### Example 5.4. Buffered layout with deferred coordinate assignment

```
// 'graph' is of type y.layout.LayoutGraph.  
// Run organic layout by implicitly wrapping its invocation using the services  
// of class BufferedLayouter.  
// The result of the layout is returned separately as a GraphLayout object,  
// i.e., the original graph's layout information is not changed.  
GraphLayout gl = new BufferedLayouter(new OrganicLayouter()).calcLayout(graph);
```

## Providing Supplemental Layout Data

The yFiles layout algorithms support advanced functionality that can take into account even individual information for single graph elements. However, individual setup like, e.g., attaching a preferred edge length to each edge, is beyond the means a graph structure provides. Instead, the data accessor concept as described in the section called “[Binding Data to Graph Elements](#)” [32] is utilized to provide the individual setup as supplemental information to a layout algorithm.

The supplemental data for the graph elements is bound to the graph using a so-called “data provider key.” During layout calculation, the algorithm then retrieves the data provider that contains the data from the graph by using the same key.

Depending on the data provider key, the algorithm expects the returned data provider to hold values of specific type for either nodes or edges of the graph. [Example 5.5, “Binding supplemental data” \[55\]](#) shows the use of an implicit data provider that returns an integral value for each edge indicating the edge’s preferred length to the organic layout algorithm.

### Example 5.5. Binding supplemental data

```
// 'graph' is of type y.layout.LayoutGraph.  
  
// Create an implicit data provider and register it with the graph using a  
// so-called data provider look-up key.  
// The key is then used by the layout algorithm to retrieve the supplemental  
// data.  
graph.addDataProvider(PREFERRED_EDGE_LENGTH_DATA, new DataProviderAdapter() {  
    public int getInt(Object o) {  
        return (int)(200 * getLength((Edge)o));  
    }  
});  
  
// Invoke organic layout on the graph.  
OrganicLayouter ol = new OrganicLayouter();  
ol.doLayout(graph);  
  
// Remove the data provider from the graph.  
graph.removeDataProvider(PREFERRED_EDGE_LENGTH_DATA);
```

# Advanced Layout Features

The layout algorithms that come with the yFiles library support a number of sophisticated and powerful features for layout generation, including:

- **Hierarchically organized graphs.** Layout of grouped graph structures.
- **Port constraints.** Restricting edge ports to a specific side of a node and/or a fixed location relative to the node's center.
- **Edge/Port grouping.** Multiple edge ports sharing a common location, and/or having a common edge segment.

Furthermore, many of the yFiles layout algorithms provide support for advanced incremental layout and the closely related concept of "layout from sketch." An introduction to both these concepts is presented in the section called "Incremental Layout" [76]

## Hierarchically Organized Graphs

The term "hierarchically organized graph" denotes a graph structure where, conceptually, nodes can be declared "children" of a common other node, their "parent." This can be exercised recursively, i.e., parents can be declared children of other parents, resulting in a hierarchy of nodes of possibly arbitrary depth.

The visual presentation of such a hierarchy is normally done by placing the children near each other and have their parent enclosing them. The parent is called a "group node," and the children are its content, they are "grouped nodes." Declaring some nodes to be children of another node is called "grouping."

### Note

Especially in the context of the yFiles Viewer distribution there is another term used for hierarchically organized graphs, they are also called "graph hierarchies." See the description of [graph hierarchies](#).

Layout support for hierarchically organized graphs primarily means proper handling of grouped nodes and their enclosing group node. There are three different group node policies available:

- Include group nodes and their contents for layout calculation. Group nodes as well as their contents are processed by the layout algorithm.
- Leave group node content fixed. The relative positions of the nodes inside a group node are left unaltered. However, the group node itself can change its position.
- Ignore group nodes entirely. Group node content is processed in a normal way by the layout algorithm, the group nodes are not processed, but only placed afterwards to again enclose all their children.

[Table 5.2, "Layout support for hierarchically organized graphs"](#) [56] lists the layout algorithms that provide support for group nodes and their content.

**Table 5.2. Layout support for hierarchically organized graphs**

Layout Style	Classname	Note
Hierarchical	<a href="#">HierarchicalGroupLayouter</a>	Support for group node policies is partly handled by prepended layout stages. See the <a href="#">description of hierarchical group layout</a> for more information.
Organic	<a href="#">OrganicLayouter</a> , <a href="#">SmartOrganicLayouter</a>	Organic layout provides direct support for all three group node policies. Class SmartOrganicLayouter uses the services of organic layout. See the descriptions of <a href="#">organic layout</a> and <a href="#">smart organic layout</a> for more information.
Orthogonal	<a href="#">OrthogonalGroupLayouter</a>	Support for group node policies is partly handled by prepended layout stages. See the <a href="#">description of orthogonal group layout</a> for more information.

Table 5.3, “Support for hierarchically organized graphs in incremental layout algorithms” [57] lists the incremental layout algorithms that provide support for group nodes and their content. Note however that group node policies as described above are not directly supported by these algorithms.

**Table 5.3. Support for hierarchically organized graphs in incremental layout algorithms**

Layout Style	Classname	Note
Hierarchical (incremental)	IncrementalHierarchicLayouter	IncrementalHierarchicLayouter provides direct support for hierarchically organized graphs for both incremental as well as non-incremental layout mode. See the <a href="#">description of IncrementalHierarchicLayouter's advanced features</a> for more information.

Table 5.4, “Routing support for hierarchically organized graphs” [57] lists the routing algorithms that provide support for hierarchically organized graphs.

**Table 5.4. Routing support for hierarchically organized graphs**

Routing Style	Classname	Note
Orthogonal	OrthogonalEdgeRouter	Support for routing with group nodes is partly handled by prepended layout stages. See the <a href="#">description of the orthogonal edge router</a> for more information.

## Setup for Layout

In the presence of package `y.view` (i.e., with the yFiles Viewer distribution), the setup for layout calculation of a `Graph2D` that has associated a `HierarchyManager` object is a matter of using the convenience methods of class `GroupLayoutConfigurator` as shown in Example 5.6, “Layout preparation with classes from package `y.view`” [57]

### Example 5.6. Layout preparation with classes from package `y.view`

```
// 'graph' is of type y.view.Graph2D.

GroupLayoutConfigurator glc = new GroupLayoutConfigurator(graph);
// Prepare all relevant information for a layout algorithm.
glc.prepareAll();

// Invoke buffered layout.
invokeBufferedLayout(graph, new HierarchicGroupLayouter(), false);

// Restore all group node specific sizes and insets after a layout algorithm
// has been run.
glc.restoreAll();
```

Example 5.7, “Layout preparation without classes from package `y.view`” [57] demonstrates how to set up a hierarchically organized graph without the convenience functionality from package `y.view`. Basically, the data providers that hold the necessary information about the graph’s hierarchical composition have to be filled manually, and be registered with the graph using the data provider look-up keys defined in interface `GroupingKeys`.

### Example 5.7. Layout preparation without classes from package `y.view`

```
// 'graph' is of type y.layout.LayoutGraph.

// Create the node maps that are to be used as data providers later on.
NodeMap groupKey = graph.createNodeMap();
NodeMap nodeID = graph.createNodeMap();
NodeMap parentNodeID = graph.createNodeMap();

// Register the node maps as data providers with the graph.
// Use the "well-known" look-up keys defined in interface GroupingKeys.
graph.addDataProvider(Grouping.GROUP_DPKY, groupKey);
graph.addDataProvider(Grouping.NODE_ID_DPKY, nodeID);
```

```

graph.addDataProvider(Grouping.PARENT_NODE_ID_DPKEY, parentNodeID);

// Now, set up the hierarchical organization of the graph, i.e., define some
// of the nodes to be group nodes and others to be their content.
setupHierarchicalOrganization(graph, groupKey, nodeID, parentNodeID);

// Invoke buffered layout.
invokeBufferedLayout(graph, new HierarchyGroupLayoutter(), false);

```

The information for the node IDs and the parent node IDs is of symbolic nature that is used in the process of layout calculation to identify the proper parent for a given child, but also to find all children that belong to the same parent. Hence, it is important for the symbolic IDs to properly match between these two data providers, so that the graph's hierarchical composition is correctly expressed.

**Example 5.8. “Encoding a graph's hierarchical composition in data providers”** [58] demonstrates possible content for the data providers. Here, the nodes themselves are used to denote symbolic IDs for both “ordinary” nodes and group nodes. Carefully observe the usage of the indirection scheme in this example for setting up the parent-child relation.

### Example 5.8. Encoding a graph's hierarchical composition in data providers

```

// Now, set up the hierarchical organization of the graph, i.e., define some
// of the nodes to be group nodes and others to be their content.
for (int i = 0; i < 10; i++) {
    // Nodes 1, 5, and 9 are defined to be group nodes.
    if (i % 4 == 1) {
        groupKey.setBool(n[i], true);
        // Set a symbolic ID for the group node that is used for look-up purposes.
        nodeID.set(n[i], n[i]);
        continue;
    }

    // Set a symbolic ID for the node that is used for look-up purposes.
    nodeID.set(n[i], n[i]);

    // Node 2 is defined child of node 1;
    // node 6 is defined child of node 5.
    if (i % 4 == 2) {
        // Establish the relation to the parent.
        parentNodeID.set(n[i], nodeID.get(n[i - 1]));
        continue;
    }

    // Node 3 is defined child of node 1;
    // node 7 is defined child of node 5.
    if (i % 4 == 3) {
        // Establish the relation to the parent.
        parentNodeID.set(n[i], nodeID.get(n[i - 2]));
    }

    // Nodes 0, 4, and 8 remain "ordinary" nodes...
}

```

A group node's size is determined by the bounding box that encloses its children and additional insets that are added to each of the box's side. To specify insets individually, a data provider can be used to hold an Insets object for each group node. This data provider is then registered with the graph using the look-up key **GROUP\_NODE\_INSETS\_DPKEY** defined in interface GroupingKeys.

**Example 5.9. “Defining a group node's insets”** [58] shows how to add individual Insets objects for group nodes to a node map, and how the node map is registered as a data provider with the graph.

### Example 5.9. Defining a group node's insets

```

// 'graph' is of type y.layout.LayoutGraph.

// Create the node map that is to be used as a data provider later on.
NodeMap groupNodeInsetss = graph.createNodeMap();

// Predefine some Insets objects.
Insets in[] = new Insets[3];
in[0] = new Insets(10, 20, 30, 40);

```

```

in[1] = new Insets(20, 20, 20, 20);
in[2] = new Insets(40, 30, 20, 10);

NodeList gnl = getListOfAllGroupNodes(graph);
for (NodeCursor nc = gnl.nodes(); nc.ok(); nc.next()) {
    Node n = nc.node();
    groupNodeInsets.set(n, in[getGroupType(n)]);
}

// Register the node map as a data provider with the graph.
// Use the "well-known" look-up keys defined in interface GroupingKeys.
graph.addDataProvider(GroupingKeys.GROUP_NODE_INSETS_DPKY, groupNodeInsets);

// Invoke buffered layout.
invokeBufferedLayout(graph, new HierarchicGroupLayouter(), false);

```

Furthermore, a group node's size can be controlled by registering a custom `GroupBoundsCalculator` implementation replacing the default `InsetsGroupBoundsCalculator` instance. To this end, the method from [API Excerpt 5.5, “Method to set another GroupBoundsCalculator implementation” \[59\]](#) is available with classes `HierarchicGroupLayouter` and `OrganicLayouter`.

### API Excerpt 5.5. Method to set another GroupBoundsCalculator implementation

```

// Getter and setter for custom GroupBoundsCalculator implementation.
GroupBoundsCalculator getGroupBoundsCalculator()

void setGroupBoundsCalculator(GroupBoundsCalculator groupBoundsCalculator)

```

Class `FixedGroupLayoutStage` adds support for the fixed group node policy to both hierarchical layout and orthogonal layout. [Example 5.10, “Setup for fixed group node content” \[59\]](#) shows how the fixed group node policy is realized as a layout stage that is prepended to the actual layout algorithm's invocation.

### Example 5.10. Setup for fixed group node content

```

void invokeBufferedLayout(LayoutGraph graph, Layouter layouter,
                          boolean orthogonal)
{
    // Create a specialized layout stage that fixes the contents of the group
    // nodes.
    FixedGroupLayoutStage fixedGroupLayoutStage = new FixedGroupLayoutStage();
    if (orthogonal)
        fixedGroupLayoutStage.setInterEdgeRoutingStyle(
            FixedGroupLayoutStage.ROUTING_STYLE_ORTHOGONAL);

    // Prepend the stage to the given layout algorithm.
    layouter.prependStage(fixedGroupLayoutStage);

    // Invoke buffered layout for the given layout algorithm.
    new BufferedLayouter(layouter).doLayout(graph);

    // Remove the prepended layout stage.
    layouter.removeStage(fixedGroupLayoutStage);
}

```

## Tutorial Demo Code

The tutorial demo application `GroupingLayoutWithoutAView.java` gives a detailed demonstration on how to set up a hierarchically organized graph without the functionality present in package `y.view`. Also, the modules that are used to set up the layout algorithms in an application context give deep insights in an algorithm's configuration:

- `HierarchicLayoutModule.java`
- `OrganicLayoutModule.java`
- `OrthogonalLayoutModule.java`

## Port Constraints

The two ends of an edge path are also called source port and target port, respectively. A port constraint serves to pinpoint an edge's end at its source node or target node. There are two kinds of port constraints available:

- Weak constraint. Determines the node's side at which an edge path's end should connect.
- Strong constraint. Determines the exact coordinates where the edge path's end should be located. The coordinates are interpreted relative to the node's center.

Both kinds of port constraints can easily be combined to express, for example, that an edge's source port should connect to the middle of the source node's upper border.

**Table 5.5. “Layout support for port constraints”** [60] lists the layout algorithms that provide support for port constraints.

**Table 5.5. Layout support for port constraints**

Layout Style	Classname	Note
Hierarchical	<a href="#">HierarchicLayoutOuter</a> , <a href="#">HierarchicGroupLayoutOuter</a> , <a href="#">IncrementalHierarchicLayoutOuter</a>	The algorithms of the family of hierarchical layout algorithms by default obey weak and strong port constraints as soon as they are set. See the descriptions of <a href="#">classic hierarchical layout</a> and <a href="#">incremental hierarchical layout</a> for more information.
Tree	<a href="#">GenericTreeLayoutOuter</a>	The predefined node placer implementations that can be used with the generic tree layout algorithm by default obey strong and weak port constraints as soon as they are set. See the <a href="#">description of generic tree layout</a> for more information.

**Table 5.6. “Routing support for port constraints”** [60] lists the routing algorithms that provide support for port constraints.

**Table 5.6. Routing support for port constraints**

Routing Style	Classname	Note
Orthogonal	<a href="#">OrthogonalEdgeRouter</a> , <a href="#">ChannelEdgeRouter</a>	Both OrthogonalEdgeRouter and ChannelEdgeRouter by default obey weak and strong port constraints as soon as they are set. See the descriptions of <a href="#">OrthogonalEdgeRouter</a> and <a href="#">ChannelEdgeRouter</a> for more information.

## Setup for Layout

**Example 5.11. “Adding source port constraints to some edges”** [60] demonstrates the creation of [PortConstraint](#) objects, and how they are stored in an edge map. The edge map is then registered as a data provider with the graph using the look-up key [SOURCE\\_PORT\\_CONSTRAINT\\_KEY](#). During layout calculation, an algorithm first retrieves the data provider using the look-up key, and afterwards retrieves the contained information.

The actual coordinates of an edge end point that has a strong port constraint associated are determined at the time a layout algorithm (or a routing algorithm) processes the edge. In other words, the *strong* characteristic of a strong port constraint is determined by its "normal" coordinates at the time of processing. Note that the specified coordinates of such edge end points are always interpreted relative to the respective node's center.

### Example 5.11. Adding source port constraints to some edges

```
// 'graph' is of type y.layout.LayoutGraph.

// Create edge map that is used as a data provider later on.
EdgeMap pcMap = graph.createEdgeMap();

// Set the coordinates for the edge's source port. (Actually, this could also
// be done anywhere prior to invoking the layout algorithm.)
graph.setSourcePointRel(edge3, new YPoint(-10, 20));
```

```

// Create PortConstraint objects:
// 1) Port constraint that allows an edge end to connect to any side of its
//    respective node.
PortConstraint pc1 = PortConstraint.create(PortConstraint.ANY_SIDE);
// 2) Port constraint that determines an edge end to connect *anywhere* to the
//    upper (NORTH) side of its respective node.
PortConstraint pc2 = PortConstraint.create(PortConstraint.NORTH);
// 3) Strong port constraint that determines an edge end to connect to the
//    lower (SOUTH) side of its respective node. The actual end point is at a
//    fixed coordinate.
PortConstraint pc3 = PortConstraint.create(PortConstraint.SOUTH, true);

// Establish a mapping from edges to port constraints.
pcMap.set(edge1, pc1);
pcMap.set(edge2, pc2);
pcMap.set(edge3, pc3);

// Register the edge map as a data provider with the graph.
// Use the "well-known" look-up key defined in interface PortConstraintKeys.
// Note that the above defined port constraints are set so that they apply to
// the source ends of their respective edges only.
graph.addDataProvider(PortConstraint.SOURCE_PORT_CONSTRAINT_KEY, pcMap);

// Invoke buffered layout.
invokeBufferedLayout(graph, new HierarchicLayouter());

```

### Tutorial Demo Code

The tutorial demo application [LayoutWithoutAView.java](#) demonstrates how to set up port constraints (both weak and strong) for edge end points without the functionality present in package `y.view`.

## Port Candidates

The concept of port candidates is an extension to that of classic port constraints [as described above](#). Port candidates can be used in conjunction with both nodes and edges, however, unlike port constraints, they present a node-centric approach. Primarily, port candidates provide a means to:

- restrict anchor locations at nodes, i.e., to define a set of locations where edges are allowed to connect to
- associate costs with a given anchor location, which effectively allows to establish an order of precedence among the set of anchor locations
- limit the number of connecting edges at a given anchor location

Class [PortCandidate](#) enables definition of port candidates that conceptually correspond to either weak port constraints, i.e., effectively describe side constraints, or strong port constraints, which encode specific anchor locations at a node. Note that port candidates that correspond to strong port constraints directly include the coordinates for the actual anchor locations.

[PortCandidate](#) additionally allows to associate costs with a given port candidate, which can be used to establish an order of precedence among multiple port candidates. When a given edge port is being assigned to any of the available port candidates at a node, those with low costs are favored compared to other port candidates with higher costs associated.

To define the set of side constraints and anchor locations at a node, multiple port candidates can easily be combined using the services of class [PortCandidateSet](#). When a [PortCandidate](#) object is added to an instance of [PortCandidateSet](#), the capacity of the port candidate can optionally be configured. The capacity of a given port candidate (sometimes also referred to as "cardinality") specifies the allowed number of connecting edges at that side or anchor location.

The [PortCandidateSet](#) objects for the node set of a graph can be registered by means of a data provider using the look-up key [NODE\\_DP\\_KEY](#).

### Matching Port Candidates

Matching port candidates means the process of distributing a node's edges to the available port candidates. All edges connecting to a node that has a set of port candidates associated with it via a [PortCandidateSet](#) object are distributed among the available port candidates with respect to:

- the costs of a given port candidate
- the number of edges that are allowed to connect to a given port candidate

For example, when the limit of allowed edges for a given port candidate with costs k is reached, i.e., the given port candidate is said to be "saturated," then the next least expensive port candidate among the remaining ones is chosen to connect edges to.

To influence the matching process, a subset of the PortCandidate objects used for a node can additionally be associated with the respective ports of its connecting edges. The subset then defines a restricted set of desired port candidates an edge prefers to connect to. The PortCandidate objects can be combined using `java.util.Collection` objects which are stored by means of data providers. The data providers are registered with the graph using the look-up keys `SOURCE_PCLIST_DPKEY` and `TARGET_PCLIST_DPKEY`.

[Table 5.7, “Layout support for port candidates” \[62\]](#)lists the layout algorithms that provide support for port candidates.

**Table 5.7. Layout support for port candidates**

Layout Style	Classname	Note
Hierarchical	<a href="#">IncrementalHierarchicLayouter</a>	Incremental hierarchical layout supports port candidates as soon as they are set. See the description of <a href="#">incremental hierarchical layout</a> for more information.

[Table 5.8, “Routing support for port candidates” \[62\]](#)lists the routing algorithms that provide support for port candidates.

**Table 5.8. Routing support for port candidates**

Routing Style	Classname	Note
Orthogonal	<a href="#">OrthogonalEdgeRouter</a> , <a href="#">ChannelEdgeRouter</a>	Both OrthogonalEdgeRouter and ChannelEdgeRouter by default support port candidates as soon as they are set. See the descriptions of <a href="#">OrthogonalEdgeRouter</a> and <a href="#">ChannelEdgeRouter</a> for more information.

## Setup for Layout

[Example 5.12, “Creating port candidate sets for nodes” \[62\]](#)demonstrates how a port candidate set is declared, and how a data provider (adapter) is registered with the graph using the look-up key `NODE_DP_KEY`. During layout calculation, an algorithm first retrieves the data provider using the look-up key, and afterwards retrieves the contained information.

### Example 5.12. Creating port candidate sets for nodes

```
// 'graph' is of type y.layout.LayoutGraph.

// Define a port candidate set that is used for all nodes.
final PortCandidateSet pcs = new PortCandidateSet();
// Create port candidates that conceptually correspond to classic side
// constraints (weak constraints) and add them to the set.
// North side: no costs associated; allows two edges to connect.
pcs.add(PortCandidate.createCandidate(PortConstraint.NORTH), 2);
// East side: costs of 1.0; allows four edges to connect.
pcs.add(PortCandidate.createCandidate(PortConstraint.EAST, 1.0), 4);
// West side: costs of 2.0, but allows an unlimited number of edges to connect.
pcs.add(PortCandidate.createCandidate(PortConstraint.WEST, 2.0),
        Integer.MAX_VALUE);

// Create a data provider (adapter) that returns the port candidate set for
// each node.
DataProvider dp = new DataProviderAdapter() {
    public Object get(Object dataHolder) {
        return ((dataHolder instanceof Node) ? pcs : null);
    }
};
// Register the data provider (adapter) with the graph.
// Use the "well-known" look-up key defined in class PortCandidateSet.
graph.addDataProvider(PortCandidateSet.NODE_DP_KEY, dp);

// Invoke buffered layout.
```

```
invokeBufferedLayout(graph, new IncrementalHierarchicLayouter());
```

## Edge/Port Grouping (Bus-style Edge Routing)

Edge grouping means bundling of a set of edges to be treated in a common manner regarding some aspects of edge path generation. Specifically, if edges at a common source node, for example, are declared an edge group at their source ends, then their source ports will be anchored at the same location.

Additionally, all grouped edges will also be routed in bus-style, i.e., their paths will share a common edge segment. If edges at different source (target) nodes are declared an edge group at their source (target) ends, then they will be routed in bus-style only.

### Note

Edge grouping is also referred to as port grouping sometimes. If edges from an edge group have associated inconsistent, or even contradicting port constraints, then the location of the common port is not guaranteed to obey any of them.

[Table 5.9, “Layout support for edge groups” \[63\]](#) lists the layout algorithms that provide support for edge groups.

**Table 5.9. Layout support for edge groups**

Layout Style	Classname	Note
Hierarchical	<a href="#">HierarchicLayouter</a> , <a href="#">HierarchicGroupLayouter</a> , <a href="#">IncrementalHierarchicLayouter</a>	The hierarchical layout algorithm by default generates edge/port groups as soon as they are declared. See the descriptions of <a href="#">classic hierarchical layout</a> and <a href="#">incremental hierarchical layout</a> for more information.
Orthogonal	<a href="#">DirectedOrthogonalLayouter</a>	The directed orthogonal layout algorithm by default generates edge/port groups as soon as they are declared. See the description of <a href="#">directed orthogonal layout</a> for more information.

[Table 5.10, “Routing support for edge groups” \[63\]](#) lists the routing algorithms that provide support for edge groups.

**Table 5.10. Routing support for edge groups**

Routing Style	Classname	Note
Orthogonal	<a href="#">OrthogonalEdgeRouter</a>	Support for edge/port groups is partly handled by prepended layout stages. See the description of <a href="#">the orthogonal edge router</a> for more information.

## Setup for Layout

[Example 5.13, “Creating an edge group at a common target node” \[63\]](#) demonstrates how edge groups are declared, and how an edge map is registered as a data provider with the graph using the look-up key `TARGET_GROUPID_KEY`. During layout calculation, an algorithm first retrieves the data provider using the look-up key, and afterwards retrieves the contained information.

### Example 5.13. Creating an edge group at a common target node

```
// 'graph' is of type y.layout.LayoutGraph.
// Create an edge map that is used as a data provider later on.
EdgeMap egMap = graph.createEdgeMap();

// Declare an edge group for the target end of all incoming edges at a specific
// node.
String targetEdgeGroupID = "All my grouped edges.";
for (EdgeCursor ec = specificNode.inEdges(); ec.ok(); ec.next()) {
```

## Edge/Port Grouping (Bus-style Edge Routing)

---

```
    egMap.set(ec.edge(), targetEdgeGroupID);  
}  
  
// Register the edge map as a data provider with the graph.  
// Use the "well-known" look-up key defined in interface PortConstraintKeys.  
graph.addDataProvider(PortConstraintKeys.TARGET_GROUPID_KEY, egMap);  
  
// Invoke buffered layout.  
invokeBufferedLayout(graph, new HierarchicLayouter());
```

# Layout Stages

A layout stage, as already explained in the section called “The Layout Stages Concept” [52], serves as a standardized container that encapsulates arbitrary layout functionality and provides a general means to string together multiple layout stages into a compound layout process. It is an implementation of interface `LayoutStage` and most commonly also an extension of abstract class `AbstractLayoutStage`.

The encapsulated layout functionality of the predefined yFiles layout stages covers special-purpose layouts (see Table 5.12, “Layout stages that provide special-purpose layout functionality” [?]) as well as enhancements to the layout process of specific layout styles (for example, tree layout) and edge routing styles, respectively (see orthogonal edge routing).

Table 5.11, “Predefined layout stages” [65] lists another kind of layout stages that contribute to the inner workings of a layout process itself.

**Table 5.11. Predefined layout stages**

Classname	Description
<code>ComponentLayouter</code>	Arranges the connected components of a graph. Part of the default compound layout process as established by class <code>CanonicalMultiStageLayouter</code> .
<code>CompositeLayoutStage</code>	Allows to establish a custom compound layout process consisting of multiple layout stages.
<code>FixedGroupLayoutStage</code>	Used to prevent an existing layout inside a group node from being altered when calculating a layout for a hierarchically organized graph.
<code>LayoutMultiplexer</code>	Enables using individual layout algorithms for components (when used as the core layouter of class <code>ComponentLayouter</code> ) or for the contents of group nodes (when used as the core layouter of class <code>RecursiveGroupLayouter</code> ).
<code>RecursiveGroupLayouter</code>	Provides support for recursively calculating a layout for hierarchically organized graphs.

Purpose and usage of `ComponentLayouter` is described in the section called “Class `ComponentLayouter`” [67]. Note that this section also shows how to use class `LayoutMultiplexer` in conjunction with `ComponentLayouter` in order to have different layout algorithms applied to the separate components of a graph.

Using class `CompositeLayoutStage` to create a custom layout process is shown in Example 5.14, “Defining a custom compound layout process using `CompositeLayoutStage`” [65].

## Example 5.14. Defining a custom compound layout process using `CompositeLayoutStage`

```
// 'hierarchic' is of type y.layout.hierarchic.HierarchicLayouter.
// String together a label layout process.
CompositeLayoutStage ll = new CompositeLayoutStage();
ll.appendStage(new LabelLayoutTranslator());
ll.appendStage(new LabelLayoutDataRefinement());
// Set the compound label layout process as the label layouter for
// HierarchicLayouter.
hierarchic.setLabelLayouter(ll);
hierarchic.setLabelLayouterEnabled(true);
```

## Tutorial Demo Code

The following tutorial demo code has examples on how to use `FixedGroupLayoutStage` and `RecursiveGroupLayouter`:

- `RecursiveGroupLayouterDemo.java` shows how `RecursiveGroupLayouter` can be used when calculating a layout for a hierarchically organized graph

- [HierarchicalLayoutModule.java](#) presents the technique of prepending a hierarchical layout with an invocation of FixedGroupLayoutStage in order to avoid altering the layout of group node contents

Table 5.12, “Layout stages that provide special-purpose layout functionality” [66] lists layout stages that provide special-purpose layout functionality.

**Table 5.12. Layout stages that provide special-purpose layout functionality**

Classname	Description
<a href="#">GraphLayoutLineWrapper</a>	"Line-wraps" or "column-wraps" a graph layout. Best suited for use in conjunction with hierarchic layout.

The [description of class GraphLayoutLineWrapper](#) explains the options of this layout stage and how to use it.

# Class ComponentLayouter

Class `ComponentLayouter` provides services for arranging the connected components of a graph. When invoked, it calls its core layouter on each of the input graph's components. After the core layouter has successively performed its layout processes, `ComponentLayouter` regains control and arranges the separate components.

Wrapping a core layouter with a `ComponentLayouter` object has several benefits:

- The core layouter needs not be able to handle non-connected graphs itself. It can safely assume that its input graph is connected. This often simplifies the formulation of a core layouter substantially.
- The core layouter needs not be able to arrange the separate components of an input graph. This lies in the `ComponentLayouter`'s responsibility.
- Customization of the layout strategy used for arranging separate components can conveniently be done by tailoring, or even exchanging, the `ComponentLayouter`. All core layouters can take advantage of this customization immediately.
- `ComponentLayouter` objects can be used in conjunction with arbitrary core layouters, since the services provided by `ComponentLayouter` are highly reusable.

## Supplemental Layout Data

Class `ComponentLayouter` knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.13, “Data provider look-up keys” \[67\]](#) lists all look-up keys for `ComponentLayouter`.

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.13. Data provider look-up keys**

Key	Element Type	Value Type	Description
<code>LAY-OUT_NODE_DPKEY</code>	Node	boolean	For each node a boolean value indicating whether it should be layouted or not.
<code>GIVEN_COMPONENT_ID_DPKEY</code>	Node	Object	For each node an arbitrary Object indicating the component it is affiliated with.

## Layout Options

Class `ComponentLayouter` provides a set of options that influence its behavior.

### Component Arrangement Style ([see API](#))

Specifies the style for arranging the components of a graph. `ComponentLayouter` supports a variety of different arrangement styles which can be set as shown in [Example 5.15, “Arranging components” \[67\]](#)

### Example 5.15. Arranging components

```
// 'graph' is of type y.layout.LayoutGraph.

OrganicLayouter ol = new OrganicLayouter();
ComponentLayouter cl = (ComponentLayouter)ol.getComponentLayouter();

// Set another arrangement style instead of the default
// ComponentLayouter.STYLE_ROWS.
cl.setStyle(ComponentLayouter.STYLE_PACKED_COMPACT_RECTANGLE);

// Buffered layout with component arrangement.
new BufferedLayouter(cl).doLayout(graph);
```

### **Component Spacing (see API)**

Defines the minimum distance between the bounding boxes of adjacent components.

### **Label Awareness (see API)**

Determines whether node and edge labels are taken into account when computing the bounding boxes of components.

### **Grid Spacing (see API)**

Defines the spacing of the grid on which the separate components are placed.

### **Preferred Layout Size (see API)**

Defines the preferred size of the layout.

### **Arranging Components (see API)**

Determines whether the separate components should be arranged or not.

### **Support for Hierarchically Organized Graphs (see API)**

Specifies whether the graph's hierarchical organization is taken into account when determining its components.

## **Advanced Layout Techniques**

Class ComponentLayouter offers the possibility to call the core layouter only on a subset of the graph's components, effectively excluding the coset of components from layout calculation. This technique is presented in Example 5.16, “Excluding components from layout calculation” [68]

### **Example 5.16. Excluding components from layout calculation**

```
// 'graph' is of type y.layout.LayoutGraph.

NodeList[] components = GraphConnectivity.connectedComponents(graph);

// Create a node map that will be used as a data provider to hold boolean
// values indicating whether a component should be considered for layout
// calculation.
NodeMap nm = graph.createNodeMap();
// Register the node map with the graph using the special look-up key defined
// by class ComponentLayouter.
graph.addDataProvider(ComponentLayouter.LAYOUT_NODE_DKEY, nm);

// Every second component of the graph is marked so that a layout is calculated
// for it. The other components, while being arranged nevertheless, are not
// considered for layout calculation.
for (int i = 0; i < components.length; i += 2)
    nm.setBool(components[i].firstNode(), true);

// The default ComponentLayouter stage of OrganicLayouter arranges all
// components.
new BufferedLayouter(new OrganicLayouter()).doLayout(graph);

// Remove the data provider from the graph.
graph.removeDataProvider(nm);
```

Example 5.17, “Using different layout algorithms for components” [68] shows how ComponentLayouter can be used in conjunction with class `LayoutMultiplexer` to invoke different layout algorithms on the separate components of a graph.

### **Example 5.17. Using different layout algorithms for components**

```
// 'graph' is of type y.layout.LayoutGraph.

NodeList[] components = GraphConnectivity.connectedComponents(graph);

// Create a node map that will be used as a data provider to hold a Layouter
// implementation for each of the components. This layouter will be used for
// layout calculation.
NodeMap nm = graph.createNodeMap();
// Register the node map with the graph using the special look-up key defined
// by class LayoutMultiplexer.
```

```
graph.addDataProvider(LayoutMultiplexer.LAYOUTER_DPKY, nm);

// For each component one of the layouters is set.
Layouter[] coreLayouter = { new HierarchicLayouter(),
                           new SmartOrganicLayouter(),
                           new OrthogonalLayouter() };
for (int i = 0; i < components.length; i++) {
    nm.set(components[i].firstNode(), coreLayouter[i % 3]);
}

// ComponentLayouter uses LayoutMultiplexer as its core layouter, which, for
// each component, invokes the layouter retrieved from the data provider
// registered with the graph.
// Afterwards, the ComponentLayouter nicely arranges the components.
ComponentLayouter cl = new ComponentLayouter();
cl.setCoreLayouter(new LayoutMultiplexer());
new BufferedLayouter(cl).doLayout(graph);

// Remove the data provider from the graph.
graph.removeDataProvider(nm);
```

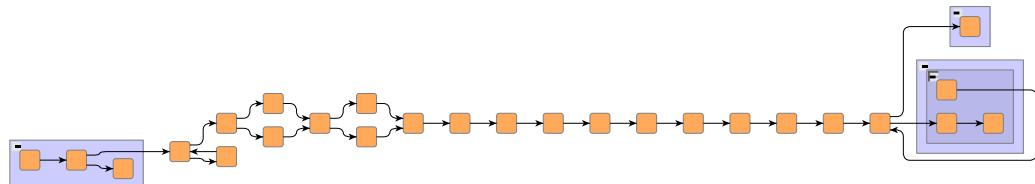
## Class GraphLayoutLineWrapper

Class [GraphLayoutLineWrapper](#) is a layout algorithm that can be used to "line-wrap" respectively "column-wrap" a graph layout. It allows, for example, to specify a desired aspect ratio for a wrapped layout, or a fixed width (height) for the lines (columns) of a wrapped layout.

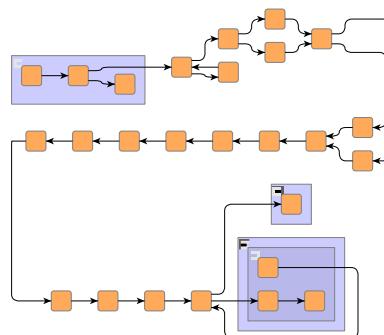
[GraphLayoutLineWrapper](#) is realized as a layout stage, i.e., it is an extension of abstract class [AbstractLayoutStage](#).

Most commonly, class [GraphLayoutLineWrapper](#) is used in conjunction with hierarchical layouts. [Figure 5.8, “Hierarchical layout before and after line-wrapping” \[70\]](#) presents a hierarchical layout that has been line-wrapped to achieve a desired aspect ratio.

**Figure 5.8. Hierarchical layout before and after line-wrapping**



Extended hierarchical left-to-right layout.



The same graph line-wrapped so that width and height are nearly the same, i.e., an aspect ratio of 1.0 is nearly achieved.

[Example 5.18, “Using GraphLayoutLineWrapper in conjunction with hierarchical layout” \[70\]](#) shows how class [GraphLayoutLineWrapper](#) can be used to wrap hierarchical top-to-bottom layouts.

### Example 5.18. Using GraphLayoutLineWrapper in conjunction with hierarchical layout

```
// 'graph' is of type y.layout.LayoutGraph.

GraphLayoutLineWrapper gllw = new GraphLayoutLineWrapper();
gllw.setColumnMode(true);
gllw.setMirroringEnabled(false);
// Use grouping-aware hierarchical layout as the core layouter.
HierarchicGroupLayouter hgl = new HierarchicGroupLayouter();
hgl.setRoutingStyle(HierarchicGroupLayouter.ROUTE_ORTHOGONAL);
gllw.setCoreLayouter(hgl);

// Use buffered layout.
GroupLayoutConfigurator glc = new GroupLayoutConfigurator(graph);
glc.prepareAll();
new BufferedLayouter(gllw).doLayout(graph);
glc.restoreAll();
```

### Layout Options

GraphLayoutLineWrapper provides a set of options that influence its behavior.

#### Column Mode ([see API](#))

Determines whether the graph layout should be column-wrapped instead of line-wrapped.

#### Spacing ([see API](#))

Defines the spacing between adjacent lines (columns) of the line-wrapped (column-wrapped) graph layout.

#### Edge Spacing ([see API](#))

Defines the spacing between adjacent edge paths that connect consecutive lines (columns) of the line-wrapped (column-wrapped) graph layout.

#### Mirroring ([see API](#))

Determines whether the lines (columns) of the line-wrapped (column-wrapped) graph layout should be arranged in an alternating manner so that every second line (column) goes from right to left (bottom to top) instead of left to right (top to bottom).

#### Pre-set vs. Automatic Line Width (Column Height) ([see API](#))

Determines whether the lines (columns) of the line-wrapped (column-wrapped) graph layout should use the width (height) as set using `setFixedWidth(double)`.

#### Fixed Line Width (Column Height) ([see API](#))

Defines the width (height) that should be used for the lines (columns) of the line-wrapped (column-wrapped) graph layout. This setting only has an effect if pre-set line width (column height) is enabled.

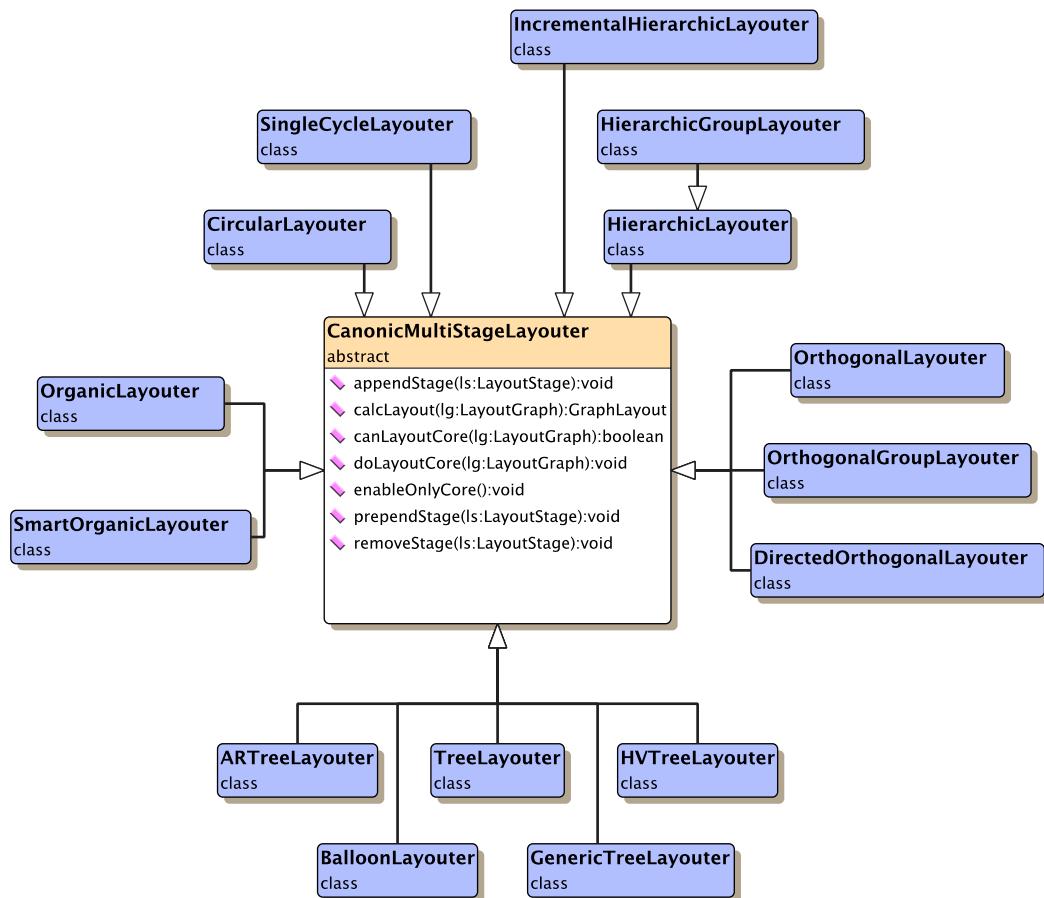
#### Aspect Ratio ([see API](#))

Defines the desired aspect ratio that should be achieved for the line-wrapped (column-wrapped) graph layout. This setting only has an effect if automatic line width (column height) is enabled.

# Major Layout Algorithms

The major layout algorithms provided by yFiles inherit from abstract class `CanonicMultiStageLayouter`. This base class provides many useful layout stages, which, e.g., route self-loops and parallel edges, or arrange non-connected graph components. [Figure 5.9, “The yFiles layout algorithms” \[72\]](#) shows the yFiles major layout algorithms.

**Figure 5.9. The yFiles layout algorithms**



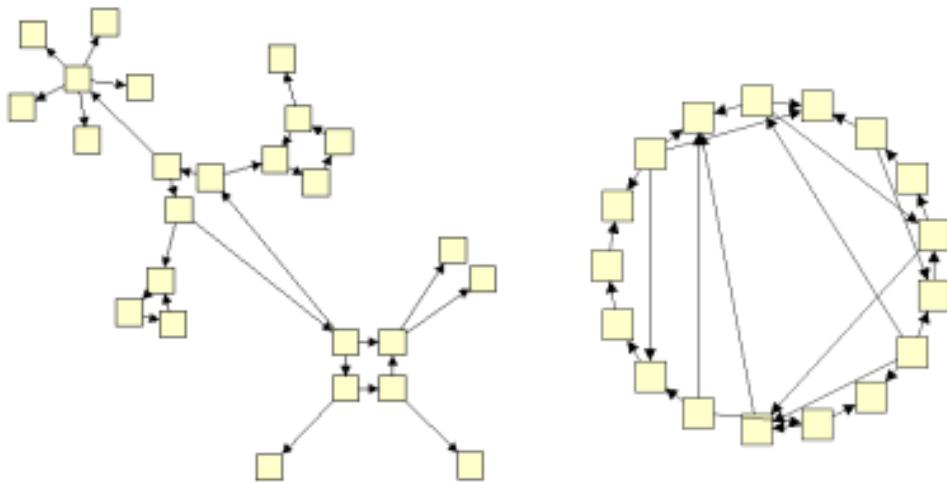
yFiles supports different major layout styles:

- Circular Layout
- Hierarchical Layout
- Organic Layout
- Orthogonal Layout
- Tree Layout

## Circular Layout

Circular layout is described in [the section called “Circular Layout” \[78\]](#)

**Figure 5.10. Sample circular layouts**

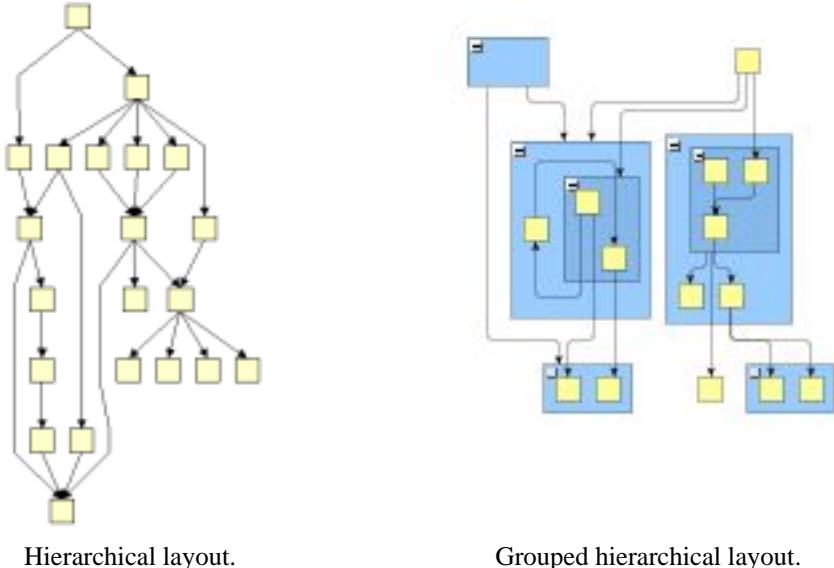


Encapsulated in class [CircularLayouter](#) and class [SingleCycleLayouter](#).

## Hierarchical Layout

Classic hierarchical layout is described in [the section called “Hierarchical Layout” \[83\]](#) incremental hierarchical layout in [the section called “Incremental Hierarchical Layout” \[97\]](#)

**Figure 5.11. Sample hierarchical layouts**



Hierarchical layout.

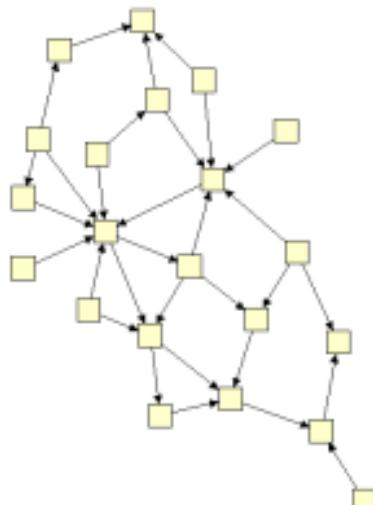
Grouped hierarchical layout.

Encapsulated in classes [HierarchicLayouter](#) and [HierarchicGroupLayouter](#), and in class [IncrementalHierarchicLayouter](#).

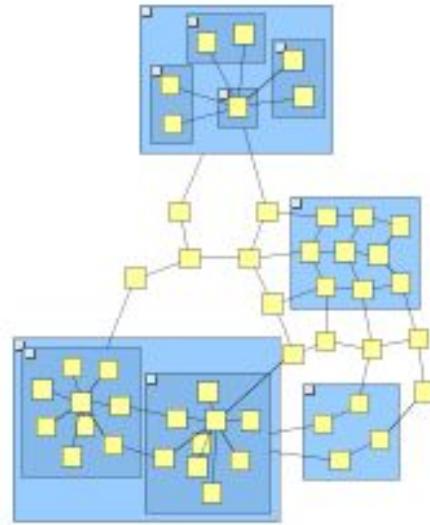
## Organic Layout

Plain organic layout is described in [the section called “Organic Layout” \[106\]](#) smart organic layout in [the section called “Smart Organic Layout” \[112\]](#) The description of interactive organic layout can be found in [the section called “Interactive Organic Layout” \[115\]](#)

**Figure 5.12. Sample organic layouts**



Organic layout.



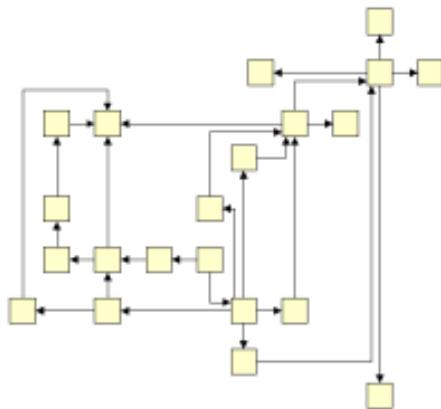
Grouped organic layout.

Encapsulated in the classes [OrganicLayouter](#) and [SmartOrganicLayouter](#). Class [InteractiveOrganicLayouter](#) encapsulates the interactive variant of organic layout.

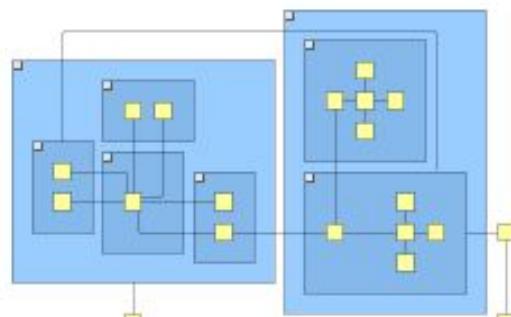
## Orthogonal Layout

Plain orthogonal layout is described in the section called “Orthogonal Layout” [121] the modification thereof that supports hierarchically organized graphs in the section called “Orthogonal Layout of Hierarchically Organized Graphs” [124]. Special support for UML-style layout is provided by the orthogonal layout variant found in the section called “Directed Orthogonal Layout” [126].

**Figure 5.13. Sample orthogonal layouts**



Orthogonal layout.



Grouped orthogonal layout.

Encapsulated in class [OrthogonalLayouter](#) and [OrthogonalGroupLayouter](#). Class [DirectedOrthogonalLayouter](#) encapsulates UML-style orthogonal layout.

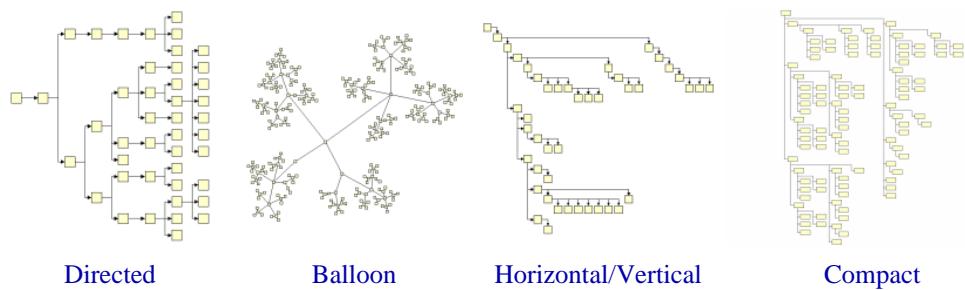
## Tree Layout

Tree layout is described in the section called “Tree Layout” [130]

**Figure 5.14. Sample tree layouts**

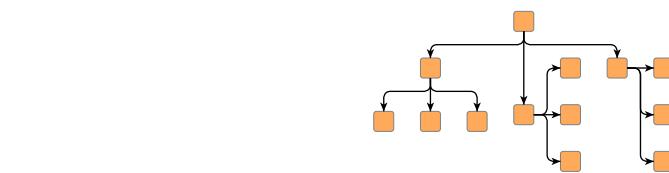
## Tree Layout

---



Encapsulated in the classes `TreeLayouter`, `BalloonLayouter`, `HVTreeLayouter`, and `ARTreeLayouter`. Also, class `GenericTreeLayouter` encapsulates tree layout, too. The generic tree layout algorithm is described in the section called “[Generic Tree Layout](#)” [137]

**Figure 5.15. Sample tree layouts (continued)**



Generic Tree Layout

## Incremental Layout

The yFiles library provides unequaled support for incremental graph layout that seamlessly integrates with the "normal," i.e., non-incremental major layout algorithms.

Compared to a "normal" layout algorithm, which computes an all-new, fresh graph layout each time it is invoked, the main advantage of incremental layout is that distinct parts of a graph can be rearranged while the remainder of the graph is not, or only slightly, changed. Especially in an interactive environment, where a human user modifies a graph, i.e., adds and/or removes graph elements, this technique makes it possible to maintain the user's so-called "mental map" over a course of subsequent graph layouts.

The term "mental map" expresses a user's experience of a graph, which is most notably influenced by a consistent and largely invariable placement of a graph's elements in a sequence of layout algorithm invocations. "Normal" layout calculation does not consider the mental map in any way, in fact it is not unlikely that a graph's layout changes substantially for modifications as small as adding a single edge!

[Table 5.14, "Layout support for incrementally modified graphs"](#) [76] lists the layout algorithms that provide support for incremental graph layout.

**Table 5.14. Layout support for incrementally modified graphs**

Layout Style	Classname	Note
Hierarchical	<a href="#">IncrementalHierarchicalLayouter</a>	Supports incremental hierarchical layout as well as classic hierarchical layout. See the <a href="#">description of incremental hierarchical layout</a> for more information.
Organic	<a href="#">OrganicLayouter</a> , <a href="#">SmartOrganicLayouter</a>	Incremental layout support is provided by means of the "Sphere of Action" (OrganicLayouter) and "Scope" (SmartOrganicLayouter) features, respectively. See the layout options of <a href="#">organic layout</a> and <a href="#">smart organic layout</a> for more information.
Tree	<a href="#">TreeLayouter</a> , <a href="#">BalloonLayouter</a> , <a href="#">GenericTreeLayouter</a>	Support for incremental layout is provided by TreeLayouter by means of the default "child comparator" implementation. BalloonLayouter can be set to "layout from sketch" mode. GenericTreeLayouter supports incremental layout by means of the default NodePlacer implementation. See the descriptions of <a href="#">tree layout</a> , <a href="#">balloon layout</a> , and <a href="#">generic tree layout</a> , respectively for more information.

[Table 5.15, "Routing support for incrementally modified graphs"](#) [76] lists the routing algorithms that provide support for incremental edge routing.

**Table 5.15. Routing support for incrementally modified graphs**

Routing Style	Classname	Note
Organic	<a href="#">OrganicEdgeRouter</a>	Incremental routing support is provided by the "Route Selected Edge Only" feature. See the <a href="#">routing options of organic edge routing</a> for more information.
Orthogonal	<a href="#">OrthogonalEdgeRouter</a> , <a href="#">ChannelEdgeRouter</a>	Incremental routing support is provided by means of the "Sphere of Action" (OrthogonalEdgeRouter) and "Affected Edges" (ChannelEdgeRouter) features, respectively. See the routing options of <a href="#">orthogonal edge routing</a> and <a href="#">channel edge routing</a> for more information.

## Use Cases

Incremental layout is closely related to "layout from sketch," where a given arrangement of nodes is taken as the starting point for a layout calculation and also as a specification for the calculation's desired outcome. Consequently, "layout from sketch" takes a major role in an incremental layout algorithm's functionality.

Incremental layout has two major use cases, which both involve "layout from sketch:"

- Interactive creation of a graph structure where the layout calculation is performed dynamically with each newly inserted graph element.
- Subsequent improvement of distinct parts from an already existing graph layout, where the remainder of the layout stays mainly unchanged.

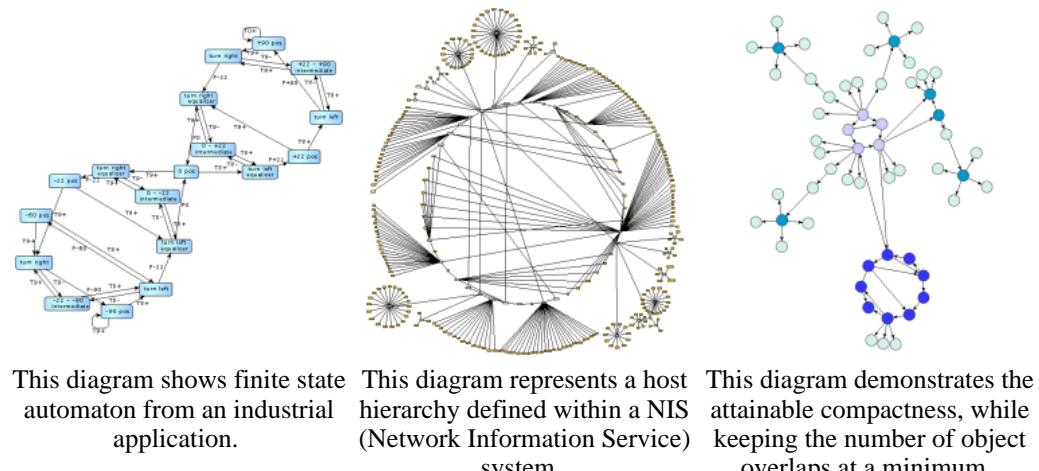
## Circular Layout

Class [CircularLayouter](#) is a layout algorithm that portraits interconnected ring and star topologies and is excellent for applications in:

- Social networking (criminology, economics, etc.)
- Network management
- WWW visualization
- eCommerce

[CircularLayouter](#) produces layouts that emphasize group and tree structures within a network. It creates node partitions by analyzing the connectivity structure of the network, and arranges the partitions as separate circles. The circles themselves are arranged in a radial tree layout fashion. [CircularLayouter](#) delegates these two major layout tasks to other, more specialized layout providers, namely classes [SingleCycleLayouter](#) and [BalloonLayouter](#).

**Figure 5.16. Sample layouts produced by class CircularLayouter**



This diagram shows finite state automaton from an industrial application.

This diagram represents a host hierarchy defined within a NIS (Network Information Service) system.

This diagram demonstrates the attainable compactness, while keeping the number of object overlaps at a minimum.

## Supplemental Layout Data

Class CircularLayouter knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.16, “Data provider look-up keys”](#) [?] lists all look-up keys for CircularLayouter.

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data”](#) [55].

**Table 5.16. Data provider look-up keys**

Key	Element Type	Value Type	Description
<a href="#">CIRCULAR_CUSTOM_GROUPS_DPKY</a>	Node	Object	For each node an arbitrary Object indicating the group it is affiliated with.

## Layout Options

Class CircularLayouter provides a set of options that influence its layout behavior. These options can be set using the setter methods of class CircularLayouter. The options are documented within the API documentation of class [CircularLayouter](#).

Additionally, there are getter methods that grant access to the layout delegates `SingleCycleLayouter` and `BalloonLayouter`. These layouters are responsible for arranging the circle and the tree components of the diagram, respectively. Modifying the options of these layout algorithms does also affect the overall layout behavior of `CircularLayouter`.

The options for `CircularLayouter` influence the general layout strategy and the interplay between `SingleCycleLayouter` and `BalloonTreeLayouter`.

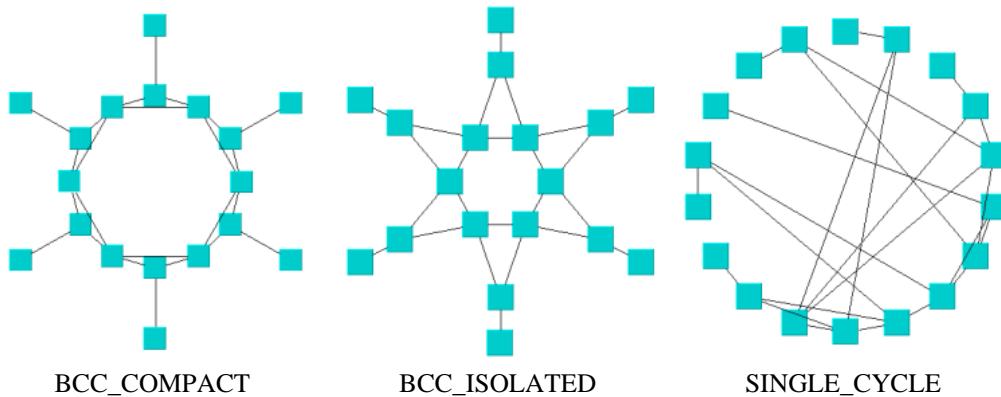
### Layout Style (see API)

Controls which policy is used to partition the nodes of the graph. Each node partition will be arranged either on a separate circle or as a separate "disk" (see also [node partition layout style](#) below). Available options are

BCC Compact	Each partition will represent a so-called biconnected component of the graph. A biconnected component consists of nodes that are reachable by two edge-disjoint paths. Nodes that belong to more than one biconnected component will be assigned exclusively to one partition.
BCC Isolated	Node partitions will be formed as with "BCC Compact" with the difference that all nodes belonging to more than one biconnected component will be assigned an isolated partition.
Single Cycle	All nodes will be arranged on a single circle.
Custom Groups	The node partitions that form circles will be given by the user. To specify the partitions, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key <code>CIRCULAR_CUSTOM_GROUPS_DPKY</code> .

[Figure 5.17, "Effect of option Layout Style" \[79\]](#) shows the effect of different layout styles applied to `CircularLayouter`. Possible values for that option are `BCC_COMPACT`, `BCC_ISOLATED`, and `SINGLE_CYCLE`. The BCC styles partition the nodes by examining the two-connectivity property of the input graph. `SINGLE_CYCLE` places all nodes on a single cycle.

**Figure 5.17. Effect of option Layout Style**



### Use Existing Drawing as Sketch (see API)

If enabled, the layout algorithm interprets the initial graph layout as a sketch for the desired outcome of the layout process. The algorithm tries to identify the nodes that should lie on a node partition's circle/disk boundary and maintain their cyclic order around the center of the node partition. (See also the section called "Partition Arrangement Options" [79] below.) Additionally, when layout style BCC Compact is used, the layout algorithm also tries to maintain the cyclic order of entire node partitions that connect to a common node.

## Partition Arrangement Options

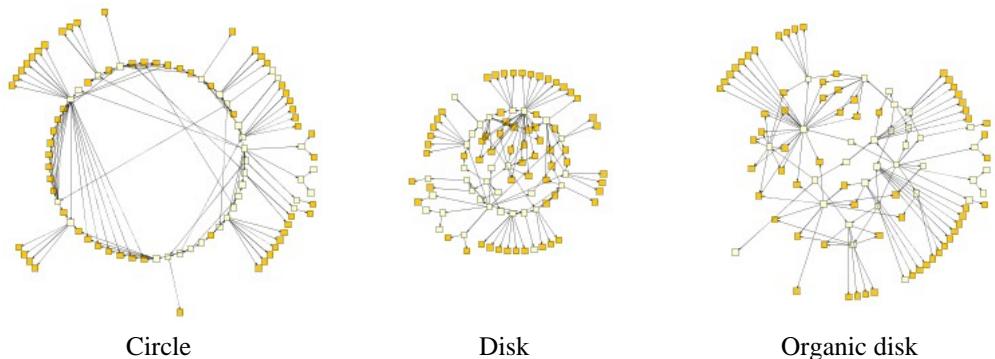
### Node Partition Layout Style (see API)

Determines the layout style for the node partitions of a graph. The nodes from a partition can be arranged either as a circle or using one of the available "disk" flavors:

- Arranging as a disk means that those nodes from a given partition which connect to nodes from other partitions are placed on the disk's boundary while all other nodes are placed on the inside.
- Arranging as an organic disk means that the organic layout paradigm is used to arrange the nodes from a partition. Nodes which connect to nodes from other partitions can be placed both on the inside or on the boundary of the disk.

Both disk arrangement styles result in graph layouts that are smaller in size, i.e., consume less space than circle arrangement style. [Figure 5.18, “Node partition layout styles” \[80\]](#) presents the available node partition layout styles and their effects for a given graph.

**Figure 5.18. Node partition layout styles**



The options for the [internally used SingleCycleLayouter](#) influence the size and compactness of the circles/disks formed by this layouter.

### Minimal Node Distance ([see API](#))

Determines the minimal distance between the borders of two adjacent nodes on the circle/the disk's boundary. The smaller the distance, the more compact the resulting layout.

### Automatic Radius ([see API](#))

Whether or not to determine the radius of each circle/disk in the layout automatically. An automatically chosen radius is usually the smallest possible radius that obeys "Minimal Node Distance."

### Fixed Radius ([see API](#))

If "Automatic Radius" is not set, then this option determines the fixed radius for all circles/disks in the resulting layout. "Minimal Node Distance" will be ignored in this case.

## Tree Arrangement Options

The options for the [internally used BalloonLayouter](#) influence the compactness of the tree-like components that this layouter produces. The tree-like structures are the parts of the layout that are not part of the circular arranged node groups.

### Preferred Child Wedge ([see API](#))

This setting determines the angular range of the sector that will be reserved for the children of a root node. The possible angular range lies between 1 and 359. The remaining angular range (360 minus x) will be automatically used to accommodate the edge that connects to the root node.

The smaller the chosen value, the greater the impression that the nodes drive away from their root nodes and the center of the graph.

Generally speaking, the compactness of the layout will decrease with smaller values. Very small values will lead to layouts that consume a lot of space.

### Minimal Edge Length (see API)

Determines the minimal length of an edge that connects two nodes that lie on separate circles (tree-edges). The smaller the chosen value, the more compact the resulting layout.

### Maximal Deviation Angle (see API)

Whenever tree edges connect to a node that lies on a circle together with other nodes, the layouter tries to direct that edge in such a way that its prolongation crosses through the center of the circle. This is not always possible though, for example, if more than one tree-edge connect to the same circle node.

This parameter determines the allowed angular deviation from the optimal edge direction as described above. The bigger the chosen value, the more compact the resulting layout. If a value smaller than 90 degrees is chosen, then the tree-edges might cross through the circular arranged components.

### Compactness Factor (see API)

This parameter influences the length of the tree-edges as it is computed by the layouter. The smaller the compactness factor, the shorter the tree-edges and the more compact the overall layout. The bigger the compactness factor the more difficult, and hence slower, the layout computation.

### Allow Overlaps (see API)

If activated, this option further increases compactness of the resulting layout, but potentially introduces slight node overlaps.

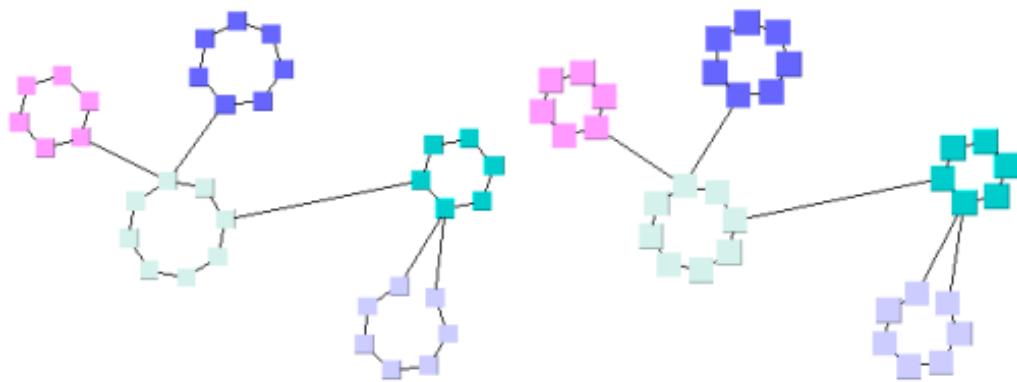
## Compact Layout

There are several options that have impact on the compactness of the circular layout. Optimizing for compactness, e.g., is achieved by

- choosing "Disk" or "Organic Disk" as the Node Partition Layout Style
- reducing "Compactness Factor" of BalloonLayouter (say, to 0.1)
- setting "Allow Overlaps" of BalloonLayouter to `true` (note that this may introduce node overlaps)
- reducing "Minimal Edge Length" of BalloonLayouter
- reducing "Minimal Node Distance" of SingleCycleLayouter
- increasing "Maximum Deviation Angle" of CircularLayouter

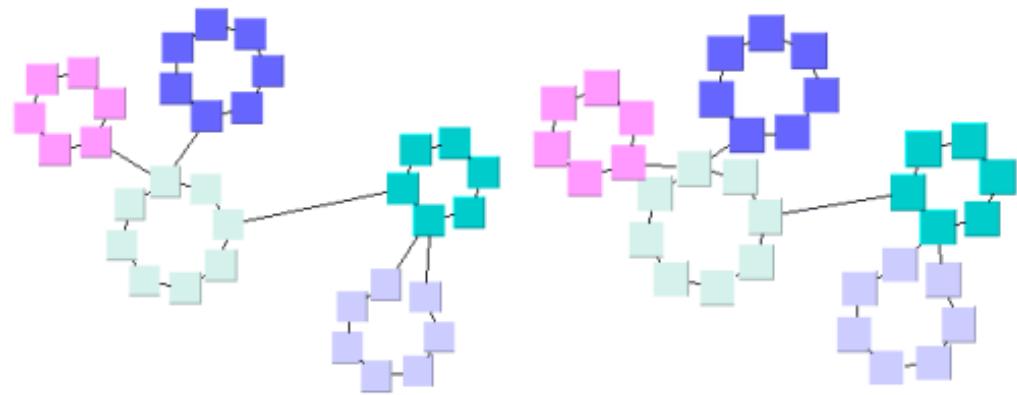
Figure 5.19, “Increasing layout compactness” [81] shows the cumulative effect on layout compactness when changing the settings of the layouter.

**Figure 5.19. Increasing layout compactness**



(a) Default settings

(b) "Minimal Node Distance" = 10.0



(c) "Allow Overlaps" = true

(d) "Minimal Edge Length" = 10

## Tutorial Demo Code

Layout module [CircularLayoutModule.java](#) presents the setup of class CircularLayouter in an application context.

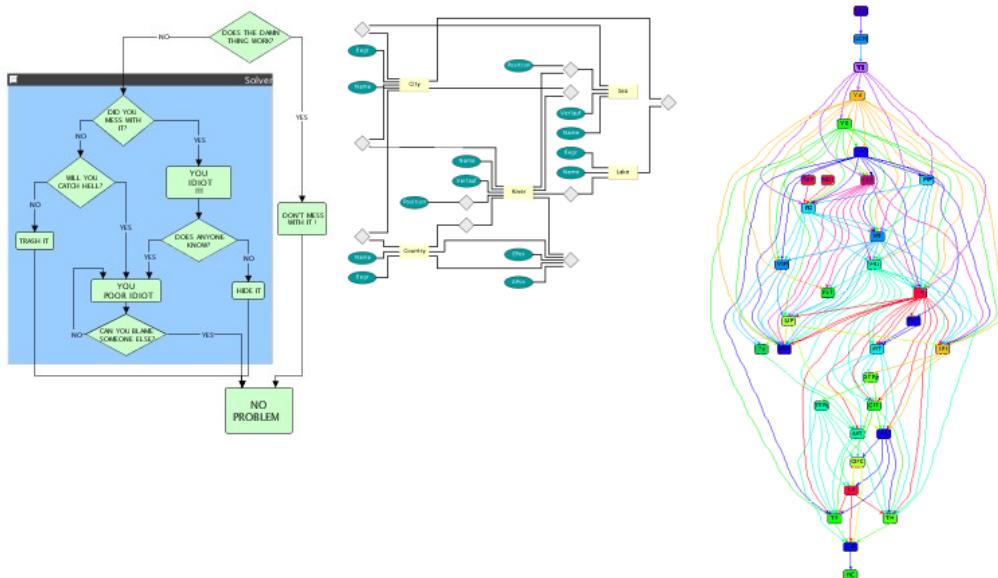
## Hierarchical Layout

Class **HierarhicLayouter** is a layout algorithm that portraits the precedence relation of directed graphs. It is ideal for many application areas, including, e.g.:

- Workflow
- Software engineering
- Customer relationship management
- Configuration management
- Process modeling
- Database modeling
- Bioinformatics

The hierarchical layout algorithm aims to highlight the main direction or *flow* within a directed graph. Cyclic dependencies of nodes will be automatically detected and resolved. Nodes will be placed in hierarchically arranged layers. Additionally, the ordering of the nodes within each layer is chosen in such a way that the number of line (or edge) crossings is small. Edge routing can be polyline, orthogonal, or in a curved style.

**Figure 5.20. Sample layouts produced by class HierarhicLayouter**



Problem solving at yWorks  
(just kidding ;-)

Entity-Relationship Diagram.

Displays pathways of the  
brain, layered by stages of  
visual processing.

## Supplemental Layout Data

Class **HierarhicLayouter** implements interface **PortConstraintKeys** and uses the data provider keys which are defined therein to retrieve supplemental layout data for a graph's elements. Also used are further data provider keys defined by a number of classes from package **y.layout.hierarhic**. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.17, “Data provider look-up keys” \[83\]](#) lists all look-up keys for **HierarhicLayouter**.

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.17. Data provider look-up keys**

## Layout Options

---

Key	Element Type	Value Type	Description
SOURCE_GROUPID_KEY	Edge	Object	For each edge an arbitrary Object indicating the group its source end is affiliated with.
TAR-GET_GROUPID_KEY	Edge	Object	For each edge an arbitrary Object indicating the group its target end is affiliated with.
SOURCE_PORT_CONSTRAINT_KEY	Edge	Port-Constraint	For each edge a PortConstraint object encoding its source end's port constraint.
TAR-GET_PORT_CONSTRAINT_KEY	Edge	Port-Constraint	For each edge a PortConstraint object encoding its target end's port constraint.
CORE_NODES	Node	boolean	For each node a boolean value indicating whether it should be placed into the first layer.
LAYER_ID_KEY	Node	int	For each node an integral value indicating the layer number it should be placed into.
EDGE_LABEL_LAYOUT_OUT_KEY	Edge	LabelLayoutData[]	For each edge an array of LabelLayoutData objects that encode size and preferred placement for all labels of the edge.
GROUP_KEY	Node	Number	For each node a Number object encapsulating an integral value that indicates the packed block the node is affiliated with.

## Layout Options

HierarchicLayouter can be configured and extended in diverse ways. This section highlights some of the configuration options available for this layouter.

## Drawing Style Options

### Minimal Layer Distance (see API)

Determines the minimal distance between nodes that reside in adjacent layers.

### Minimal Node Distance (see API)

Determines the minimal distance between adjacent nodes that reside in the same layer.

### Minimal Edge Distance (see API)

Determines the distance between adjacent pairs of horizontal edge segments, and between horizontal edge segments and nodes.

### Minimal First Segment Length (see API)

Determines the minimal length of the first segment in an orthogonal edge routing. This applies to the routing of self-loops, same-layer edges, and orthogonal inter-layer edge routing.

### Maximal Duration (see API)

Sets the maximal duration of the layout process in milliseconds. The maximal duration is a soft bound, which may be exceeded in case essential parts of the layout calculation are still missing. Most likely the layout quality will increase with a longer calculation period. In most cases the layouter will consume the complete time assigned to it.

### Layout Orientation (see API)

Determines the main layout orientation. The layouter tries to arrange nodes in such a way that all edges point in the main layout direction.

Top to Bottom      The main layout orientation will be from top to bottom. Note that the doc-

umentation for the other layout options assumes that this default layout orientation is being used.

Bottom to Top

The main layout orientation will be from bottom to top.

Left to Right

The main layout orientation will be from left to right.

Right to Left

The main layout orientation will be from right to left.

## Layout Style (see API)

Influences the horizontal spacing between nodes, the number of bends of the edges, and the overall balance of the layout.

Linear Segments

Nodes will be placed in such a way that edge segments tend to have very few bends. This is a very good choice in combination with "Orthogonal Edge Routing." Disadvantage of this placement policy is that the width of the layout will increase.

Polyline

Nodes will be placed in such a way that the width of the layout gets very small without introducing node overlaps. A drawback of this placement policy is that edges will have a lot of bends.

Pendulum

A sound combination of "Linear Segments" and "Polyline."

Simplex

Produces high quality drawings. This node placer is based on the network simplex approach. Similar to "Linear Segments," the nodes will be placed in such a way that edge segments tend to have very few bends. Additionally, the resulting layout will be more balanced and more compact. This node placer tends to be slower than "Linear Segments."

Median Simplex

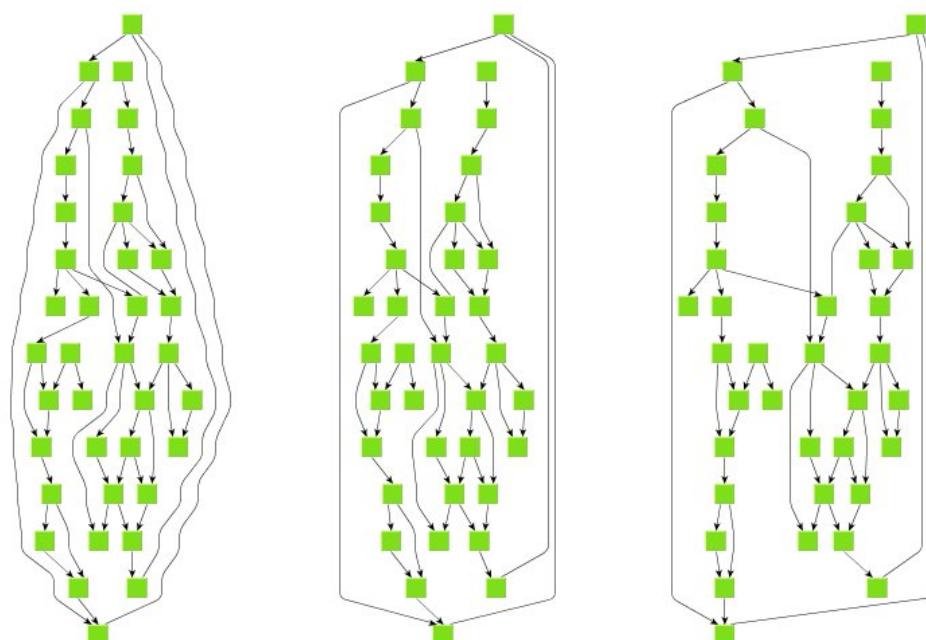
This drawer is based on the "Simplex" drawer, but produces slightly different results. It tends to produce more locally symmetric layouts for the sake of a few more bends.

Tree

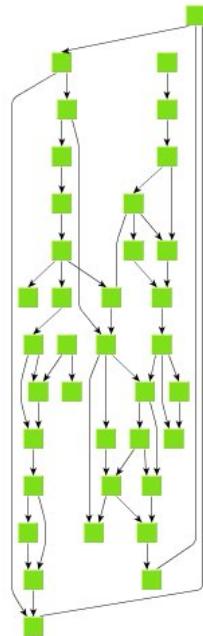
Produces very nice layouts in case the graph is a tree. If the graph is not a tree, the placement policy "Linear Segments" will be used.

Figure 5.21, “Different hierachic layout styles” [85]displays the effect of using different layout styles. There is a tradeoff involved between the straightness of edge paths and the compactness of the diagram.

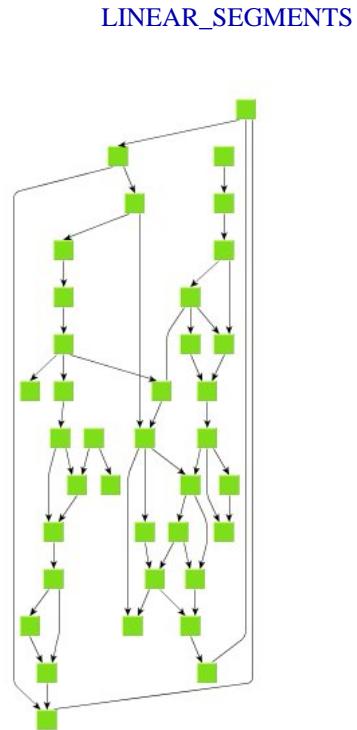
**Figure 5.21. Different hierachic layout styles**



POLYLINE



PENDULUM



SIMPLEX

MEDIAN\_SIMPLEX

## Routing Style (see API)

Determines the edge routing style.

Polyline	Edge paths will be routed as a polyline with a certain number of bends.
Orthogonal	Edge paths will be routed in an orthogonal style, i.e., only vertical and horizontal line segments will be used. Orthogonal edge routing increases the height of the layout.

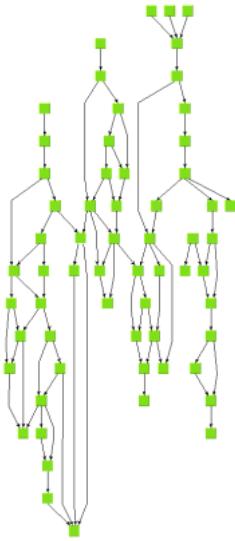
Figure 5.22, “Different routing styles and different edge path realizers” [86] below shows the effect of different routing styles available for class HierarchicLayouter.

### Note

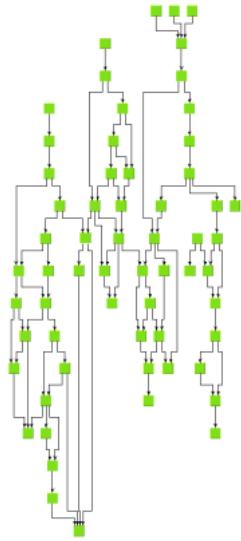
The visual representation of an edge (path) is defined by edge realizers, i.e., objects of type [EdgeRealizer](#) from the yFiles library package [y.view](#). Both type and configuration of a realizer object determine how to draw an edge given its control points.

See also the section called “Edge Realizers” [170] for descriptions of the yFiles predefined EdgeRealizer implementations.

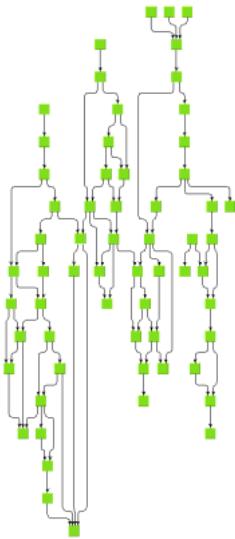
**Figure 5.22. Different routing styles and different edge path realizers**



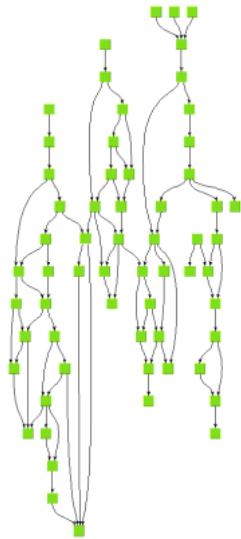
**ROUTE\_POLYLINE** using **PolyLineEdgeRealizer**.



**ROUTE\_ORTHOGONAL** using **PolyLineEdgeRealizer**.



**ROUTE\_ORTHOGONAL** using **PolyLineEdgeRealizer** and **setSmoothedBends()** feature.



**ROUTE\_ORTHOGONAL** using **BzierEdgeRealizer**.

If enabled, all edges that do not point in the main layout direction will be routed as back-loops. Back-loops start at the bottom side of their source node, then turn around 180 degrees, go upward, then turn around 180 degrees again and finally connect to the upper side of their target node.

## Layer Assignment Options

HierachicLayouter assigns the nodes to separate layers. The nodes within each layer will be placed on the same horizontal line. The layers will be numbered from 1 to maxLayer and arranged vertically starting with the small-numbered layers. The rank of a node is the number of the layer it belongs to.

An important layering strategy for the hierachic layout style is called *Hierarchical Layering*. A hierarchical layering tries to assign nodes to layers in a way that as much as possible edges of the graph will point to the main layout direction, i.e., the start nodes of the edges will have a smaller rank than the corresponding end nodes. Also, a hierarchical layering will never put two connected nodes in the same layer.

## Layering Strategy (see API)

Hierarchical - Topmost	A simple hierarchical layering variant. All nodes with in-degree zero will be assigned to the topmost layer of the layout. The number of separate layers will be as small as possible.
Hierarchical - Optimal	An optimal hierarchical layering strategy. The layer distance of an edge is the absolute difference between the layer numbers of its source and target node. Layer assignment will be done in such a way that the overall sum of the layer distances of all edges in the layout is minimal.
Hierarchical - Tight Tree Heuristic	A good heuristic that approximates the ranking done by "Topological - Optimal."
Hierarchical - Downshift Heuristic	An even faster heuristic that approximates the ranking done by "Topological - Optimal" by down-shifting some nodes in the layering. The quality is usually worse than the one produced by "Tight Tree Heuristic."
From Sketch	A very interesting layer assignment strategy that uses the initial y-coordinates of the nodes to determine a node layering. It tries to find a layering that is similar to the one in the input graph. When this layering strategy is used, the layouter may place nodes in the same layer, even though they are connected by an edge. These inner layer edges are always routed in an orthogonal style.
User-Defined Layering	The ranks of the nodes will be given by the user. To specify the ranks, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key <code>LAYER_ID_KEY</code> .
BFS Layering	Layering based on a breadth-first search (BFS). All edges will span at most one layer in the resulting drawing. Edges between nodes that belong to the same layer are possible. To specify nodes that should be placed into the first layer, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key <code>CORE_NODES</code> . Note that in the absence of such a data provider all nodes that have no incoming edges are placed into the first layer.

## Constrained Layer Assignment

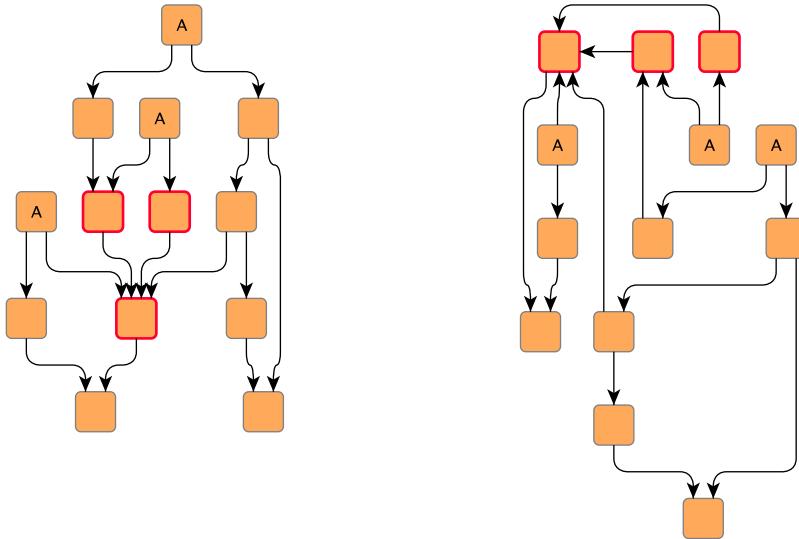
Class `ConstraintLayerer` is an implementation of the `Layerer` interface that enables user-defined constrained layering. Nodes can be restricted to be placed either

- absolute, i.e., into the first or last layer of the layout, or
- relative to a given reference node into the same layer, a layer preceding that of the reference node, or a layer following that of the reference node.

The remaining nodes, that have no constraints defined, are processed by the core layerer that is set with `ConstraintLayerer`. By default, the core layerer uses the Tight Tree heuristic.

Figure 5.23, “Constrained hierarchical layering” [88] shows a resulting hierarchical layout where nodes with an absolute constraint specified for them are placed in the topmost layer (note the emphasis on these nodes). Normally, i.e., when no constraints are specified, these nodes are placed in the very center of the graph as can be observed in the original hierarchical layout.

**Figure 5.23. Constrained hierarchical layering**



Original hierarchical layout.

Resulting hierarchical layout where the constrained nodes are placed in the topmost layer.

[Example 5.19, “Setting the core layerer for ConstraintLayerer” \[89\]](#) presents how hierarchical layout is set up for constrained layer assignment where the previously configured Layerer implementation is used as ConstraintLayerer's core layerer. Nodes that have no constraints defined will be placed according to the layering strategy of this Layerer.

### Example 5.19. Setting the core layerer for ConstraintLayerer

```
// Instantiate hierarchical layout and, in particular, configure the layerer.
HierachicLayouter hl = new HierachicLayouter();
hl.setLayeringStrategy(HierachicLayouter.LAYERING_HIERARCHICAL_DOWNSHIFT);
// Instantiate a ConstraintLayerer.
ConstraintLayerer cl = new ConstraintLayerer();
// Set the previously configured layerer as ConstraintLayerer's core layerer.
cl.setCoreLayerer(hl.getLayerer());
// Set the ConstraintLayerer as the new layerer for hierarchical layout.
hl.setLayerer(cl);
```

To express both absolute and relative constraints, ConstraintLayerer offers a so-called "constraint factory" which provides the methods for specifying constraints for the nodes of a graph. [API Excerpt 5.6, “Getter method for ConstraintFactory” \[89\]](#) shows the static getter method for the constraint factory.

### API Excerpt 5.6. Getter method for ConstraintFactory

```
// Getter for the constraint factory.
static ConstraintFactory createConstraintFactory(Graph graph)
```

The returned constraint factory is specific for the graph given at instantiation. [API Excerpt 5.7, “ConstraintFactory methods for specifying constraints” \[89\]](#) lists the methods that allow to define both absolute and relative constraints for nodes.

### API Excerpt 5.7. ConstraintFactory methods for specifying constraints

```
// Absolute constraints.
void addPlaceNodeAtTopConstraint(Node n)
void addPlaceNodeAtBottomConstraint(Node n)

// Constraints relative to a given reference node.
void addPlaceNodeInSameLayerConstraint(Node reference, Node sameLayer)
```

## Node Order Options

```
void addPlaceNodeAboveConstraint(Node reference, Node above)
void addPlaceNodeBelowConstraint(Node reference, Node below)
```

Using class ConstraintLayerer in a hierarchical layout and defining constraints for the nodes of a graph is demonstrated in the tutorial demo application [ConstraintLayererDemo.java](#).

## Node Order Options

This section describes options that influence the ordering of the nodes within a layer and thus the number of edge crossings of the resulting layout.

Within class HierarchicLayouter an implementation of the interface [LayerSequencer](#) is responsible for determining the order of nodes within a layer. The default implementation of this interface is [ClassicLayerSequencer](#). The options below describe some settings of this class.

### Weight Heuristic (see API)

Determines the strategy used to find a better node ordering within a layer.

- |            |                             |
|------------|-----------------------------|
| Barycenter | Uses the barycenter method. |
| Median     | Uses the median method.     |

### Use Transposition (see API)

If enabled, a postprocessing further reduces the number of edge crossings. The postprocessing step can be rather time consuming. This feature is disabled by default. It is advisable to activate this processing step for high quality layout results.

### Remove False Crossings (see API)

If enabled, a postprocessing step tries to eliminate all false edge crossings. A false edge crossing is an edge crossing between two edges that share a common terminal node.

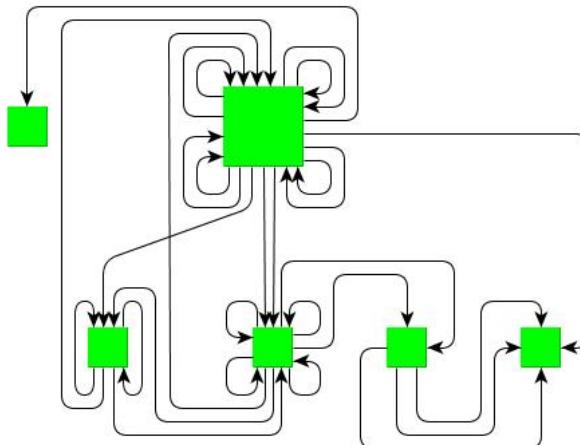
## Advanced Layout Features

### Weak Port Constraints (see API)

The hierachic layout algorithm can be configured to obey port constraints. Using port constraints for both ends of each edge in the graph it is possible to specify at which side of the source and target node an edge path must connect. This feature is supported for all different kinds of edges: ordinary edges, self-loops, and same layer edges.

The algorithm can be instructed to place each port at either the North, East, South, or West side of the node, or to choose the most appropriate side itself.

**Figure 5.24. Constraint on which side edges should connect to nodes**



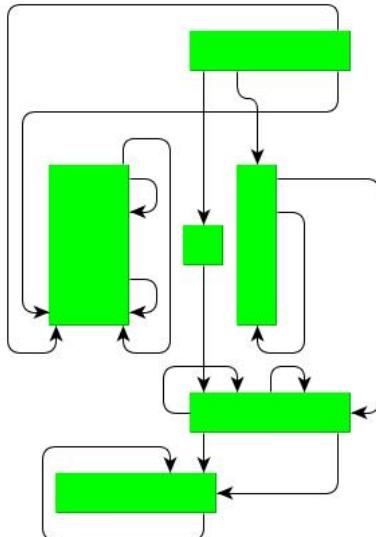
## Strong Port Constraints ([see API](#))

Opposed to "Weak Port Constraints," using "Strong Port Constraints" it is possible to not only specify the side of the node at which an edge must connect to it, but also the exact port position.

Both strong and weak port constraints can be mixed easily in the drawing. Strong port constraints are supported by all three types of edges.

[Figure 5.25, “Constraint at which exact points edges should connect to nodes” \[91\]](#) demonstrates strong port constraints (the rather "odd" port positions have been set to be "strong").

**Figure 5.25. Constraint at which exact points edges should connect to nodes**



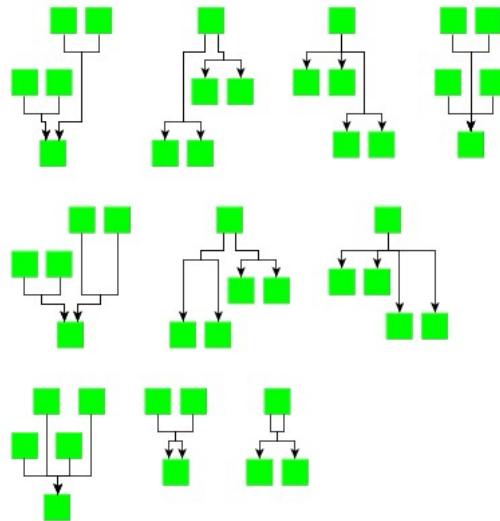
## Edge/Port Grouping (Bus-style Edge Routing) ([see API](#) and [API](#))

A special feature of the hierarchical layout algorithm is its ability to group multiple ports (edge end points) together to be anchored at the same location. This can be specified for both source ports and target ports.

Edges that belong to the same group at a specific end will additionally be routed in bus-style, i.e., if multiple edges start or end at the same layer and belong to the same group, even if they do not share the same node at that end point, they will be merged together in a bus structure in that layer.

This highly versatile configuration policy can be used to specify different interesting edge routing behaviors. Some of them are shown in [Figure 5.26, “Multiple different edge group configurations resulting in bus style edge routings” \[91\]](#)

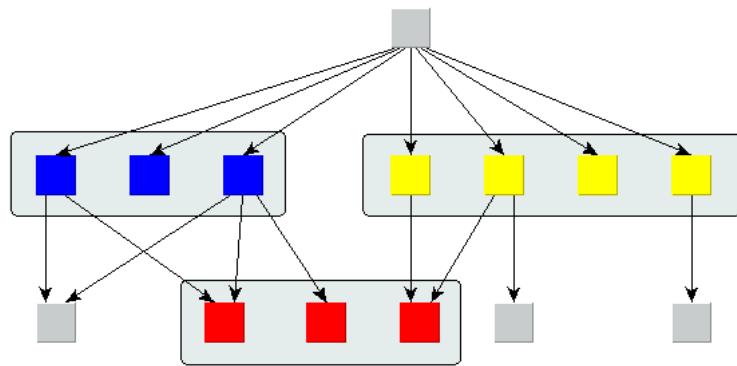
**Figure 5.26. Multiple different edge group configurations resulting in bus style edge routings**



## Packed Blocks

The hierarchical layout algorithm allows to bundle nodes within a layer to a so-called "packed block" of nodes. A packed block will always be placed as an inseparable unit on the layer. Note that this will only influence nodes that lie in the same layer. [Figure 5.27, “Packed blocks” \[92\]](#) shows three packed blocks on different layers of the graph. In the figure, the blocks are emphasized both by an enclosing rectangle and also by different node colors.

**Figure 5.27. Packed blocks**

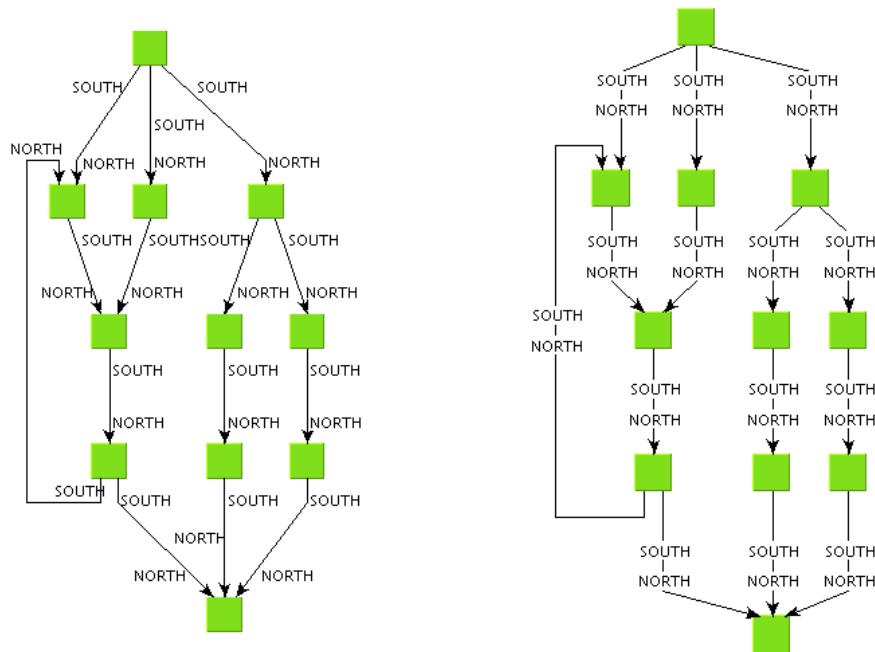


To specify the packed block a node should be put into, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key [GROUP\\_KEY](#). Note that in the absence of the data provider no packed blocks are created.

## Integrated Labeling

Besides the general labeling support as described in the section called “General Labeling” [158] which is available with all yFiles layout algorithms, the hierarchical layout algorithm additionally features integrated labeling. Integrated labeling is available for edge labels, they are taken into consideration when determining both node placement and edge path generation. With this strategy it is guaranteed that no edge label will overlap other objects in the diagram.

**Figure 5.28. Automatic edge labeling methods**



General edge labeling as a postprocessing step. Edge labeling as part of main layout procedure.  
Labels may overlap other objects. Labels only overlap their associated edges.

To specify size and preferred placement of edge labels when using integrated labeling, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key [EDGE\\_LABEL\\_LAYOUT\\_KEY](#).

Enabling integrated labeling with `HierachicLayouter` and using the services of class `LabelLayoutTranslator` to conveniently have such a data provider created and bound to the graph is described in the section called “[Integrated Labeling](#)” [160]

## Package Structure

Package `y.layout.hierachic` contains all classes that constitute hierarchical layout. The main layouter is encapsulated in the class `HierachicLayouter`. A variant of this class, that is capable of laying out hierarchically organized graphs, is `HierachicGroupLayouter`. The structure of the hierarchical layout implementation follows the Strategy design pattern. It is composed of implementations of the interface classes `Layerer`, `LayerSequencer`, and `Drawer`. Each of which is responsible for a certain phase within the layout process. Programmers can provide their specialized implementations of layout phases and can thus customize and extend layout behavior in a very powerful way.

**Figure 5.29. Structure of the hierachic layout package**



## Tutorial Demo Code

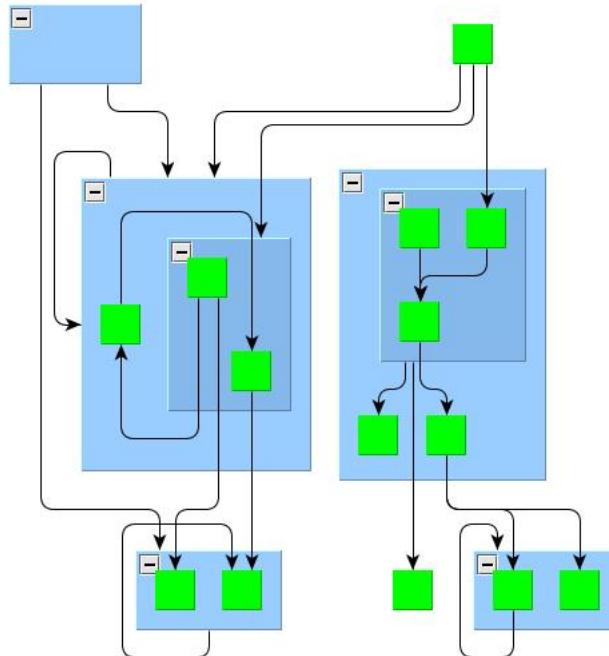
The following yFiles tutorial demo programs demonstrate how to use class `HierachicLayouter` in an application context.

- `LayoutWithoutAView.java`
- `HierachicLayoutModule.java`
- `LayoutModuleDemo.java`
- `HierachicLayouterDemo.java`
- `LayoutDemo.java`

# Hierarchical Layout of Hierarchically Organized Graphs

Class [HieranicGroupLayouter](#) is a derived variant of the classic [HieranicLayouter](#) that allows to layout hierarchically organized graphs. It recursively traverses a given graph bottom-up from the most nested groups to the top-level nodes and calculates both position and dimension of the group nodes in the hierarchical layout.

**Figure 5.30. Hierarchical layout of a hierarchically organized graph**



## Supplemental Layout Data

Class [HieranicGroupLayouter](#) knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.18, “Data provider look-up keys” \[95\]](#) lists all look-up keys for [HieranicGroupLayouter](#).

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.18. Data provider look-up keys**

Key	Element Type	Value Type	Description
<a href="#">GROUP_DPKEY</a>	Node	boolean	For each node a boolean value indicating whether it is a group node or not.
<a href="#">NODE_ID_DPKEY</a>	Node	Object	For each node an Object that serves as a unique ID.
<a href="#">PARENT_NODE_ID_DPKY</a>	Node	Object	For each node an Object indicating the group node it belongs to. The Object matches the unique ID of a group node that is in the same graph.
<a href="#">GROUP_NODE_INSETS_DPKEY</a>	Node	Insets	For each group node a <code>java.awt.Insets</code> object that encodes the group node's insets.

Setup of a graph's hierarchical organization and using the grouping keys ([GROUP\\_DPKEY](#),

NODE\_ID\_DPKEY, and PARENT\_NODE\_ID\_DPKEY) is described in detail in the section called “Setup for Layout” [57].

## Layout Options

HierachicGroupLayouter is an extension of the [hierarchical layout algorithm](#). Beyond the options of this layouter, further support for the specific needs of hierarchically organized graphs is provided.

Additional size requirements of group nodes (e.g., special insets or space for custom labels) can be specified in one of two different ways. Either a custom implementation of interface [GroupBoundsCalculator](#) can be set using the method from [API Excerpt 5.8, “Customizing insets calculation for group nodes”](#) [96] or the default implementation for that interface, class [InsetsGroupBoundsCalculator](#), can be used.

### API Excerpt 5.8. Customizing insets calculation for group nodes

```
// Method from class HierachicGroupLayouter.  
void setGroupBoundsCalculator(GroupBoundsCalculator gbc)
```

Choosing the latter alternative additionally requires the use of a data provider that provides `java.awt.Insets` objects for each group node. The data provider must be registered with the input graph using the key [GROUP\\_NODE\\_INSETS\\_DPKEY](#). During layout calculation it is then used by class [InsetsGroupBoundsCalculator](#).

## Tutorial Demo Code

The following yFiles tutorial demo programs demonstrate how to use class HierachicGroupLayouter in an application context.

- [GroupingLayoutWithoutAView.java](#)
- [HierachicLayoutModule.java](#)

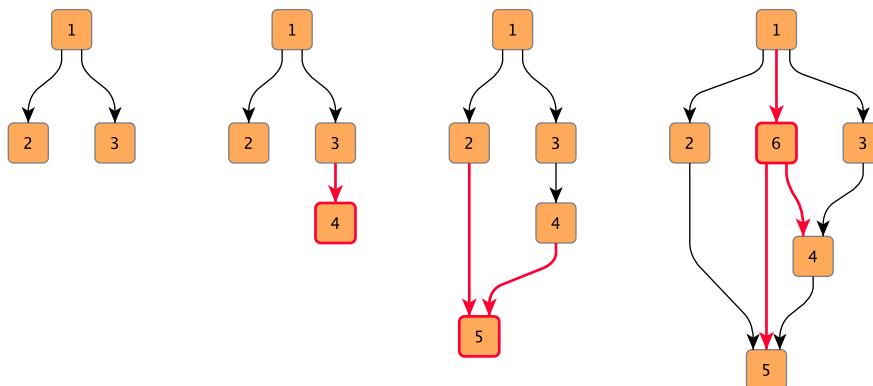
## Incremental Hierarchical Layout

Class `IncrementalHierarchicLayouter` is a hierarchical layout provider that supports both incremental graph layout as well as "normal" hierarchical layout. It is a variant of the [classic hierarchical layout](#) providing a number of options similar to those of class `HierarchicLayouter`. Furthermore, this layout provider also supports incremental and "normal" hierarchical layout of hierarchically organized graphs.

Note that the non-incremental hierarchical layout which is provided by class `IncrementalHierarchicLayouter` is referred to as "layout from scratch."

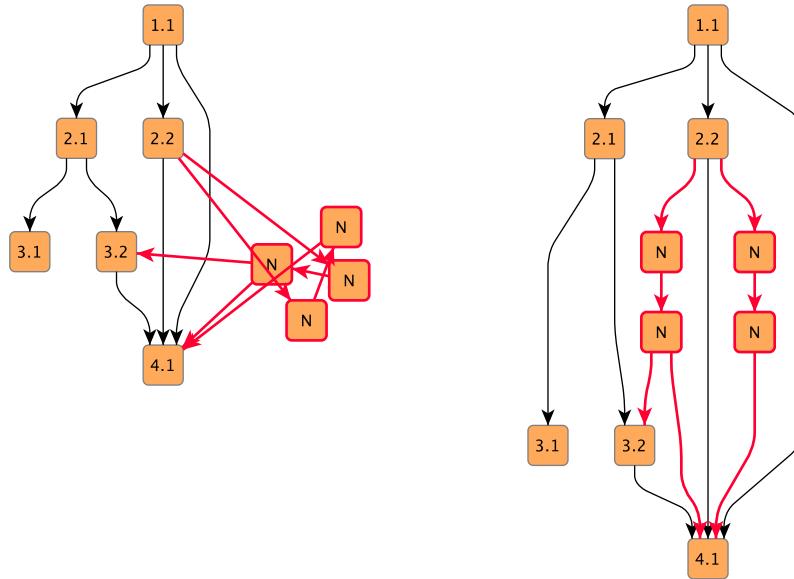
[Figure 5.31, "Sequence of incremental layouts"](#) [97] shows a sequence of incremental layouts generated by class `IncrementalHierarchicalLayouter`. Starting with a given graph, new graph elements are inserted optimally into the existing drawing from the step before. Note the emphasis for newly added elements.

**Figure 5.31. Sequence of incremental layouts**



The second major use case for incremental layout, the optimization of distinct parts from an existing hierarchical layout is shown in [Figure 5.32, "Incremental layout used for optimization"](#) [?]. There, an entire subgraph is calculated anew and optimally placed into the given drawing.

**Figure 5.32. Incremental layout used for optimization**



## Specifying Hints

Calculation of incremental hierarchical layouts heavily relies on the services of a so-called "hint factory." A hint factory is responsible for creating hint objects for both nodes and edges. These objects are then used by the incremental layout algorithm to optimally:

- insert nodes into specified layers of the existing drawing
- place nodes into suitable layers of the existing drawing
- place nodes into suitable layers of the existing drawing with respect to their current coordinates (either both directions, or confined to only one direction)
- route edges

When inserting nodes or routing edges according to their hint, nodes and edges from the graph that have no hint object associated preserve their original relative order both within layers as well as from layer to layer.

Class `IncrementalHierarchicLayouter` has a getter method that returns a hint factory object of type `IncrementalHintsFactory`. Code that shows the usage of a hint factory is presented in [Example 5.20, “Getting and using a hint factory”](#) [98]

### Example 5.20. Getting and using a hint factory

```
// 'graph' is of type y.layout.LayoutGraph.

// Create the incremental layout.
IncrementalHierarchicLayouter ihl = new IncrementalHierarchicLayouter();

// Create a map to store the hints for the incremental layout mechanism.
DataMap hintMap = Maps.createHashedDataMap();
graph.addDataProvider(IncrementalHierarchicLayouter.INCREMENTAL_HINTS_DPKY,
                     hintMap);

// Get the hint factory from the incremental layout algorithm.
IncrementalHintsFactory hintsFactory = ihl.createIncrementalHintsFactory();

// Get a NodeList with those nodes that should be processed using incremental
// layout semantics.
NodeList incNL = getIncrementalNodeList();

// Associate the incremental nodes with hints from the hint factory.
for (NodeCursor nc = incNL.nodes(); nc.ok(); nc.next()) {
    hintMap.set(nc.node(), hintsFactory.createLayerIncrementallyHint(nc.node()));
}
```

```
// Now, set incremental mode and invoke layout calculation.
ihl.setLayoutMode(IncrementalHierarchicLayouter.LAYOUT_MODE_INCREMENTAL);
new BufferedLayouter(ihl).calcLayout(graph);
```

## Supplemental Layout Data

In addition to the data provider keys known by class HierarchicLayouter (listed in [Table 5.17, “Data provider look-up keys” \[83\]](#)), class IncrementalHierarchicLayouter knows further keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.19, “Data provider look-up keys” \[99\]](#) lists all look-up keys for IncrementalHierarchicLayouter.

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.19. Data provider look-up keys**

Key	Element Type	Value Type	Description
<a href="#">INCREMENT-AL_HINTS_DPKEY</a>	Node, Edge	Object	For each incrementally added node or edge a hint object that marks the respective graph element to be inserted into the hierarchical layout in an optimal manner. The hint object is created by a hint factory, like, e.g., the hint factory that is returned by method <code>createIncrementalHintsFactory()</code> .
<a href="#">LAYER_VALUE_HOLDER_DPKEY</a>	Node	Int-Value-Holder	For each node an <a href="#">IntValueHolder</a> implementation that is used by the layout algorithm to return the index of the actual layer a node has been assigned to.
<a href="#">NODE_LAYOUT_DESCRIPTOR_DPKEY</a>	Node	NodeLayoutDescriptor	For each node a <a href="#">NodeLayoutDescriptor</a> object that configures a number of node-related options.
<a href="#">EDGE_LAYOUT_DESCRIPTOR_DPKEY</a>	Edge	EdgeLayoutDescriptor	For each edge an <a href="#">EdgeLayoutDescriptor</a> object that configures a number of edge-related options.
<a href="#">SWIM-LANE_DESCRIPTOR_DPKEY</a>	Node	SwimLaneDescriptor	For each node a <a href="#">SwimLaneDescriptor</a> object that configures a number of swimlane-related options.
<a href="#">SOURCE_GROUPID_KEY</a>	Edge	Object	For each edge an arbitrary Object indicating the group its source end is affiliated with.
<a href="#">TARGET_GROUPID_KEY</a>	Edge	Object	For each edge an arbitrary Object indicating the group its target end is affiliated with.
<a href="#">SOURCE_PORT_CONSTRAINT_KEY</a>	Edge	PortConstraint	For each edge a PortConstraint object encoding its source end's port constraint.
<a href="#">TARGET_PORT_CONSTRAINT_KEY</a>	Edge	PortConstraint	For each edge a PortConstraint object encoding its target end's port constraint.
<a href="#">NODE_DP_KEY</a>	Node	PortCandidateSet	For each node a PortCandidateSet object encoding the set of allowed anchor locations for edges.
<a href="#">SOURCE_PCLIST_DPKEY</a>	Edge	Collection	For each edge a <code>java.util.Collection</code> of PortCandidate objects that encode the subset of desired anchor locations where the source port likes to connect to.

## Layout Options

Key	Element Type	Value Type	Description
TAR-GET_PCLIST_DPKEY	Edge	Collection	For each edge a <code>java.util.Collection</code> of <code>PortCandidate</code> objects that encode the subset of desired anchor locations where the target port likes to connect to.
GROUP_DPKEY	Node	boolean	For each node a boolean value indicating whether it is a group node or not.
NODE_ID_DPKEY	Node	Object	For each node an Object that serves as a unique ID.
PARENT_NODE_ID_DPK	Node	Object	For each node an Object indicating the group node it belongs to. The Object matches the unique ID of a group node that is in the same graph.

## Important

The data provider that is registered using the look-up key INCREMENTAL\_HINTS\_DPKEY holds data for both types of graph elements. Hence, neither `NoodeMap` nor `EdgeMap` implementations can be used as the basis for this data provider.

An alternative basis for the data provider would be, e.g., a `DataMap` as returned by method `createHashedDataMap()`, or any custom `DataProvider` implementation that is not restricted to one type of graph element.

Setup of a graph's hierarchical organization and using the grouping keys (GROUP\_DPKEY, NODE\_ID\_DPKEY, and PARENT\_NODE\_ID\_DPK) is described in detail in the section called "Setup for Layout" [57].

## Layout Options

Unlike the classic hierarchical layout algorithm this specialized layout algorithm offers only few options that are directly accessible for configuration.

### Layout Mode (see API)

Determines the general layout mode.

When the layout is calculated incrementally, a data provider that holds so-called hint objects for each graph element that should be processed using incremental semantics is looked up. The data provider is expected to be registered with the graph using key INCREMENTAL\_HINTS\_DPKEY.

- |               |   |
|---------------|---|
| Incrementally | Sets the layout algorithm to incremental mode, i.e., elements that are marked for incremental processing will be inserted into the already calculated layout of the remaining part of the graph in an optimal manner.   |
| From Scratch  | Sets the layout algorithm to recompute the entire layout from scratch. This is similar to the services provided by class <code>HierachicLayouter</code> , i.e., a "normal" layout calculation is started. Layout from scratch is the default setting for the "Layout Mode" feature. |

### Node Layout Descriptor (see API)

Sets the default `NodeLayoutDescriptor` instance that is used for all nodes that have no individual layout descriptor associated. Some of the layout and drawing options that can be set for a node by means of this descriptor are listed in the section called "Related Classes" [103]

To specify individual descriptors for nodes, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key NODE\_LAYOUT\_DESCRIPTOR\_DPKEY. Note that in the absence of an individual descriptor the default `NodeLayoutDescriptor` instance will be used.

### Edge Layout Descriptor (see API)

Sets the default `EdgeLayoutDescriptor` instance that is used for all edges that have no individual layout descriptor associated. Some of the layout and drawing options that can be set for an edge

by means of this descriptor are listed in [the section called “Related Classes” \[103\]](#)

To specify individual descriptors for edges, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key `EDGE_LAYOUT_DESCRIPTOR_DPKEY`. Note that in the absence of an individual descriptor the default EdgeLayoutDescriptor instance will be used.

## Drawing Style Options

### Minimum Layer Distance ([see API](#))

Determines the minimal distance between adjacent layers.

### Node To Node Distance ([see API](#))

Determines the minimal distance between two adjacent nodes that reside in the same layer.

### Edge To Edge Distance ([see API](#))

Determines the distance between two adjacent edge segments in one layer.

### Node To Edge Distance ([see API](#))

Determines the minimal distance between an edge and an adjacent node in one layer.

## Advanced Layout Features

Class IncrementalHierarchicLayouter basically supports the same or very similar functionality (including knowledge and usage of corresponding data provider keys) as the [classic hierarchical group layout algorithm](#), including:

- port constraints
- edge/port grouping (bus-style edge routing)
- hierarchically organized graphs

All of these advanced features are observed as soon as there are data providers registered with a graph using appropriate look-up keys.

In addition to the support provided for port constraints, incremental hierarchical layout also supports the concept of port candidates. See [the section called “Port Candidates” \[61\]](#) for a detailed description of that concept.

## Layout of Hierarchically Organized Graphs

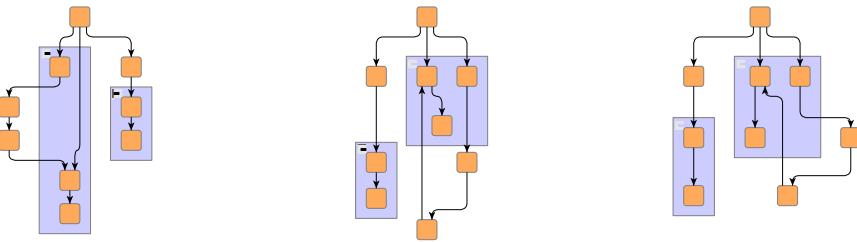
IncrementalHierarchicLayouter supports different layer assignment policies for graphs with grouped nodes. The layering for both incremental as well as "normal" layout can be determined in either of two ways:

- flat, i.e., nodes are assigned to layers regardless of nesting level within group nodes
- recursively, i.e., layer assignment is computed from the most nested group up to the nodes in the root graph

**Figure 5.33, “Flat vs. recursive layer assignment” [101]** compares the layer assignment policies.

When layer assignment is done "flat," the group nodes of a graph and their adjacent edges are ignored. In particular, this means that the layering of grouped nodes can be influenced by nodes outside of the group node. In contrast, when using recursive layer assignment, grouped nodes are processed without interference from nodes outside of their group node.

### Figure 5.33. Flat vs. recursive layer assignment



"Flat" layer assignment policy where group nodes and their adjacent edges are ignored.

Recursive layer assignment policy.

Recursive layer assignment policy with compaction enabled.

Recursive processing of the grouped nodes is the default behavior. [Example 5.21, “Setting up “flat” layer assignment” \[102\]](#) shows how to set up "flat" layer assignment with IncrementalHierarchicLayouter.

### Example 5.21. Setting up "flat" layer assignment

```
// Set up IHL for hierarchically organized graphs.
IncrementalHierarchicLayouter ihl = new IncrementalHierarchicLayouter();
ihl.setRecursiveGroupLayeringEnabled(false);
```

Recursive layer assignment optionally uses a compaction step where empty layers next to group nodes are filled with nodes from layers below these group nodes. When compaction is disabled, an alignment policy is used to specify where "ordinary" nodes that are in a layer with group nodes are placed relative to these group nodes. [API Excerpt 5.9, “Setter methods for recursive layering options” \[102\]](#) lists the methods for configuring these options.

### API Excerpt 5.9. Setter methods for recursive layering options

```
// Layer compaction.
void setGroupCompactionEnabled(boolean groupCompactionEnabled)

// Group node-relative alignment policy.
void setGroupAlignmentPolicy(byte groupAlignmentPolicy)
```

IncrementalHierarchicLayouter's support for hierarchically organized graphs currently does not support the following features for edge ends that directly connect to group nodes: port constraints, port candidates, and edge/port grouping (bus-style edge routing). Also, swimlane layout in conjunction with hierarchically organized graphs is not supported yet.

Please see also the section called “[Layout of Hierarchically Organized Graphs](#)” [103] below.

## Integrated Labeling

Integrated labeling can be enabled or disabled using the methods from [API Excerpt 5.10, “Method to enable/disable integrated labeling” \[102\]](#)

### API Excerpt 5.10. Method to enable/disable integrated labeling

```
// Getter and setter for integrated labeling.
boolean isIntegratedEdgeLabelingEnabled()

void setIntegratedEdgeLabelingEnabled(boolean enabled)
```

## Returning Layer Indices

Class IncrementalHierarchicLayouter will return the layer index for every node, when the graph has a data provider registered with it that can be looked up using the key **LAY-**

**ER\_VALUE HOLDER\_DPKEY**. The data provider is expected to hold IntValueHolder objects, which are retrieved and filled with the layer indices after layout calculation. Class **IntValueHolderAdapter** can be used as a convenient means to appropriately wrap a NodeMap for that task.

## Related Classes

Classes **NodeLayoutDescriptor** and **EdgeLayoutDescriptor** can be used to configure the layout result of an invocation of class **IncrementalHierarchicLayouter**. If set individually for single graph elements by means of a data provider, the layout result can be customized in the most detailed way. For example, the following options can be set for nodes and edges, respectively:

- relative alignment of nodes within their layer
- preferred minimum distance from obstacles (both nodes and edges)
- orthogonal vs. poly-line routing for edges
- minimum length of first and last edge segment, respectively

[API Excerpt 5.11, “Setter methods from class NodeLayoutDescriptor” \[103\]](#) lists some of the methods from class **NodeLayoutDescriptor** that can be used for node configuration.

### API Excerpt 5.11. Setter methods from class NodeLayoutDescriptor

```
// Setter methods from NodeLayoutDescriptor.  
void setLayerAlignment(double alignment)  
  
void setMinimumDistance(double distance)  
void setMinimumLayerHeight(double height)  
  
void setNodeLabelMode(byte mode)
```

[API Excerpt 5.12, “Setter methods from class EdgeLayoutDescriptor” \[103\]](#) lists some of the setter methods from class **EdgeLayoutDescriptor** that can be used for edge configuration.

### API Excerpt 5.12. Setter methods from class EdgeLayoutDescriptor

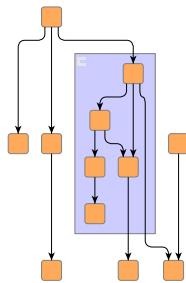
```
// Setter methods from EdgeLayoutDescriptor.  
void setMinimumDistance(double distance)  
  
void setMinimumFirstSegmentLength(double length)  
void setMinimumLastSegmentLength(double length)  
void setMinimumLength(double length)  
void setMinimumSlope(double slope)  
  
void setOrthogonallyRouted(boolean orthogonal)
```

## Layout of Hierarchically Organized Graphs

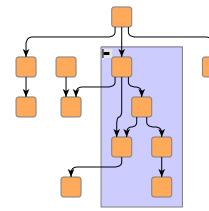
Due to the nature of class **IncrementalHierarchicLayouter**'s layout capabilities, which cover both non-incremental as well as incremental layout, the layout of a graph with grouped nodes will most likely differ from what class **HierarchicGroupLayouter**, which supports non-incremental hierarchical layout of hierarchically organized graphs only, yields.

[Figure 5.34, “Hierarchical layout of grouped nodes: HierarchicGroupLayouter vs. IncrementalHierarchicLayouter” \[103\]](#) compares the hierarchical layout of a graph with grouped nodes as calculated by class **HierarchicGroupLayouter** with the outcome of a from-scratch layout by class **IncrementalHierarchicLayouter** for the same graph (using “flat” layer assignment policy). The most notable difference can be seen with the group nodes which are always placed into a single distinct layer by **HierarchicGroupLayouter** whereas they can span multiple layers when placed by **IncrementalHierarchicLayouter**.

**Figure 5.34. Hierarchical layout of grouped nodes: HierarchicGroupLayouter vs. IncrementalHierarchicLayouter**



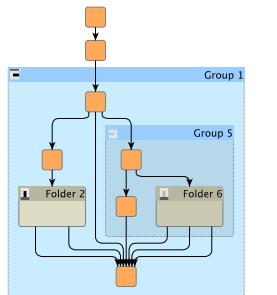
Hierarchical layout of grouped nodes calculated by class `HierarchicGroupLayouter`.



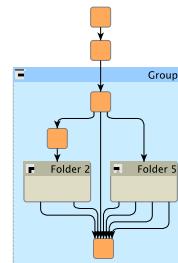
Hierarchical "from-scratch" layout of grouped nodes calculated by class `IncrementalHierarchicLayouter`.

`IncrementalHierarchicLayouter`'s support for incrementally calculating layouts of hierarchically organized graphs enables smooth transitions when realizing collapsing and expanding of group nodes. [Figure 5.35, “Incremental hierarchical layout when group nodes are collapsed and expanded” \[104\]](#) presents the results of both these operations. The resulting folder node and group node, respectively, is incrementally inserted into the existing layout.

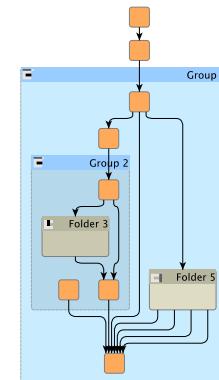
**Figure 5.35. Incremental hierarchical layout when group nodes are collapsed and expanded**



Original hierarchical layout with group nodes.



Collapsed group node incrementally inserted into the layout.



Previously collapsed group node expanded and incrementally inserted.

Incremental hierarchical layout of graphs with grouped nodes is demonstrated in the tutorial demo application [IncrementalHierarchicGroupDemo.java](#).

## Tutorial Demo Code

Using both the incremental as well as the classic layout functionality of class `IncrementalHierarchicLayouter` is presented in detail in the following tutorial demo applications:

- [SimpleIncrementalHierarchicLayouterDemo.java](#)
- [IncrementalHierarchicLayouterDemo.java](#)
- [SimpleSwimLaneLayouterDemo.java](#)
- [IncrementalLayoutWithoutAView.java](#)
- [SwimLaneLayoutWithoutAView.java](#)
- [IncrementalHierarchicGroupDemo.java](#)

Configuration of the incremental hierarchical layout algorithm can also be observed in [IncrementalHierarchicalLayoutModule.java](#).

## Organic Layout

Class [OrganicLayouter](#) is a multi-purpose layouter for undirected graphs. It produces clear representations of complex networks and is especially fit for application areas such as

- Bioinformatics
- Enterprise networking
- Knowledge representation
- System management
- WWW visualization
- Mesh visualization

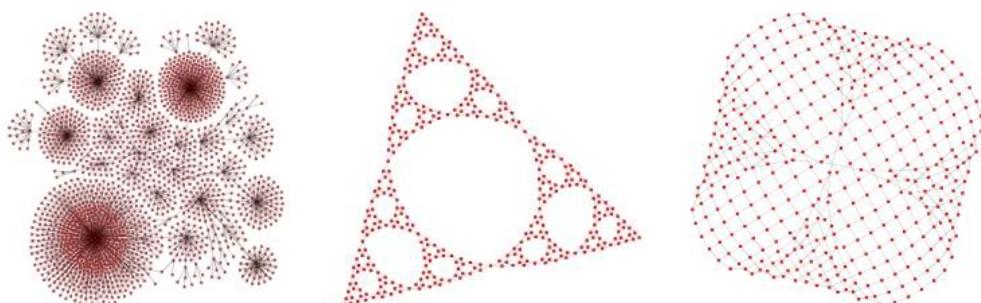
OrganicLayouter is based on the force-directed layout paradigm. When calculating a layout, the nodes are considered to be physical objects with mutually repulsive forces, like, e.g., protons or electrons. The connections between nodes also follow the physical analogy and are considered to be metal springs attached to the pair of nodes. These springs produce repulsive or attractive forces between their end points if they are too short or too long. The layouter simulates these physical forces and rearranges the positions of the nodes in such a way that the sum of the forces emitted by the nodes and the edges reaches a (local) minimum.

Resulting layouts often expose the inherent symmetric and clustered structure of a graph, a well-balanced distribution of nodes and few edge crossings.

The layouter is well suited for the visualization of highly connected backbone regions with attached peripheral ring or star structures. These structurally different regions of a network can be easily identified by looking at a drawing produced by this layouter.

See the [Smart Organic Layouter](#) for another layouter which makes use of this layouter and related layouting techniques.

**Figure 5.36. Sample layouts produced by class OrganicLayouter**



The network represents mount point relationships in a shared file system.

The inherent symmetry of this recursively defined graph gets exposed nicely.

The inherent symmetry of this extended short-cut grid-graph gets exposed nicely.

## Supplemental Layout Data

Class [OrganicLayouter](#) knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.20, “Data provider look-up keys”](#) [?] lists all look-up keys for [OrganicLayouter](#).

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data”](#) [55].

**Table 5.20. Data provider look-up keys**

Key	Ele- ment Type	Value Type	Description
GROUP_DPKEY	Node	boolean	For each node a boolean value indicating whether it is a group node or not.
NODE_ID_DPKEY	Node	Object	For each node an Object that serves as a unique ID.
PARENT_NODE_ID_DPK EY	Node	Object	For each node an Object indicating the group node it belongs to. The Object matches the unique ID of a group node that is in the same graph.
SPHERE_OF_ACTIO N_NODES	Node	boolean	For each node a boolean value indicating whether it should be treated as selected or not.
PRE- FERRED_EDGE_Le NGTH_DATA	Edge	int	For each edge an integral value that indicates its preferred length.
FIXED_GROUP_NO DES_DPKEY	Node	boolean	For each group node a boolean value indicating whether its content should remain fixed or not.

Setup of a graph's hierarchical organization and using the grouping keys (GROUP\_DPKEY, NODE\_ID\_DPKEY, and PARENT\_NODE\_ID\_DPK\_EY) is described in detail in the section called “Setup for Layout” [57].

## Layout Options

Organic layout provides a set of options that influence its behavior. These options can be set using the setter methods of class OrganicLayouter.

### Sphere of Action (see API)

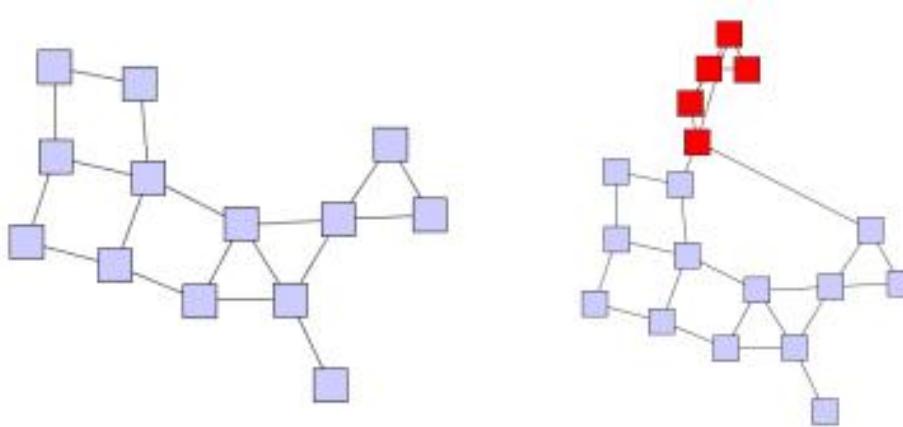
Determines the set of nodes from the graph that should be processed.

When only a subset should be rearranged, a data provider holding the selection state for each node is looked up. The data provider is expected to be registered with the graph using key SPHERE\_OF\_ACTION\_NODES.

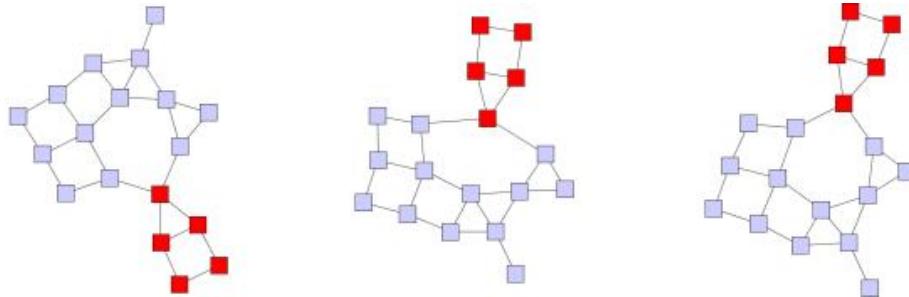
All	The whole graph will be considered for layout.
Mainly Selection	The selected nodes will be rearranged. The unselected nodes are only allowed to move to a certain degree.
Only Selection	Only the selected nodes will be rearranged.

Figure 5.37, “Initial graph and an augmentation” [107] and Figure 5.38, “Incremental layout of augmented graph” [108] show how the “Sphere of Action” option enables different kinds of **incremental layout** strategies. To perform incremental layout, some nodes have to be marked as “selected.” Only (or mostly) the selected nodes will be arranged by OrganicLayouter. All other nodes will stay at their original position.

**Figure 5.37. Initial graph and an augmentation**



**Figure 5.38. Incremental layout of augmented graph**



Setting "Sphere of Action" to **ALL** results in a complete re-layout of the augmented graph. No incremental feature here.

Setting "Sphere of Action" to **ONLY\_SELECTION**. This places only selected (here: red) nodes and leaves other nodes untouched. Simple incremental feature.

Setting "Sphere of Action" to **MAINLY\_SELECTION**. This places selected (here: red) nodes and moves other nodes only a little bit. Smart incremental feature.

### Initial Placement (see API)

Determines the initial placement of the nodes from where the layuter starts to find a good resulting layout.

Random	The nodes are randomly placed. This is a good choice, if nothing is known about the input graph or if completely different layout results are desired for successive calls to the layouter.
At (0, 0)	All nodes will be placed at the same location (0, 0).
As Is	The layouter starts with the given coordinates. This is a good choice, if the given layout is similar to the desired layout.

### Preferred Edge Length (see API)

Allows to specify the general preferred length for all edges. The layouter tries to arrange the nodes in such a way that the edges have the desired edge length. If "Obey Node Size" is checked, then the edge length is measured from node border to node border. Otherwise, the edge length is measured from node center to node center.

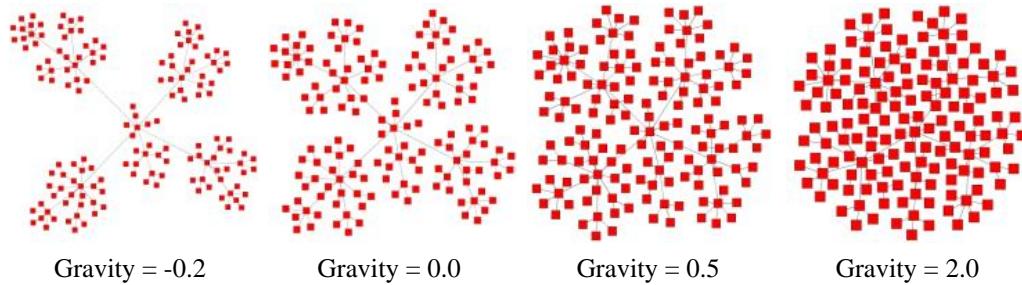
To specify the preferred edge length for each edge individually, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key **PREFERRED\_EDGE\_LENGTH\_DATA**. Note that in the absence of an individual preferred edge length the general preferred edge length will be used.

### Gravity Factor (see API)

Regulates the tendency of the nodes to be placed near the barycenter of the resulting layout. The greater the factor, the closer are the nodes to the barycenter. Negative values lead to huge layouts.

[Figure 5.39, "Effect of the gravity factor"](#) [109] shows the effect of different gravity settings. The magnitude of the gravity factor determines the strength of the force towards the barycenter of the graph. A high gravity factor tends to cluster nodes around the barycenter of the graph. A low factor stretches the outskirts of the graph far away from the center. By default a value of 0.0 is assumed.

**Figure 5.39. Effect of the gravity factor**



### [Obey Node Size \(see API\)](#)

Whether or not to obey the size of the nodes when calculating the layout. The distance between two nodes is calculated with respect to the node sizes, i.e., for equal preferred edge lengths, activating this feature will lead to bigger layouts.

### [Attraction \(see API\)](#)

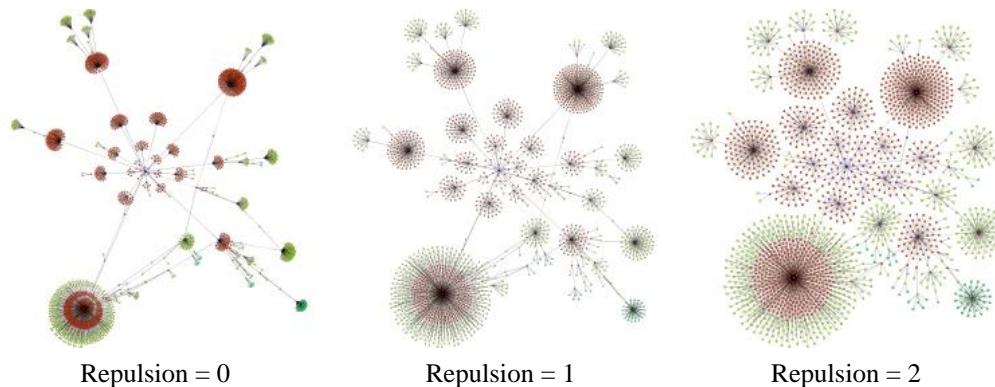
This setting determines the strength of the force that edges oppose on their adjacent nodes. Higher values result in edges which better obey the given preferred edge length.

### [Repulsion \(see API\)](#)

This setting determines the strength of the forces that nodes oppose on their topological neighbors. Higher values result in greater node distances. In this case, preferred edge lengths are more likely to be longer than the preferred edge length.

[Figure 5.40, "A graph laid out using different repulsion factors"](#) [109] shows how the "Repulsion" option affects the layout. The value determines repulsion through the inverse power of the distance between two nodes, i.e., smaller values lead to slowly degrading repulsion, whereas greater values simulate local repulsion only. Notice the increasing localness of mutual node repulsion for increasing factors.

**Figure 5.40. A graph laid out using different repulsion factors**



### [Activate Tree Beautifier \(see API\)](#)

Activates an experimental feature that lays out tree-like sub-structures of the graph in an optimized way.

### Iteration Factor ([see API](#))

Influences the quality and execution time of this layouter. Greater values may result in better quality and most likely longer execution time.

### Maximum Duration ([see API](#))

Sets the maximal duration of the layout process in milliseconds. If this upper bound is hit during the layout process, the quality of the may not be optimal. Increasing this values increases the likeliness of an optimal layout.

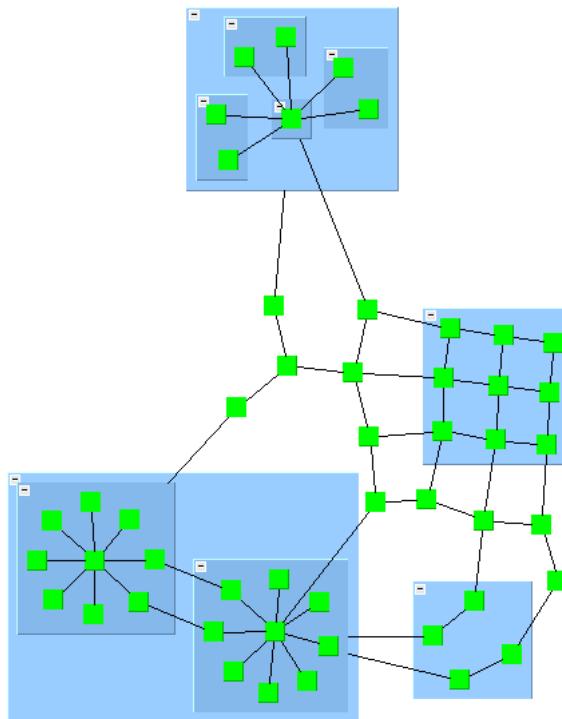
### Activate Deterministic Mode ([see API](#))

Whether or not the layout process should be deterministic. In deterministic mode the layouter produces identical results for identical input graphs.

## Advanced Layout Features

The organic layout algorithm implemented in yFiles allows to layout hierarchically organized graphs. Both position and dimension of group nodes will be calculated by the algorithm. Additional size requirements (insets or custom labels) can be provided for each group node separately (see [API](#) and [API](#)).

**Figure 5.41. Organic layout of a hierarchically organized graph**



### Group Node Policy ([see API](#))

Determines the basic policy for the layout process.

To keep the contents of only certain group nodes fixed, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key [FIXED\\_GROUP\\_NODES\\_DPKEY](#).

Layout Groups

Instructs the algorithm to layout group nodes as well as their contents in a global manner. The positions of all visible

	nodes will be recalculated.
Fix Contents of Groups	The algorithm will keep the relative position of nodes inside of group nodes and performs the layout on the top level only, taking the position of nodes inside the group nodes into account.
Ignore Groups	This setting will make the algorithm ignore group nodes entirely. Nodes will be rearranged as if there were no group nodes. This may lead to overlapping group nodes.

### Group Node Compactness (see API)

This setting can be used to control the compactness of group nodes. Larger values will lead to more compact group nodes but inter-edges may be longer and nodes inside group nodes tend to get clutched together at the center of the group node. Smaller values will result in larger group nodes in favor of shorter inter-edge lengths and more evenly spread nodes inside group nodes.

### Incremental Layout

OrganicLayouter supports incremental layout through the "Sphere of Action" feature. See the [above description](#).

## Tutorial Demo Code

Layout module [OrganicLayoutModule.java](#) presents the setup of class OrganicLayouter in an application context.

## Smart Organic Layout

Smart organic layout is an improved variant of class [OrganicLayouter](#). It combines and configures different techniques and layout algorithms depending on the user's demands. Its strength is its ability to control the ratio of quality versus running time, as well as its ability to produce organic layouts while guaranteeing the absence of node overlaps.

### Supplemental Layout Data

Class [SmartOrganicLayouter](#) knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.20, “Data provider look-up keys” \[106\]](#) lists all look-up keys for [SmartOrganicLayouter](#).

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.21. Data provider look-up keys**

Key	Element Type	Value Type	Description
<a href="#">NODE_SUBSET_DATA</a>	Node	boolean	For each node a boolean value indicating whether it should be treated as selected or not.
<a href="#">PREFERRED_EDGE_LENGTH_DATA</a>	Edge	int	For each edge an integral value that indicates its preferred length.

## Layout Options

Class [SmartOrganicLayouter](#) provides a set of options that influence the layout behavior. Some of these options are very closely related to those from class [OrganicLayouter](#), so the [documentation of organic algorithm](#) may be useful, too.

### Scope ([see API](#))

Determines the set of nodes from the graph that should be processed.

When only a subset should be rearranged, a data provider holding the selection state for each node is looked up. The data provider is expected to be registered with the graph using key [NODE\\_SUBSET\\_DATA](#).

- |               |   |
|---------------|---|
| All           | The whole graph will be considered for layout.  |
| Mainly Subset | The selected nodes will be rearranged. The unselected nodes are only allowed to move to a certain degree. |
| Subset        | Only the selected nodes will be rearranged.   |

### Scope in Unconnected Graphs ([see API](#))

When enabled, the [ComponentLayouter](#) that is used with this [CanonicMultiStageLayouter](#) is configured in such a way that separate components containing nodes which should not be moved will not be rearranged.

### Preferred Edge Length ([see API](#))

Allows to specify the general preferred length of all edges. The layouter tries to arrange the nodes in such a way that the edges have the desired edge length. If "Node Size Aware" is checked, then the edge length is measured from node border to node border. Otherwise, the edge length is measured from node center to node center.

To specify the preferred edge length for each edge individually, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key [PREFERRED\\_EDGE\\_LENGTH\\_DATA](#). Note that in the absence of an individual preferred edge length the general preferred edge length will be used.

### **Node Size Aware (see API)**

Whether or not to obey the size of the nodes when calculating the layout. The distance between two nodes is calculated with respect to the node sizes, i.e., for equal preferred edge lengths, activating this feature will lead to bigger layouts.

### **Node Overlaps Allowed (see API)**

Whether or not nodes are allowed to overlap. If this feature is disabled, the "Minimal Node Distance" will be used to arrange the nodes in such a way that the given distance will be obeyed.

### **Minimal Node Distance (see API)**

The minimal node distance which will be used if "Node Overlaps Allowed" is disabled.

### **Compactness (see API)**

Adjusting this value can lead to a variety of differing layouts. For small values, the resulting layout will use a lot of space and nodes tend to be far away from each other. Values around 0.5 lead to evenly distributed nodes, whereas values near 1.0 produce highly compact layouts.

### **Quality/Time Ratio (see API)**

This setting can be used to adjust the quality versus the running time of the layout algorithm. Small values lead to short running times, while greater values result in better quality. For large graph structures (hundreds and thousands of nodes) it is advisable to begin with smaller values and to gradually increase them.

### **Maximum Duration (see API)**

Sets the maximal duration of the layout process in milliseconds. If this upper bound is hit during the layout process, the quality of the layout may not be optimal. Increasing this values increases the likeliness of an optimal layout.

### **Deterministic (see API)**

Whether or not the layout process should be deterministic. In deterministic mode the layouter produces identical results for identical input graphs and identical settings.

## **Output Restrictions**

Class SmartOrganicLayouter allows to impose so-called "output restrictions" on the result of the layout calculation. More precisely, the layout result can be determined to fit into a specified region, e.g., a rectangle or a circle. Also, the layout result can be restricted to a specified aspect ratio, too.

Class [OutputRestriction](#) serves as a factory to create predefined output restrictions that can be used in conjunction with setter method [setOutputRestriction](#).

## **Advanced Layout Features**

### **Incremental Layout**

SmartOrganicLayouter supports incremental layout through the "Scope" feature. See the [above description](#).

### **Quality Versus Time**

Smart organic layout can choose between different strategies, combinations and configurations of a couple of related algorithms to control the running time versus the quality of the layout.

Using small values for property "Quality/Time Ratio," it is possible to produce acceptable layouts for huge graphs (thousands of nodes within seconds), whereas setting this value to greater values leads to high-quality layouts, which may take longer (up to a few minutes for hundreds of nodes).

### **Compactness**

This algorithm can be configured to produce significantly more or less compact layouts.

Small values for property "Compactness" lead to large area drawings with a relatively sparse

node distribution. Values greater than 0.5 lead to artificially compacted layouts, and values next to 1.0 will produce layouts with near to minimal space allocations. Note that "Compactness" will be obeyed only for medium or high quality settings.

## Node Overlap Removal

Using property "Node Overlaps Allowed," smart organic layout can be configured to guarantee the absence of node overlaps even for very dense graphs.

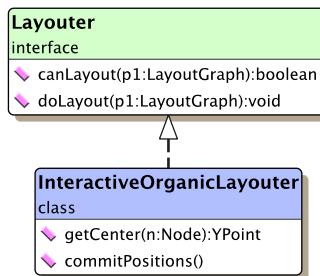
Additionally, using property "Minimal Node Distance," it is possible to specify a minimal node distance, i.e., it can be guaranteed that there will be a certain amount of space around each node, which is not occupied by other nodes.

# Interactive Organic Layout

Class `InteractiveOrganicLayouter` provides [organic layout](#) for use in interactive environments. Its strength is its capability to generate continuous updates to the layout of a graph during calculation. Furthermore, it also allows a user to seemingly simultaneously perform arbitrary modifications to a graph, which are subsequently accounted for in the layout calculation.

In contrast to the majority of yFiles layout algorithms, class `InteractiveOrganicLayouter` does not inherit from `CanonicMultiStageLayouter`, but is instead provided as an implementation of interface `Layouter` as depicted in [Figure 5.42, “Interactive organic layout implementation”](#) [115]

**Figure 5.42. Interactive organic layout implementation**



Among other things, this means that there is no special support for routing of either self-loops or parallel edges, for example. In particular, when a graph contains parallel edges, they will be routed using overlapping edge paths.

## General Usage

`InteractiveOrganicLayouter` is designed to be used in a concurrent setup where the layout algorithm runs in a thread of its own. This enables an application that executes on the event-dispatching thread (EDT) and presents a graphical user interface (GUI) to the user to remain fully interactive even while the layout algorithm performs its calculations. Moreover, the actual setup also enables modifications to the graph seemingly simultaneous to the layout calculation by `InteractiveOrganicLayouter`.

## Setup

First and foremost, the setup for using interactive organic layout relies on a thread to encapsulate the layout functionality. The thread is created and started, and subsequently begins layout calculation for the given graph. Creation of the layout thread is shown in [Example 5.22, “Creating the layout thread”](#) [115]

### Example 5.22. Creating the layout thread

```

void createAndStartLayoutThread(InteractiveOrganicLayouter iol, Graph2D graph)
{
    // Create a copy of the given graph. Using an instance of CopiedLayoutGraph
    // helps avoiding race conditions with graph *structure* modifications from a
    // user.
    final CopiedLayoutGraph clg = new CopiedLayoutGraph(graph);

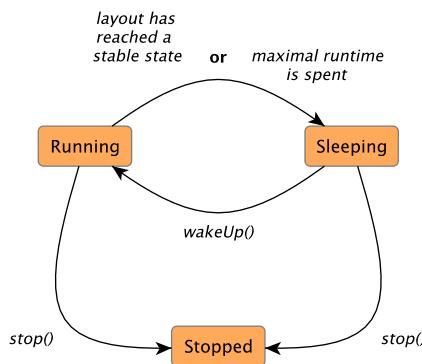
    // Create a new thread that runs the actual interactive organic layout.
    Thread layoutThread = new Thread(new Runnable() {
        public void run()
        {
            iol.doLayout(clg);
        }
    });
    // Configure and start the thread.
    layoutThread.setPriority(Thread.MIN_PRIORITY);
    layoutThread.start();
}
  
```

If the graph that is to be processed by `InteractiveOrganicLayouter` is intended to allow structural modifications by a user, then it is strongly recommended to use an instance of [CopiedLayoutGraph](#) instead of a simple `LayoutGraph`. Doing so helps avoiding race conditions and synchronization problems that occur when nodes and/or edges can be added to or removed from the graph while the layout algorithm performs its calculations.

### State

Interactive organic layout algorithm can be in either of three states: running, sleeping, or stopped. When running, the algorithm lives on the allowed maximal runtime and continuously calculates "intermediate" layouts for the given graph. When sleeping, the algorithm does nothing, but is still "alive." In contrast, when stopped, the algorithm is terminated. [Figure 5.43, "State diagram for `InteractiveOrganicLayouter`"](#) [116] depicts the states and the possible state transitions of `InteractiveOrganicLayouter`.

**Figure 5.43. State diagram for `InteractiveOrganicLayouter`**



[API Excerpt 5.13, "Methods for controlling the state of the interactive organic layout algorithm"](#) [116] lists the methods that can be used to query the current state of an `InteractiveOrganicLayouter` instance, the methods to change the state, and also the method to set the maximal runtime. Note that when an instance is changed from "sleeping" to "running" state, the maximal runtime is again fully available.

### API Excerpt 5.13. Methods for controlling the state of the interactive organic layout algorithm

```

// Getter methods to query the algorithm's current state.
boolean isRunning()
boolean isSleeping()
boolean isStopped()

// Methods to change the current state.
void wakeUp()
void stop()

// Determines the maximal runtime when in "running" state.
void setMaxTime(long maxTime)
  
```

### Polling for Results

The concurrent setup as used with `InteractiveOrganicLayouter` means that the layout process does not return a single result by itself, but instead continuously generates "intermediate" results as long as it is in "running" state. These results are in the layout algorithm's data structures only, but can be polled by a user at any time and be used as continuous updates to the layout of a graph.

The methods listed in [API Excerpt 5.14, "Node position getter methods from `InteractiveOrganicLayouter`"](#) [117] allow to easily poll either the current coordinates of single nodes or commit all

current positions of nodes to the original graph at once.

### **API Excerpt 5.14. Node position getter methods from InteractiveOrganicLayouter**

```
// Retrieve the current center coordinate(s) of a single node from
// InteractiveOrganicLayouter's data structures.
double getCenterX(Node node)
double getCenterY(Node node)
YPoint getCenter(Node node)

// Updates all node locations with the coordinates from
// InteractiveOrganicLayouter's data structures.
void commitPositions()
double commitPositionsSmoothly(double maxMovement, double percentage)
```

Typically, to assign the "intermediate" results in a continuous fashion to the original graph, a technique similar to that outlined in [Example 5.23, "Repeatedly polling InteractiveOrganicLayouter for the current node positions"](#) [117] can be used. The current positions of all nodes are repeatedly polled by means of a `javax.swing.Timer`.

### **Example 5.23. Repeatedly polling InteractiveOrganicLayouter for the current node positions**

```
void createAndStartPollingTimer(final InteractiveOrganicLayouter iol)
{
    // Create and start a Timer object that will poll the given
    // InteractiveOrganicLayouter instance every 25 milliseconds to update the
    // node positions in the original graph.
    new javax.swing.Timer(25, new ActionListener() {
        public void actionPerformed(ActionEvent ae) {
            {
                // Test if the layout algorithm is in running state.
                if (iol == null || !iol.isRunning())
                    return;
                // Smoothly update the node positions from the layout algorithm's data
                // structures to the original graph.
                iol.commitPositionsSmoothly(50, 0.15);
            }
        }).start();
}
```

## **Interaction**

The interactive organic layout algorithm allows a user to seemingly simultaneously modify node positions and also node-related parameters while layout calculation takes place. The methods listed in [API Excerpt 5.15, "Setter methods from InteractiveOrganicLayouter"](#) [117] can be used to update `InteractiveOrganicLayouter`'s data structures with new node positions, and also to specify further node-related parameters that determine, e.g., the size of a node or its inertia.

### **API Excerpt 5.15. Setter methods from InteractiveOrganicLayouter**

```
// Set the current center coordinate(s) of a single node in
// InteractiveOrganicLayouter's data structures.
void setCenterX(Node node, double x)
void setCenterY(Node node, double y)
void setCenter(Node node, double x, double y)

// Further node-related parameters used by InteractiveOrganicLayouter.
void setInertia(Node node, double inertia)
void setRadius(Node node, double radius)
```

In addition, `InteractiveOrganicLayouter` optionally also supports modifications to the original graph's structure, i.e., nodes and/or edges can be added to or removed from the graph and the layout calculation subsequently properly accounts for these changes. The setter method from [API Excerpt 5.16, "Methods related to automatic structure updates"](#) [118] has to be used to enable support for structural modifications.

## Important

To avoid race conditions and synchronization problems when structural changes to the graph are allowed, it is strongly recommended to provide `InteractiveOrganicLayouter` a copy of the original graph using an instance of `CopiedLayoutGraph`. See also the section called “Setup” [115].

### API Excerpt 5.16. Methods related to automatic structure updates

```
// Enables/disables automatic updates for structural changes in the original
// graph.
boolean isAutomaticStructureUpdateEnabled()
void setAutomaticStructureUpdateEnabled(boolean enable)
```

To properly support arbitrary structural modifications to the original graph, interactive organic layout relies on proper notification by means of graph events. To this end, a `GraphListener` is registered with the original graph that informs `InteractiveOrganicLayouter` of any structural changes that occur. See also the description of graph listeners in the section called “Events and Listeners” [25].

As an alternative to using automatic structure updates, the structure of the original graph can also be synchronized explicitly using code similar to that shown in Example 5.24, “Performing explicit structure updates” [118]

### Example 5.24. Performing explicit structure updates

```
// 'iol' is of type y.layout.organic.InteractiveOrganicLayouter.
boolean enabled = iol.isAutomaticStructureUpdateEnabled();
// Disable automatic structure updates.
iol.setAutomaticStructureUpdateEnabled(false);

// Perform the actual graph structure modifications on the original graph.
// For example, remove some nodes and edges...
modifyOriginalGraphStructureHere();

// Synchronize the structure of the original graph with that of the copy given
// to InteractiveOrganicLayouter.
iol.syncStructure();

// Reset to previous behavior for automatic structure updates.
iol.setAutomaticStructureUpdateEnabled(enabled);
```

## Supplemental Layout Data

Class `InteractiveOrganicLayouter` knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. Table 5.22, “Data provider look-up keys” [118] lists all look-up keys for `InteractiveOrganicLayouter`.

Binding supplemental layout data to a graph is described in the section called “Providing Supplemental Layout Data” [55].

**Table 5.22. Data provider look-up keys**

Key	Element Type	Value Type	Description
PRE-FERRED_EDGE_LENGTH_DATA	Edge	double	For each edge a double value that indicates its preferred length.

## Layout Options

Interactive organic layout provides a set of options that influence its behavior. These options can

be set using the setter methods of class `InteractiveOrganicLayouter`.

### Maximal Runtime (see API)

Sets the maximal duration of the layout process in milliseconds. After the layout process has finished or the specified time is spent, `InteractiveOrganicLayouter` will sleep, i.e., it has to be awakened before another layout process can be invoked (see also the description of [running/sleeping state of `InteractiveOrganicLayouter`](#)).

If this upper bound is hit during the layout process, the quality of the layout may not be optimal. Increasing this values increases the likeliness of an optimal layout.

### Preferred Node Distance (see API)

The preferred node distance which will be used to calculate the layout. The layout algorithm tries to arrange the nodes in such a way that the desired distance is complied with.

### Preferred Edge Length

Allows to specify both the general preferred length for all edges as well as individual preferred edge lengths. The layout algorithm tries to arrange the nodes in such a way that the edges have the desired edge length. Note that the edge length is measured from node border to node border.

### API Excerpt 5.17. Methods related to the preferred edge length

```
// Setter methods for both general preferred edge lengths as well as individual
// preferred edge lengths.
void setPreferredEdgeLength(double preferredEdgeLength)
void setPreferredEdgeLength(Edge edge, double newEdgeLength)
```

To specify the preferred edge length for each edge individually, a data provider holding such supplemental layout data can alternatively be bound to the graph. The data provider is expected to be registered with the graph using key `PREFERRED_EDGE_LENGTH_DATA`. Note that in the absence of an individual preferred edge length the general preferred edge length will be used.

### Important

Although the data provider look-up key defined in class `OrganicLayouter` is used to specify individual preferred edge lengths, the actual type of data provided are double values.

### Quality (see API)

This setting can be used to adjust the quality of the layout calculated by the layout algorithm. Small values lead to short running times, while greater values result in better quality. For large graph structures (hundreds and thousands of nodes) it is advisable to begin with smaller values and to gradually increase them.

### Output Restrictions

Class `InteractiveOrganicLayouter` allows to impose so-called "output restrictions" on the result of the layout calculation. More precisely, the layout result can be determined to fit into a specified region, e.g., a rectangle or a circle. Also, the layout result can be restricted to a specified aspect ratio, too.

Class `OutputRestriction` serves as a factory to create predefined output restrictions that can be used in conjunction with setter method `setOutputRestriction`.

## Tutorial Demo Code

Using the layout functionality of class `InteractiveOrganicLayouter` in an interactive environment where the user can "live"-drag nodes is presented in detail in the following tutorial demo applications:

- [InteractiveOrganicDemo.java](#)
- [NavigationDemo.java](#)
- [AnimatedNavigationDemo.java](#)



## Orthogonal Layout

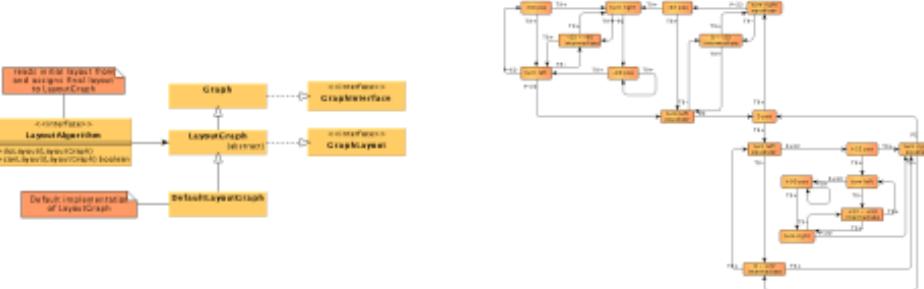
Class [OrthogonalLayouter](#) is a multi-purpose layout provider for undirected graphs. It produces clear representations of complex networks and is especially fit for application areas such as

- Software engineering
- Database schema
- System management
- Knowledge representation

The orthogonal layout algorithm is based on the topology-shape-metrics approach and consists of three phases. In the first phase the edge crossings in the drawing are calculated. The second phase computes the bends in the drawing, in the third phase the final coordinates are determined.

The layout algorithm is well suited for medium-sized sparse graphs. It produces compact drawings with no overlaps, few crossings, and few bends.

**Figure 5.44. Sample layouts produced by class OrthogonalLayouter**

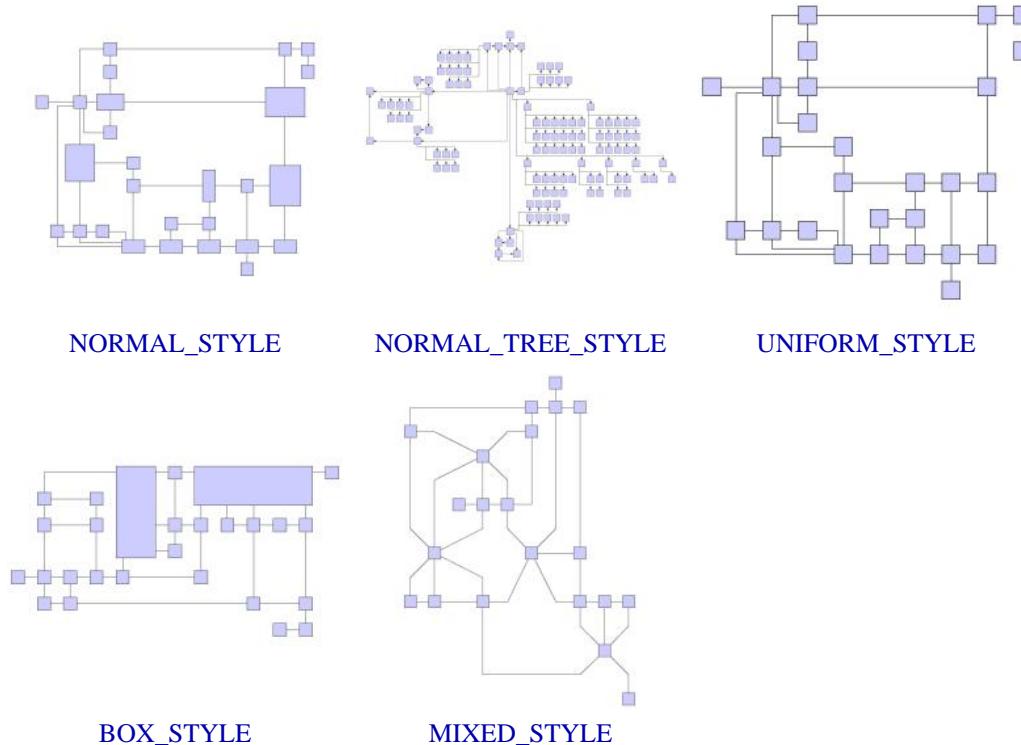


## Layout Options

---

Normal	Node sizes will not be changed by this layouter. The drawing will contain very few bends only.
Normal + Tree	Same as "Normal," but larger subtrees are processed using a specialized tree layout algorithm, which is better suited for tree-like structures than the original orthogonal layout style.
Uniform Node Sizes	All node sizes will be changed to equal size before the graph is processed.
Node Boxes	Nodes are resized according to the number and position of their neighbors to reduce the overall number of bends.
Mixed	Resembles "Node Boxes," but resizes all nodes to equal size. Introduces additional bends and routes the last edge segment of these edges non-orthogonally to their adjacent nodes.
Mixed (Size Fixed)	Proceeds similar to layout style "Mixed," but maintains original node sizes.
Node Boxes (Size Fixed)	Proceeds similar to layout style "Node Boxes," but maintains original node sizes.

**Figure 5.45. Sample layouts for the available orthogonal layout styles**



### Grid ([see API](#))

Defines the virtual grid spacing used by the layout algorithm. Each node will be placed in such a way that its center point lies on a grid point. Edges will be routed in such a way that their segments lie on grid lines, if the terminal nodes of the edges allow to place the ports accordingly. Note that this option is only guaranteed to be obeyed for "Normal Layout Style," for all other styles it is used as a hint only.

### Length Reduction ([see API](#))

If enabled, the overall edge lengths will be reduced. The cost for the reduction is an increased execution time of the layout algorithm.

### Optimize Perceived Bends (see API)

If enabled, avoids helical arrangement of chains of nodes.

### Use Existing Drawing as Sketch (see API)

If enabled, the layout algorithm will interpret the initial graph layout as a sketch of the resulting orthogonal layout. The layout algorithm tries to "orthogonalize" the given sketch without making too much modifications in respect to the original drawing.

### Crossing Postprocessing (see API)

If enabled, the overall number of edge crossings will be reduced. The cost for the reduction is increased execution time of the layouter.

### Use Randomization (see API)

If enabled, the overall layout quality will increase with high probability. The cost for this layout optimization is increased execution time and non-deterministic results for subsequent layout invocations.

## Advanced Layout Features

### Integrated Labeling

Besides the general labeling support as described in the section called “General Labeling” [158] which is available with all yFiles layout algorithms, the orthogonal layout algorithm additionally features integrated labeling. Integrated labeling is available for edge labels and can be used in conjunction with layout style “Normal.” It takes the edge labels into consideration when determining both node placement and edge path generation. With this strategy it is guaranteed that no edge label will overlap other objects in the diagram.

To specify size and preferred placement of edge labels when using integrated labeling, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key `EDGE_LABEL_LAYOUT_KEY`.

Enabling integrated labeling with `OrthogonalLayouter` and using the services of class `LabelLayoutTranslator` to conveniently have such a data provider created and bound to the graph is described in the section called “Integrated Labeling” [160]

### Orthogonal Layout of Hierarchically Organized Graphs

See the section called “Orthogonal Layout of Hierarchically Organized Graphs” [124] on how to calculate an orthogonal style layout with hierarchically organized graphs.

### Tutorial Demo Code

Layout module `OrthogonalLayoutModule.java` presents the setup of class `OrthogonalLayouter` in an application context.

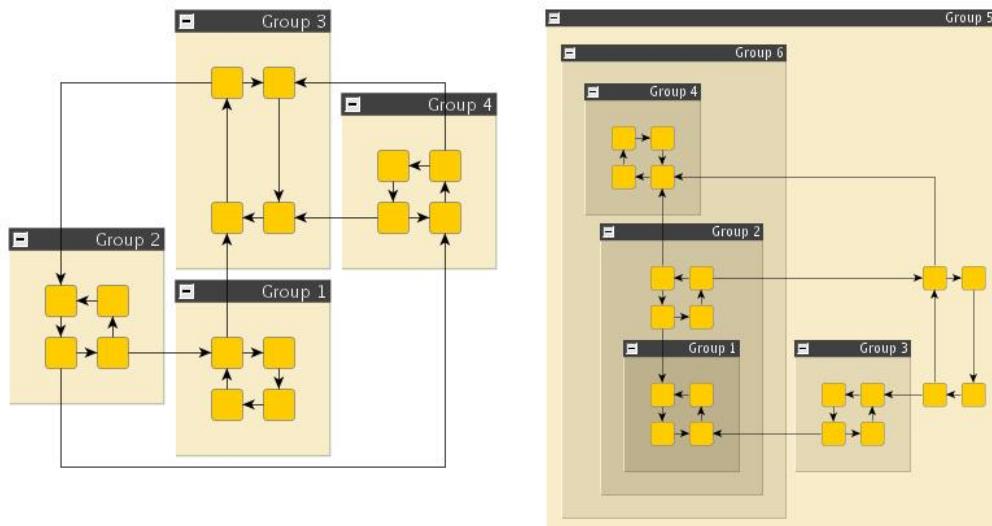
# Orthogonal Layout of Hierarchically Organized Graphs

Class [OrthogonalGroupLayouter](#) is an orthogonal layout provider specialized for laying out hierarchically organized graphs.

Those parts of a graph that belong to the same group will be placed in a way that they share a common rectangular area in the layout space. The positions and sizes of these group nodes will be calculated by the algorithm.

The layout algorithm is well suited for medium-sized sparse graphs. It produces compact drawings with no overlaps, few crossings, and few bends.

**Figure 5.46. Sample group layouts produced by class OrthogonalGroupLayouter**



A diagram with four groups and edges connecting nodes within these groups.

A diagram with grouped groups.

## Supplemental Layout Data

Class [OrthogonalGroupLayouter](#) knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.24, “Data provider look-up keys” \[124\]](#) lists all look-up keys for [OrthogonalGroupLayouter](#).

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.24. Data provider look-up keys**

Key	Element Type	Value Type	Description
<a href="#">GROUP_DPKY</a>	Node	boolean	For each node a boolean value indicating whether it is a group node or not.
<a href="#">NODE_ID_DPKY</a>	Node	Object	For each node an Object that serves as a unique ID.
<a href="#">PAR-ENT_NODE_ID_DPKY</a>	Node	Object	For each node an Object indicating the group node it belongs to. The Object matches the unique ID of a group node that is in the same graph.
<a href="#">GROUP_NODE_INS</a>	Node	Insets	For each group node a <code>java.awt.Insets</code> object

Key	Ele- ment Type	Value Type	Description
ETS_DPKEY			that encodes the group node's insets.

Setup of a graph's hierarchical organization and using the grouping keys (GROUP\_DPKEY, NODE\_ID\_DPKEY, and PARENT\_NODE\_ID\_DPKEY) is described in detail in [the section called "Setup for Layout" \[57\]](#).

## Layout Options

Unlike the more general [orthogonal layout algorithm](#) this specialized layout algorithm provides only a few options.

### Grid ([see API](#))

Defines the virtual grid spacing used by the layouter. Each node will be placed in such a way that its center point lies on a grid point. Edges will be routed in such a way that their segments lie on grid lines, if the terminal nodes of the edges allow to place the ports accordingly.

### Layout Quality ([see API](#))

Sets the desired layout quality. Higher values result in less connection crossings and smaller layout area, but also increased computation time.

The additional size requirements of group nodes (e.g., special insets or space for custom labels) can be specified by the use of a data provider that provides `java.awt.Insets` objects for each group node. The data provider must be registered with the input graph using the key `GROUP_NODE_INSETS_DPKEY`.

## Tutorial Demo Code

Layout module `OrthogonalLayoutModule.java` presents the setup of class `OrthogonalGroupLayouter` in an application context.

# Directed Orthogonal Layout

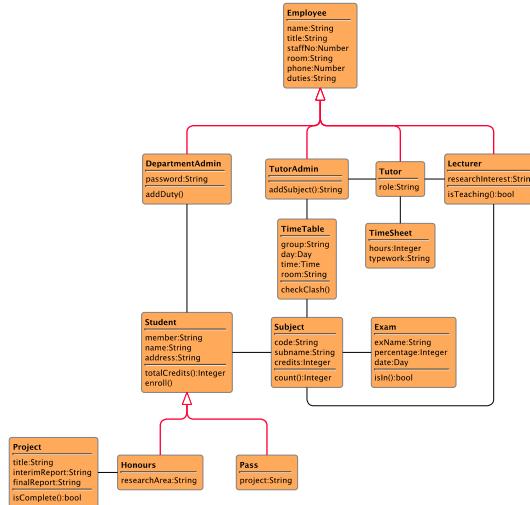
Class `DirectedOrthogonalLayouter` is an orthogonal layout provider that generates drawings with a strong focus on edge routing that satisfies sophisticated edge path-related constraints. It supports advanced edge path generation

- with respect to the so-called "main layout direction," a predefined orientation, like, e.g., bottom-to-top, and also
- using a common anchor point and common edge segments for edges that connect to a common node.

The latter is also referred to as edge/port grouping, which is described in [the section called "Edge/Port Grouping \(Bus-style Edge Routing\)" \[63\]](#).

Together these features enable high-quality graph layout which is a perfect match for UML-style class diagrams, for example. The layout algorithm is well suited for medium-sized graphs where a subset of edges should be routed with respect to the main layout direction. It produces compact drawings with no overlaps and few crossings.

**Figure 5.47. Sample layout produced by class `DirectedOrthogonalLayouter`**



A diagram with UML-style edge routing and edge groups.

## Supplemental Layout Data

Class `DirectedOrthogonalLayouter` knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.25, "Data provider look-up keys" \[126\]](#) lists all look-up keys for `DirectedOrthogonalLayouter`.

Binding supplemental layout data to a graph is described in [the section called "Providing Supplemental Layout Data" \[55\]](#).

**Table 5.25. Data provider look-up keys**

Key	Element Type	Value Type	Description
<code>DIRECTED_EDGE_DPKEY</code>	Edge	boolean	For each edge a boolean value indicating whether it should be routed having main layout direction.
<code>EDGE_LABEL_LAY</code>	Edge	La-	For each edge an array of LabelLayoutData objects

Key	Ele- ment Type	Value Type	Description
OUT_KEY		belLay- out- Data[]	that encode size and preferred placement for all labels of the edge.
SOURCE_GROUPID_KEY	Edge	Object	For each edge an arbitrary Object indicating the group its source end is affiliated with.
TAR- GET_GROUPID_KEY	Edge	Object	For each edge an arbitrary Object indicating the group its target end is affiliated with.

## Layout Options

Unlike the more general [orthogonal layout algorithm](#) this specialized layout algorithm provides only few options.

### Grid ([see API](#))

Defines the virtual grid spacing used by the layouter. Each node will be placed in such a way that its center point lies on a grid point. Edges will be routed in such a way that their segments lie on grid lines, if the terminal nodes of the edges allow to place the ports accordingly.

### Use Existing Drawing as Sketch ([see API](#))

If enabled, the layout algorithm will interpret the initial graph layout as a sketch of the resulting orthogonal layout. The layout algorithm tries to "orthogonalize" the given sketch without making too much modifications with respect to the original drawing.

## Advanced Layout Features

### Directed Edges

To specify the subset of so-called "directed" edges that should be routed with respect to the main layout direction, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key [DIREC-TED\\_EDGE\\_DPKY](#).

The main layout direction can be configured using the services of class [OrientationLayouter](#) as outlined in [Example 5.25, “Configuring the main layout direction”](#) [127]. Note that the default layout direction is top-to-bottom.

### Example 5.25. Configuring the main layout direction

```
// 'dol' is of type y.layout.orthogonal.DirectedOrthogonalLayouter.

// Set the main layout direction using the OrientationLayouter provided by
// CanonicMultiStageLayouter (the superclass of DirectedOrthogonalLayouter).
OrientationLayouter ol = dol.getOrientationLayouter();
ol.setOrientation(OrientationLayouter.LEFT_TO_RIGHT);

// Invoke buffered layout.
new BufferedLayouter(dol).doLayout(graph);
```

### Edge/Port Grouping (Bus-style Edge Routing)

A special feature of the directed orthogonal layout algorithm is its ability to group multiple ports (edge end points) together to be anchored at the same location. This can be specified for both source ports and target ports.

Edges that belong to the same group at a specific end will additionally be routed in bus-style.

To specify source groups and/or target groups for sets of edges, data providers holding such supplemental layout data must be bound to the graph. The data providers are expected to be registered with the graph using keys [SOURCE\\_GROUPID\\_KEY](#) and [TARGET\\_GROUPID\\_KEY](#), respectively. The setup of edge groups is described in the section

called “Edge/Port Grouping (Bus-style Edge Routing)” [63].

### Integrated Labeling

Besides the general labeling support as described in [the section called “General Labeling” \[158\]](#) which is available with all yFiles layout algorithms, directed orthogonal layout additionally features integrated labeling. Integrated labeling is available for edge labels, they are taken into consideration when determining both node placement and edge path generation. With this strategy it is guaranteed that no edge label will overlap other objects in the diagram.

To specify size and preferred placement of edge labels when using integrated labeling, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key [EDGE\\_LABEL\\_LAYOUT\\_KEY](#).

Enabling integrated labeling with `DirectedOrthogonalLayouter` and using the services of class `LabelLayoutTranslator` to conveniently have such a data provider created and bound to the graph is described in [the section called “Integrated Labeling” \[160\]](#)

## Tutorial Demo Code

Layout module [DirectedOrthogonalLayoutModule.java](#) presents the setup of class `DirectedOrthogonalLayouter` in an application context. Tutorial demo application [UMLClassDiagramLayouterDemo.java](#) shows how the directed orthogonal layout algorithm can be used to generate UML class diagrams.

# Compact Orthogonal Layout

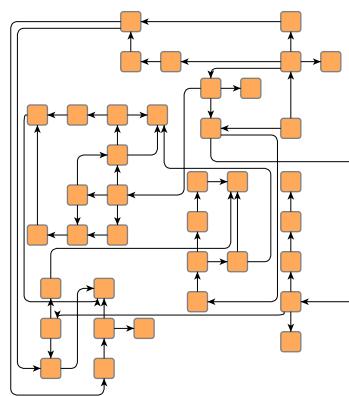
Class [CompactOrthogonalLayouter](#) is a specialized variant of the [orthogonal layout algorithm](#). It uses a divide-and-conquer approach to achieve compact orthogonal layouts that fit into, respectively are near, a desired aspect ratio.

In contrast to the majority of yFiles layout algorithms, class [CompactOrthogonalLayouter](#) does not inherit from [CanonicMultiStageLayouter](#), but is instead an extension of abstract class [AbstractLayoutStage](#).

[CompactOrthogonalLayouter](#) combines different phases that:

1. find partitions in the node set of the graph
2. calculate the orthogonal layout for the partitions
3. arrange the partitions in a compact manner
4. route the connecting edges between partitions

**Figure 5.48. Sample layout produced by class CompactOrthogonalLayouter**



A compact orthogonal diagram with an aspect ratio of 1.0

## Layout Options

This specialized layout algorithm provides only few options by itself. However, the respective classes that are used for the different layout phases can be customized individually. In particular, this also includes the [orthogonal layout algorithm](#) which is used as the core layouter.

### Grid Spacing (see API)

Defines the virtual grid spacing used by the layouter. Each node is placed in such a way that its center point lies on a grid point.

### Aspect Ratio (see API)

Specifies the desired aspect ratio that the resulting diagram should fit into. The algorithm tries to reach this value, depending on the actual graph structure, however, this may not always feasible.

## Tutorial Demo Code

Layout module [CompactOrthogonalLayoutModule.java](#) presents the setup of class [CompactOrthogonalLayouter](#) in an application context.

# Tree Layout

The yFiles tree layouter family specializes in the layout of tree-structured graphs. The need to visualize directed or undirected trees arises in many application areas, e.g.,

- Dataflow analysis
- Software engineering
- Network management
- Bioinformatics

Tree layout is provided in a number of different styles:

- Directed
- Balloon
- Horizontal/Vertical
- Compact

In addition, the [generic tree layout algorithm](#) provides the basis for a multitude of tree layout schemes.

## Enhancing the Layout Process

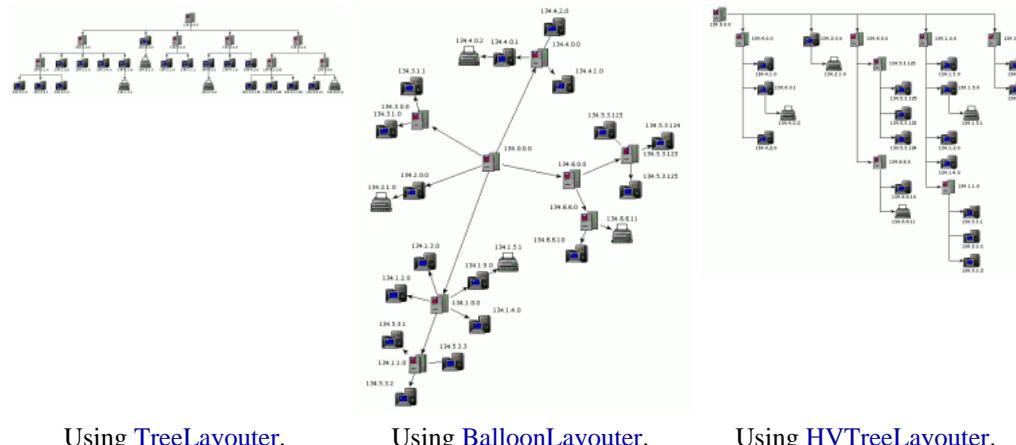
[Table 5.26, “Layout Stages”](#) [130] lists the layout stages that can be used with all tree layout algorithms to enhance the layout process. Layout module [TreeLayoutModule.java](#) presents the setup of the tree layout algorithms in an application context. It also demonstrates how to set up and use the tree reduction layout stage.

**Table 5.26. Layout Stages**

Classname	Description
<a href="#">TreeReductionStage</a>	Adds support for tree-like graphs.

Class [TreeReductionStage](#) is a layout stage that transforms graphs into proper trees. It automatically removes all non-tree edges prior to an algorithm's run and re-inserts them thereafter. When the edges are re-inserted, different routing styles can be applied.

**Figure 5.49. Sample layouts produced with the tree layouters defined in package [y.layout.tree](#)**



## Directed

Class [TreeLayouter](#) is a layouter mainly used for directed trees that have a unique root element. Starting with the root node the nodes are arranged either from top to bottom, left to right, right to left, or bottom to top. The edges of a graph can either be routed as straight lines or in an orthogonal bus-like fashion.

## Supplemental Layout Data

TreeLayouter knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.27, “Data provider look-up keys” \[131\]](#) lists all look-up keys for TreeLayouter.

Binding supplemental layout data to a graph is described in the section called “[Providing Supplemental Layout Data](#)” [55].

**Table 5.27. Data provider look-up keys**

Key	Ele- ment Type	Value Type	Description
<a href="#">EDGE_LABEL_LAYOUT_KEY</a>	Edge	La- belLay- out- Data[]	For each edge an array of LabelLayoutData objects that encode size and preferred placement for all labels of the edge.
<a href="#">NODE_LABEL_LAYOUT_KEY</a>	Node	La- belLay- out- Data[]	For each node an array of LabelLayoutData objects that encode size and preferred placement for all labels of the node.

## Layout Options

These options configure class TreeLayouter in detail.

### Minimal Layer Distance ([see API](#))

Determines the minimal distance between parent and child nodes.

### Minimal Node Distance ([see API](#))

Determines the minimal distance between the siblings of a node.

### Orientation ([see API](#))

Determines the main layout orientation. The layouter tries to arrange nodes in such a way that all edges point in the main layout direction.

Top to Bottom	The main layout orientation will be from top to bottom. Note that the documentation for the other layout options assumes that this default layout orientation is being used.
Bottom to Top	The main layout orientation will be from bottom to top.
Left to Right	The main layout orientation will be from left to right.
Right to Left	The main layout orientation will be from right to left.

### Port Style ([see API](#))

Determines the port assignment policy to be used.

Node Centered	Ports will be placed at the center of the corresponding nodes.
Border Centered	Ports will be placed at the center of the border of the corresponding nodes.
Border Distributed	Ports will be distributed along the border of the corresponding nodes.

### Orthogonal Edge Routing (see API)

If set, all edges will be routed orthogonally in a bus-like fashion. If not set, the edges will be routed as straight line segments.

### Bus Alignment

When the edges from a given (root) node to its children are routed in an orthogonal bus-like fashion, the bus can be vertically aligned within the space between the two layers using the setter method shown in [API Excerpt 5.18, “Getter/setter methods for bus alignment” \[132\]](#)

#### API Excerpt 5.18. Getter/setter methods for bus alignment

```
double getBusAlignment()
void setBusAlignment(double busAlignment)
```

### Vertical Alignment

All children of a given (root) node can be vertically aligned using the setter method shown in [API Excerpt 5.19, “Getter/setter methods for vertical alignment” \[132\]](#)

#### API Excerpt 5.19. Getter/setter methods for vertical alignment

```
double getVerticalAlignment()
void setVerticalAlignment(double verticalAlignment)
```

## Advanced Layout Features

### Integrated Labeling

Besides the general labeling support as described in [the section called “General Labeling” \[158\]](#) which is available with all yFiles layout algorithms, TreeLayouter additionally features integrated labeling.

Integrated labeling is available for both node labels and edge labels. They are taken into consideration when determining the positions for the nodes of the tree. With this strategy it is guaranteed that no label will overlap other objects in the diagram. [API Excerpt 5.20, “Support for integrated labeling” \[132\]](#) lists the methods that can be used to enable integrated labeling.

#### API Excerpt 5.20. Support for integrated labeling

```
// Getter methods.
boolean isIntegratedEdgeLabeling()
boolean isIntegratedNodeLabeling()

// Setter methods.
void setIntegratedEdgeLabeling(boolean integratedEdgeLabeling)
void setIntegratedNodeLabeling(boolean integratedNodeLabeling)
```

To specify size and preferred placement of node labels and edge labels when using integrated labeling, data providers holding such supplemental layout data must be bound to the graph. The data providers are expected to be registered with the graph using keys `NODE_LABEL_LAYOUT_KEY` and `EDGE_LABEL_LAYOUT_KEY`, respectively.

### Incremental Layout

Class TreeLayouter supports incremental layout by means of a `java.util.Comparator` implementation that provides dynamic rearrangement of all child nodes in a given subtree according to their relative coordinates. Based on this scheme, the default comparator is able to incrementally insert new child nodes at optimal positions with respect to already arranged child nodes.

The setter method shown in [API Excerpt 5.21, “Setter method for class TreeLayouter’s child comparator” \[133\]](#) permits to register custom comparator implementations that act globally on the

entire tree.

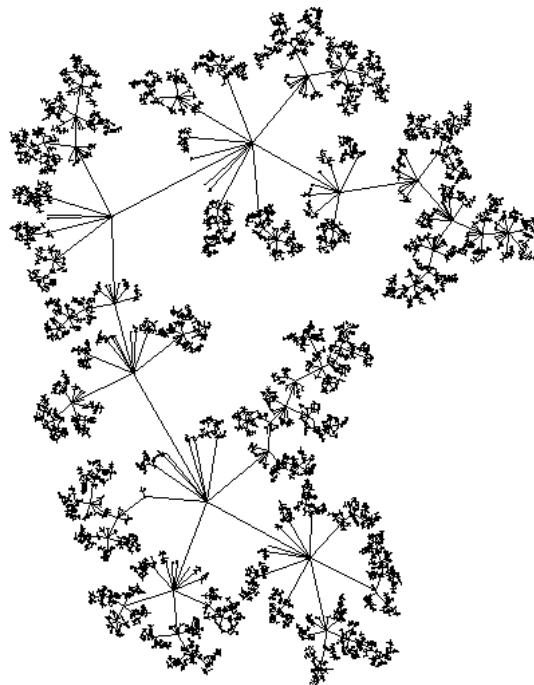
### API Excerpt 5.21. Setter method for class TreeLayouter's child comparator

```
void setComparator(Comparator childComparator)
```

## Balloon

Class [BalloonLayouter](#) is a tree layouter that positions the subtrees rooted at a node in a radial fashion around that node. It is ideally suited for huge trees (say, 10,000 nodes) since it computes fast layouts that are quite compact.

**Figure 5.50. Sample layout of a tree that contains 10,000 nodes**



## Layout Options

These options configure class BalloonLayouter in detail.

### Root Node Policy ([see API](#))

Determines which node should be used as root of the tree.

Directed Root	Chooses a node with in-degree zero, if present. A good choice for directed rooted trees.
Center Root	Chooses the root such that the depth of the resulting tree gets minimized.
Weighted Center Root	Chooses the root such that the number of paths between any two nodes that traverse the root is maximal. This seems to be a natural root for undirected trees.

### Preferred Root Wedge ([see API](#))

This setting determines the angular range of the sector that will be reserved around the root node of the graph to accommodate the attached subtrees.

### Preferred Child Wedge ([see API](#))

This setting determines the angular range of the sector that will be reserved for the children of a root node. The possible angular range lies between 1 and 359. The remaining angular range (360 minus x) will be automatically used to accommodate the edge that connects to the root node.

The smaller the chosen value, the more the impression that the nodes drive away from their root nodes and the center of the graph.

Generally speaking, the compactness of the layout will decrease with smaller values. Very small values will lead to layouts that consume a lot of space.

### **Minimal Edge Length (see API)**

Determines the minimal length of an edge. The smaller the chosen value the more compact the resulting layout.

### **Compactness Factor (see API)**

This parameter influences the length of the tree edges as it is computed by the layouter. The smaller the compactness factor, the shorter the tree-edges and the more compact the overall layout. The bigger the compactness factor, the more difficult, and hence slower, the layout computation.

### **Allow Overlaps (see API)**

If activated, this option further increases compactness of the resulting layout, but potentially introduces slight node overlaps.

## **Advanced Layout Features**

### **Incremental Layout**

BalloonLayouter can be set to "layout from sketch" mode to provide support for incremental layout. In this mode, a diagram's current drawing is taken into account when calculating a new layout. The setter method shown in [API Excerpt 5.22, "Setter method for "layout from sketch" mode" \[134\]](#) can be used to enable "layout from sketch" mode.

#### **API Excerpt 5.22. Setter method for "layout from sketch" mode**

```
void setFromSketchModeEnabled(boolean fromSketchModeEnabled)
```

### **Horizontal/Vertical**

Class HVTreeLayouter allows to layout a tree such that each subtree rooted at a node can either have a horizontal or vertical layout. For each node the layout orientation of its child nodes can be specified using the data provider look-up key **SUBTREE\_ORIENTATION**.

### **Supplemental Layout Data**

Class HVTreeLayouter knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.28, "Data provider look-up keys" \[?\]](#) lists all look-up keys for HVTreeLayouter.

Binding supplemental layout data to a graph is described in [the section called "Providing Supplemental Layout Data" \[55\]](#).

**Table 5.28. Data provider look-up keys**

Key	Element Type	Value Type	Description
<b>SUBTREE_ORIENTATION</b>	Node	Object	For each root node either <b>HORIZONTAL_SUBTREE</b> or <b>VERTICAL_SUBTREE</b> to indicate horizontal or vertical subtree placement.

## Layout Options

These options configure class HVTreeLayouter in detail.

### Horizontal Space (see API)

The minimal horizontal distance between adjacent nodes.

### Vertical Space (see API)

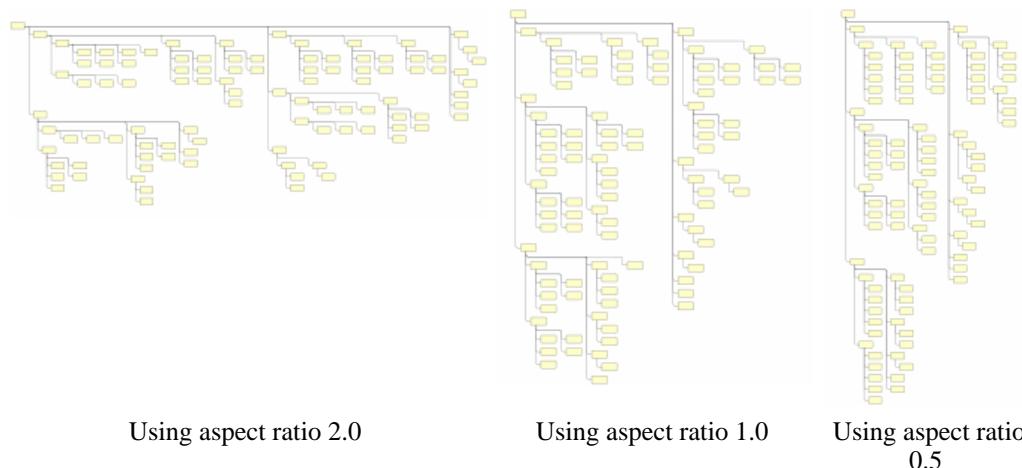
The minimal vertical distance between adjacent nodes.

Class HVTreeLayouter allows to arrange subtrees either horizontally or vertically. To specify the desired placement, a data provider holding such supplemental layout data can be bound to the graph. The data provider is expected to be registered with the graph using key `SUBTREE_ORIENTATION`. Note that in the absence of the data provider horizontal placement is chosen.

## Compact

Class `ARTreeLayouter` generates compact orthogonal tree drawings. As a layout constraint a preferred aspect ratio (relation of width to height) can be given. This is especially useful when the graph should fit perfectly on a page of given size.

**Figure 5.51. Sample layouts of the same tree using different preferred aspect ratio settings**



## Supplemental Layout Data

Class ARTreeLayouter knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. Table 5.29, “Data provider look-up keys” [?] lists all look-up keys for ARTreeLayouter.

Binding supplemental layout data to a graph is described in the section called “Providing Supplemental Layout Data” [55].

**Table 5.29. Data provider look-up keys**

Key	Element Type	Value Type	Description
<code>RATIO</code>	Node	Number	For each root node a Number object encapsulating a double value that indicates the subtree's aspect ratio.
<code>ROOT_PLACEMENT</code>	Node	Object	For each root node one of <code>PLACEMENT_TOP</code> , <code>PLACEMENT_CORNER</code> , <code>PLACEMENT_CORNER_SIDE</code> , or <code>PLACE-</code>

## Layout Options

---

Key	Ele- ment Type	Value Type	Description
			MENT_CORNER_TOP to indicate the root node's placement.
ROUTING_POLICY	Node	Object	For each root node either ROUTING_HORIZONTAL or ROUTING_VERTICAL to indicate horizontal or vertical edge routing.

## Layout Options

These options configure class ARTreeLayouter in detail.

### Horizontal Space ([see API](#))

The minimal horizontal distance between adjacent nodes.

### Vertical Space ([see API](#))

The minimal vertical distance between adjacent nodes.

### Bend Distance ([see API](#))

Determines the preferred minimal distance between each two bends of an edge and between the first and last edges and the corresponding ports.

### Preferred Aspect Ratio ([see API](#))

Determines the preferred aspect ratio (width by height) of the resulting layout. This option allows for creating layouts which, e.g., fit perfectly onto the page of a book.

## Generic Tree Layout

Class `GenericTreeLayouter` defines a generic basis for tree layout that supports a multitude of different styles and arrangements. Major aspects of layout calculation can be customized for an invocation both globally for the entire tree, but also individually per subtree. This powerful scheme allows entirely different layout styles to be applied to the subtrees of a given tree.

## Supplemental Layout Data

Class `GenericTreeLayouter` knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.30, “Data provider look-up keys” \[137\]](#) lists all look-up keys for `GenericTreeLayouter`.

Binding supplemental layout data to a graph is described in the section called “[Providing Supplemental Layout Data](#)” [55].

**Table 5.30. Data provider look-up keys**

Key	Element Type	Value Type	Description
<code>CHILD_COMPARATOR_DPKEY</code>	Node	Comparator	For each node a Comparator object that is used to sort the outgoing edges.
<code>EDGE_LABEL_LAYOUT_OUT_KEY</code>	Edge	LabelLayoutData[]	For each edge an array of LabelLayoutData objects that encode size and preferred placement for all labels of the edge.
<code>NODE_LABEL_LAYOUT_KEY</code>	Node	LabelLayoutData[]	For each node an array of LabelLayoutData objects that encode size and preferred placement for all labels of the node.
<code>NODE_PLACER_DPKEY</code>	Node	NodePlacer	For each node a <code>NodePlacer</code> implementation that places the node itself and its subtree.
<code>PORT_ASSIGNMENT_DPKEY</code>	Node	PortAssignment	For each node a <code>PortAssignment</code> implementation that assigns ports to the edges of a node.
<code>SOURCE_PORT_CONSTRAINT_KEY</code>	Edge	PortConstraint	For each edge a PortConstraint object encoding its source end's port constraint.
<code>TARGET_PORT_CONSTRAINT_KEY</code>	Edge	PortConstraint	For each edge a PortConstraint object encoding its target end's port constraint.

## Layout Options

`GenericTreeLayouter` delegates major aspects of layout calculation to implementations for interfaces `NodePlacer` and `PortAssignment`:

- `NodePlacer` is responsible for recursively arranging a subtree's root node together with its children (starting at the leaf nodes of a tree),
- `PortAssignment` encapsulates edge port assignment for a node's edges.

Classes `DefaultNodePlacer` and `DefaultPortAssignment` are used as the default implementations for `NodePlacer` and `PortAssignment`, respectively.

Both `NodePlacer` as well as `PortAssignment` implementations can be registered with `GenericTreeLayouter` to handle all nodes, i.e., act globally on the entire tree, or to handle only a specific root node, i.e., act locally on a subtree. To specify the node placer or port assignment for each subtree individually, data providers holding such supplemental layout data must be bound to the graph. The data providers are expected to be registered with the graph using keys

`NODE_PLACER_DPKEY` and `PORT_ASSIGNMENT_DPKEY`, respectively.

Note that in the absence of an individual node placer or port assignment implementation the respective default implementation will be used for a given node. [API Excerpt 5.23, “Getter/setter methods for the default node placer/port assignment implementations” \[138\]](#) lists `GenericTreeLayouter`'s methods to register the default node placer and port assignment implementations.

### API Excerpt 5.23. Getter/setter methods for the default node placer/port assignment implementations

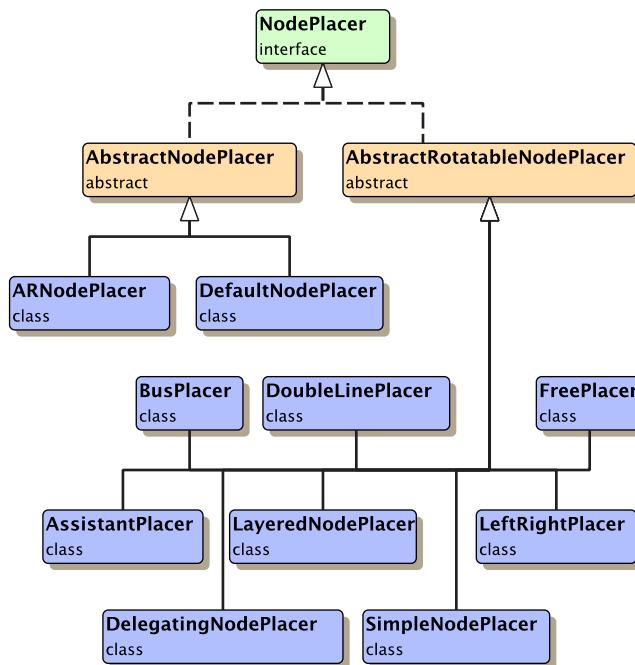
```
// Getter and setter methods for the globally used node placer implementation.
NodePlacer getDefaultNodePlacer()
void setDefaultNodePlacer(NodePlacer defaultNodePlacer)

// Getter and setter methods for the globally used port assignment
// implementation.
PortAssignment getDefaultPortAssignment()
void setDefaultPortAssignment(PortAssignment defaultPortAssignment)
```

## Node Placer Implementations

In addition to class `DefaultNodePlacer`, package `y.layout.tree` contains further predefined node placer implementations that provide a variety of subtree arrangement schemes. Their common superclass, abstract class `AbstractRotatableNodePlacer`, enables these node placers to transparently deal with child nodes that are actually rotated subtrees. [Figure 5.52, “Node placer hierarchy” \[138\]](#) depicts the node placer class hierarchy.

**Figure 5.52. Node placer hierarchy**



[Table 5.31, “Node placer implementations” \[138\]](#) lists the predefined node placer implementations that extend abstract class `AbstractNodePlacer`.

**Table 5.31. Node placer implementations**

Classname	Description
<a href="#">ARNodePlacer</a>	All subtrees are placed such that a given aspect ratio for the entire tree

Classname	Description
	is obeyed.
DefaultNodePlacer	Child nodes are placed horizontally aligned below their root node. The bus-like routing for all edges to the child nodes extends horizontally.

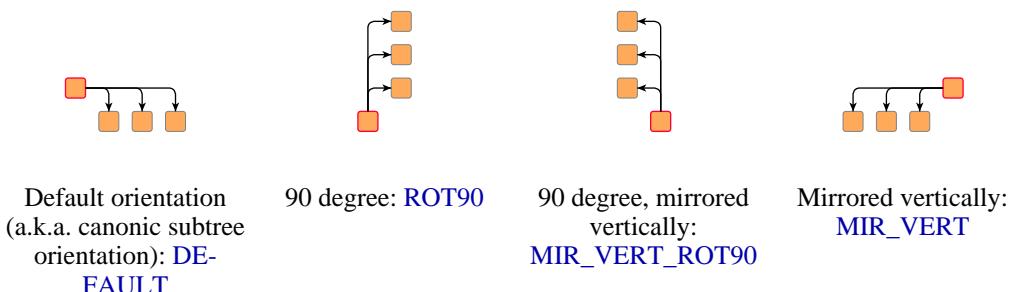
Table 5.32, “Node placer implementations that support rotation” [139] lists predefined node placer implementations that support rotated subtrees. Note that the given descriptions assume a “canonic subtree orientation,” where child nodes are placed (more or less) below their root node.

**Table 5.32. Node placer implementations that support rotation**

Classname	Description
AssistantPlacer	Applies a composite placement scheme where two kinds of nodes are differentiated; <a href="#">see below</a> for a more detailed description.
BusPlacer	Child nodes are placed evenly distributed to the left and right of their root node and also in a second line below the root node. The nodes within each line are horizontally aligned. The bus-like routing for all edges to the child nodes extends horizontally.
DelegatingNodePlacer	Applies a composite placement scheme where two node placers are used to each arrange one half of the child nodes.
DoubleLinePlacer	Child nodes are placed staggered in two lines below their root node, each line containing one half of the nodes. The nodes within each line are horizontally aligned. The bus-like routing for all edges to the child nodes extends horizontally.
FreePlacer	Child nodes can be placed anywhere, their coordinates are not modified. The bus-like routing for all edges to the child nodes extends horizontally.
LayeredNodePlacer	Subtrees can be placed such that child nodes from the same layer (i.e., nodes having the same BFS distance to the tree’s root node) are aligned.
LeftRightPlacer	Child nodes are placed below their root node to the left and right of the downward extending bus-like routing for all edges to the child nodes.
SimpleNodePlacer	Child nodes are placed horizontally aligned below their root node. The bus-like routing for all edges to the child nodes extends horizontally.

AbstractRotatableNodePlacer provides support for eight different subtree orientation schemes which are expressed using suitable modification matrices of type `AbstractRotatableNodePlacer.Matrix`. Figure 5.53, “Supported subtree orientations” [139] shows all orientations beginning with the so-called “canonic subtree orientation.” Note that the rotations provided by static inner class `AbstractRotatableNodePlacer.Matrix` are using mathematical rotation sense, i.e., they are counter-clockwise.

**Figure 5.53. Supported subtree orientations**





Mirrored horizontally: 90 degree, mirrored horizontally: 270 degree: ROT270 180 degree: ROT180  
**MIR\_HOR** **MIR\_HOR\_ROT90**

A concrete subclass of `AbstractRotatableNodePlacer` is given the modification matrix that describes the desired subtree orientation at initialization time. [API Excerpt 5.24, “Matrix-related configuration support provided by class `AbstractRotatableNodePlacer`” \[140\]](#) lists the methods/constructors that provide matrix-related configuration support.

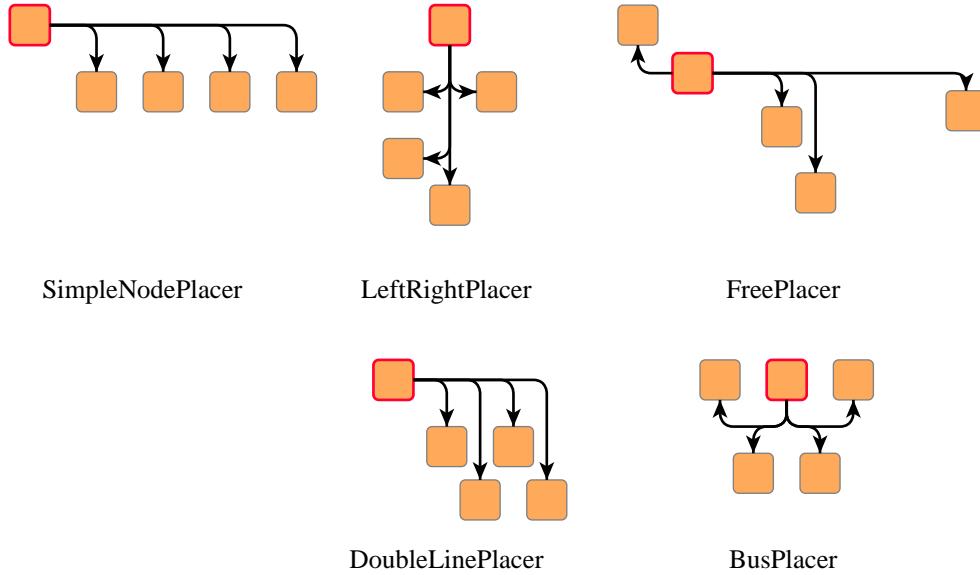
#### **API Excerpt 5.24. Matrix-related configuration support provided by class `AbstractRotatableNodePlacer`**

```
// Constructor. (To be overridden in concrete subclasses.)
protected AbstractRotatableNodePlacer(Matrix modificationMatrix)

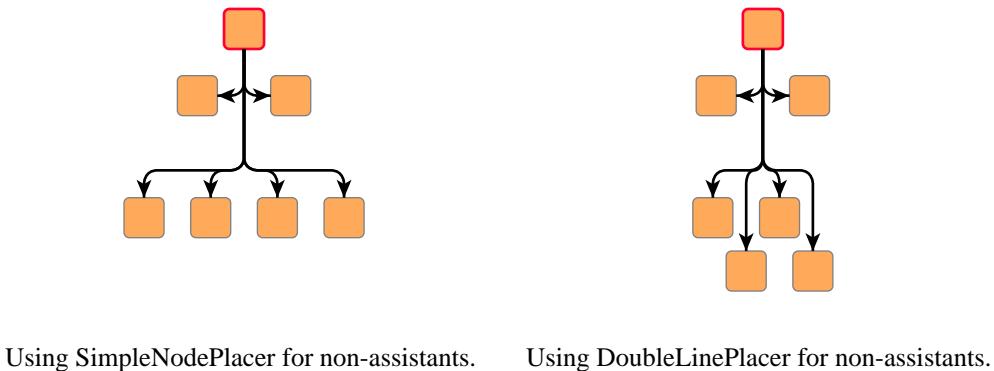
// Getter method for current modification matrix.
Matrix getModificationMatrix()
```

[Figure 5.54, “Subtree arrangement with NodePlacer implementations” \[140\]](#) shows arrangements of an example (sub)tree (in canonic subtree orientation) using different node placer implementations. Note the (sub)tree’s emphasized root node.

**Figure 5.54. Subtree arrangement with NodePlacer implementations**



Class `AssistantPlacer` uses a data provider that holds boolean values which denote whether a node is a so-called “assistant” or not. The data provider is bound to the tree using the look-up key `ASSISTANT_DPKY`. Assistants are placed using `LeftRightPlacer`, non-assistants are placed using another node placer which can be configured using a setter method. [Figure 5.55, “Subtree arrangement provided by AssistantPlacer” \[141\]](#) depicts the supported arrangements for the so-called non-assistant nodes.

**Figure 5.55. Subtree arrangement provided by AssistentPlacer**

## Advanced Layout Features

### Integrated Labeling

Besides the general labeling support as described in the section called “General Labeling” [158] which is available with all yFiles layout algorithms, generic tree layout additionally features integrated labeling.

Integrated labeling is available for both node labels and edge labels. They are taken into consideration when determining the shape of a subtree. With this strategy it is guaranteed that no label will overlap other objects in the diagram. [API Excerpt 5.25, “Support for integrated labeling”](#) [?] lists the methods that can be used to enable integrated labeling.

### API Excerpt 5.25. Support for integrated labeling

```
// Getter methods.
boolean isIntegratedEdgeLabeling()
boolean isIntegratedNodeLabeling()

// Setter methods.
void setIntegratedEdgeLabeling(boolean integratedEdgeLabeling)
void setIntegratedNodeLabeling(boolean integratedNodeLabeling)
```

To specify size and preferred placement of node labels and edge labels when using integrated labeling, data providers holding such supplemental layout data must be bound to the graph. The data providers are expected to be registered with the graph using keys `NODE_LABEL_LAYOUT_KEY` and `EDGE_LABEL_LAYOUT_KEY`, respectively.

### Port Constraints

Obeying port constraints for edges is part of the responsibilities of a node placer implementation. A node placer determines:

- the source ports of all edges that connect a subtree's root node to its child nodes
- the target port of the edge that connects to the subtree's root node

The predefined node placer implementations from package `y.layout.tree` obey both types of port constraints, weak and strong. The port constraints are retrieved from data providers that are bound to the graph using the look-up keys `SOURCE_PORT_CONSTRAINT_KEY` and `TARGET_PORT_CONSTRAINT_KEY`, respectively.

By means of interface `PortAssignment` the ports of all edges adjacent to a given tree node can be assigned. To this end, method `assignPorts` is called immediately before a subtree's node placer is invoked. Obviously, assigning edge ports at a tree node is confined to the target port of the incoming edge from the parent node, and the source ports of all outgoing edges to child nodes.

## Incremental Layout

GenericTreeLayouter can be used in combination with a `java.util.Comparator` implementation that allows to dynamically determine the sequence of child nodes in a subtree. Supplied with a root node's outgoing tree edges, the comparator decides on their precedence. Based on this scheme, the comparator that is conveniently returned by class `DefaultNodePlacer` provides support for incrementally inserting new child nodes at optimal positions with respect to already arranged child nodes.

Comparators can be registered both to handle all nodes, i.e., act globally on the entire tree, or to handle only a specific root node, i.e., act locally on a subtree. [API Excerpt 5.26, “Support for using comparators with GenericTreeLayouter” \[142\]](#) presents the methods that can be used to configure the default comparator for an entire tree.

### API Excerpt 5.26. Support for using comparators with GenericTreeLayouter

```
// Getter/setter methods for using comparators with GenericTreeLayouter.  
Comparator getDefaultChildComparator()  
void setDefaultChildComparator(Comparator defaultChildComparator)
```

To specify comparators individually for each subtree, a data provider holding such supplemental layout data must be bound to the graph. The data provider is expected to be registered with the graph using key `CHILD_COMPARATOR_DPKY`.

Class `DefaultNodePlacer` provides a suitable default comparator that can conveniently be used to support incremental tree layout scenarios with subtrees. The comparator enables dynamic rearrangement of all child nodes in a given subtree according to their relative coordinates. [Example 5.26, “Setup for incremental layout with GenericTreeLayouter” \[142\]](#) demonstrates the setup of `GenericTreeLayouter` using the default node placer and port assignment implementations, and additionally registering the default comparator from class `DefaultNodePlacer`.

### Example 5.26. Setup for incremental layout with GenericTreeLayouter

```
// Create a GenericTreeLayouter object that uses default node placer and  
// port assignment implementations, i.e., classes DefaultNodePlacer and  
// DefaultPortAssignment, respectively.  
GenericTreeLayouter gtl = new GenericTreeLayouter();  
// Register the comparator returned by class DefaultNodePlacer to take care of  
// the sequences of child nodes in each subtree.  
gtl.setDefaultChildComparator(gtl.getDefaultNodePlacer().createComparator());
```

## Tutorial Demo Code

The tutorial demo application `IncrementalTreeLayouterDemo.java` provides an extensive presentation of the powerful features that are offered by `GenericTreeLayouter` using the default node placer and port assignment implementations. It also demonstrates the incremental tree layout capabilities of `GenericTreeLayouter`. Setup and usage of `GenericTreeLayouter` with other node placer implementations is demonstrated in the following tutorial demo applications:

- `RotatableNodePlacersDemo.java`
- `AssistantPlacerDemo.java`

Also, `TreeLayoutConfigurationDemo.java` and `TreeLayoutConfiguration.java` present further advanced generic tree layout configurations.

# Edge Routing Algorithms

All edge routing algorithms provided by yFiles implement interface [LayoutStage](#). LayoutStage makes it easy to use an edge routing algorithm as a postprocessing step to any of the major layout algorithms, for example.

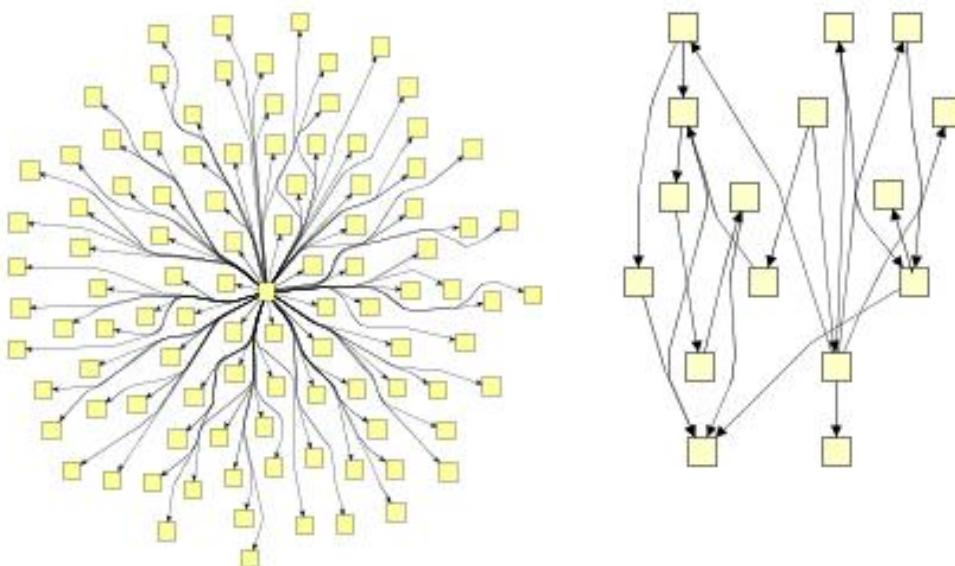
yFiles supports different edge routing styles:

- [Organic Routing](#)
- [Orthogonal Routing](#)

## Organic Routing

Organic edge routing is described in [the section called “Organic Edge Routing” \[145\]](#)

**Figure 5.56. Sample organic edge routings**



This style is encapsulated in class [OrganicEdgeRouter](#).

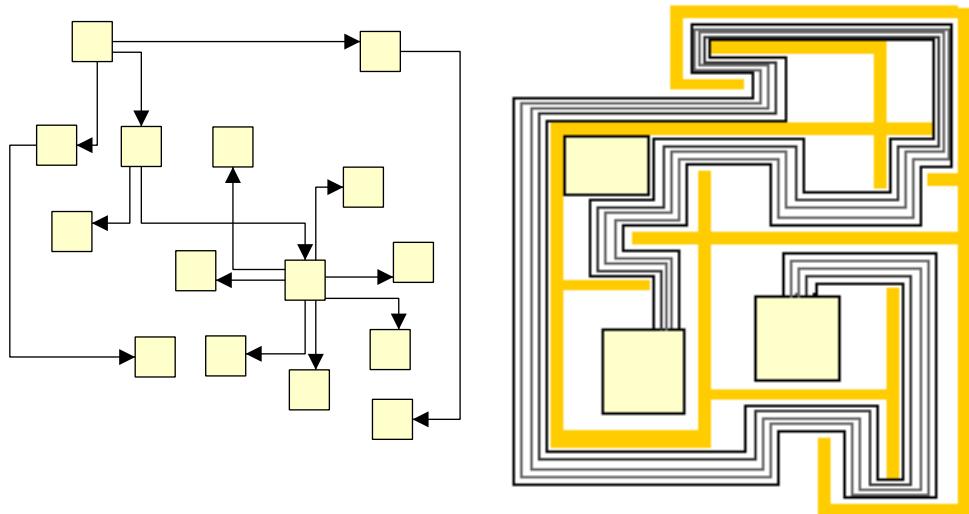
## Orthogonal Routing

Orthogonal edge routing is described in [the section called “Orthogonal Edge Routing” \[147\]](#) and [the section called “Channel Edge Routing” \[153\]](#)

**Figure 5.57. Sample orthogonal edge routings**

## Orthogonal Routing

---



This style is encapsulated in classes [OrthogonalEdgeRouter](#) and [ChannelEdgeRouter](#).

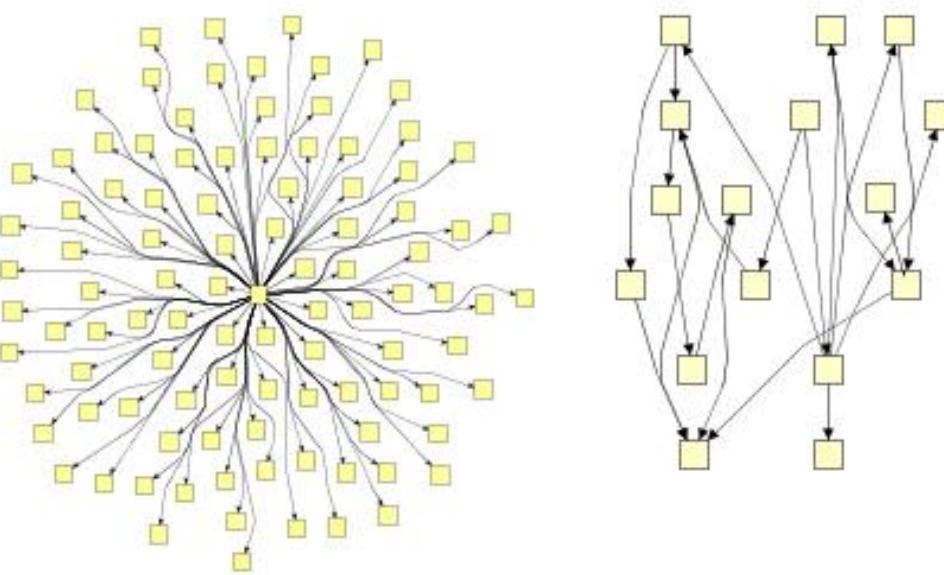
## Organic Edge Routing

[OrganicEdgeRouter](#) routes edges organically to ensure that edges do not overlap nodes and keep a specifiable minimal distance to the nodes. It is especially well suited for non-orthogonal, organic or cyclic layout styles.

The algorithm is based on a force directed layout paradigm. Nodes act as repulsive forces on edges in order to guarantee a certain minimal distance between nodes and edges. Edges themselves tend to contract themselves. Using simulated-annealing this finally leads to edge layouts, which are calculated for each edge separately.

This algorithm will only work correctly if there already is enough room between each pair of nodes in the whole graph, i.e., there should be at least three times the minimal distance room between each pair of nodes.

**Figure 5.58. Sample edge routings produced with OrganicEdgeRouter**



Postprocessing edge routing of an organically laid out tree.

Automatic edge routing of a hand-laid-out graph structure.

## Routing Options

### Route Selected Edges Only ([see API](#))

If this option is activated only selected edges will be considered for rerouting.

### Minimal Distance ([see API](#))

This specifies the minimal allowed distance between nodes and edges.

### Use Existing Bends ([see API](#))

This option specifies whether existing bends should be used as an initial solution for the new routing.

### Route Only Necessary ([see API](#))

If this option is activated only edges that violate the minimal distance criterium will be rerouted.

## Tutorial Demo Code

The following yFiles source code demo programs demonstrate how OrganicEdgeRouter can be used within an application.

## Tutorial Demo Code

---

- [OrganicLayouterDemo.java](#)

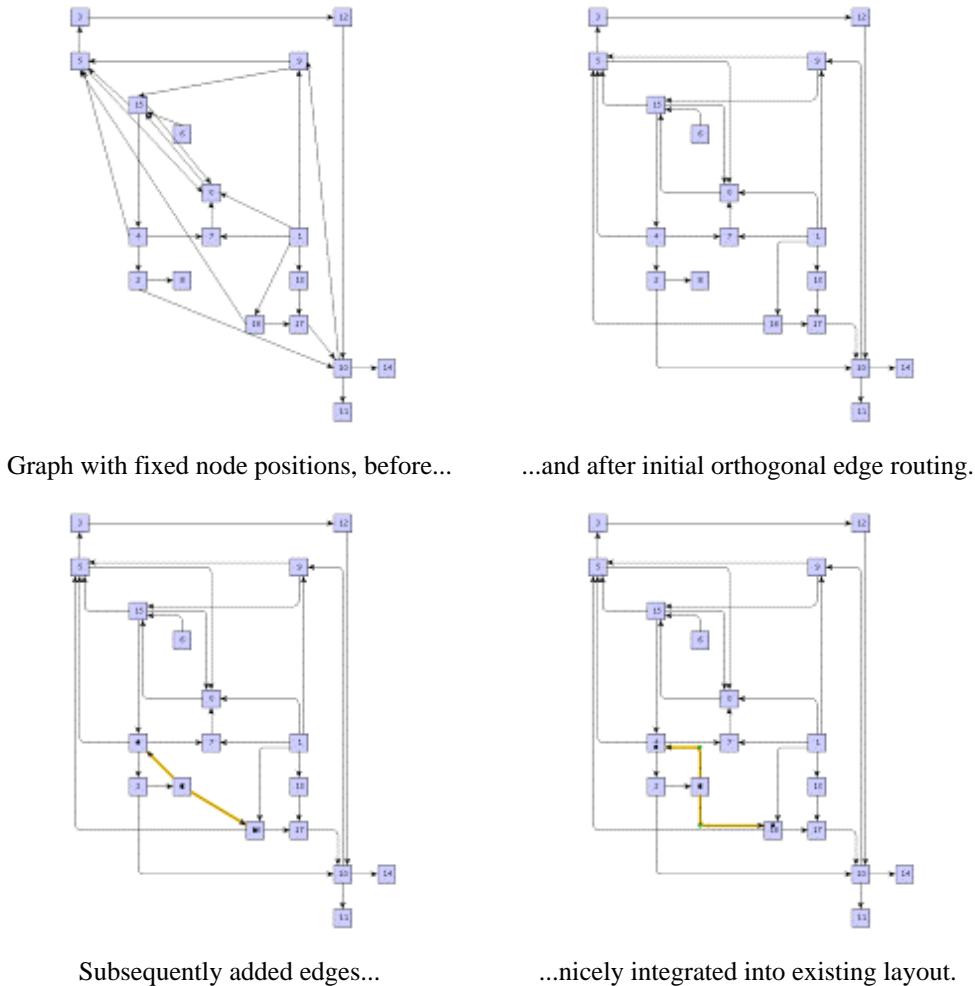
## Orthogonal Edge Routing

[OrthogonalEdgeRouter](#) is a versatile and powerful layout algorithm for routing a diagram's edges using vertical and horizontal line segments only. The positions of the diagram's nodes will remain fixed. Usually, the routed edges will not cut through any nodes or overlap any other edges.

The possibilities offered by this router make it a perfect layouter for interactive or incremental scenarios where

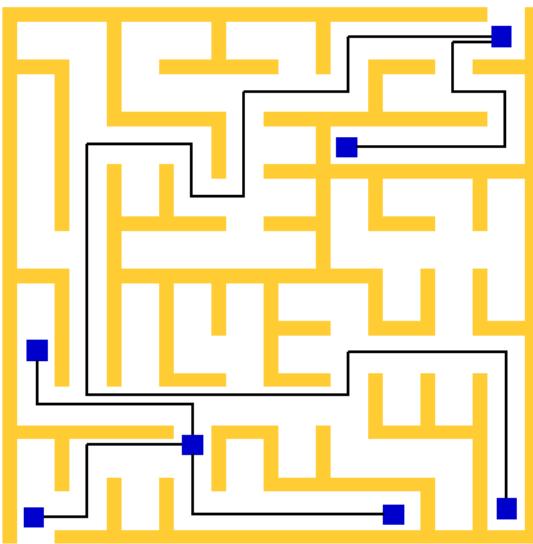
- some edges should be re-drawn orthogonally, after the user moves some nodes, or
- subsequently added edges should be drawn to fit the existing diagram.

**Figure 5.59. Common use-cases for OrthogonalEdgeRouter**



Behind the scenes of orthogonal edge router works a sophisticated path-finding algorithm that can even find routes through a maze. Not only will it find a way but also one with fewest possible changes in direction.

**Figure 5.60. Finding a way through a maze**



## Supplemental Layout Data

Class OrthogonalEdgeRouter knows a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.33, “Data provider look-up keys” \[148\]](#) lists all look-up keys for OrthogonalEdgeRouter.

Binding supplemental layout data to a graph is described in [the section called “Providing Supplemental Layout Data” \[55\]](#).

**Table 5.33. Data provider look-up keys**

Key	Element Type	Value Type	Description
SOURCE_PORT_CONSTRAINT_KEY	Edge	PortConstraint	For each edge a PortConstraint object encoding its source end's port constraint.
TAR-GET_PORT_CONSTRAINT_KEY	Edge	PortConstraint	For each edge a PortConstraint object encoding its target end's port constraint.
SOURCE_PCLIST_DPKKEY	Edge	Collection	For each edge a java.util.Collection of PortCandidate objects that encode the subset of desired anchor locations where the source port likes to connect to.
TAR-GET_PCLIST_DPKKEY	Edge	Collection	For each edge a java.util.Collection of PortCandidate objects that encode the subset of desired anchor locations where the target port likes to connect to.

## Routing Options

OrthogonalEdgeRouter provides a set of options that influence the routing behavior. This section highlights some of the configuration options available.

### Scope ([see API](#))

Determines the set of edges that the router should process.

All Edges

Routes all edges in the graph.

Selected Edges	Routes only the selected edges in the graph.
Edges at Selected Nodes	Routes only the edges connected to selected nodes.

### Minimum Edge Distance ([see API](#))

Determines the distance between any two edge segments. The edge router adheres to the set value as possible, but reduces the distance value selectively, i.e., only for a currently processed edge, when there is too little space to find a path with the proper value.

### Use Custom Minimum Distance to Nodes ([see API](#))

If set, then a custom value for minimum distance between any edge segment and any node side will be used. Otherwise, this distance will automatically be derived from the minimum distance between any two edge segments. Since this option can increase computation time, it is disabled by default.

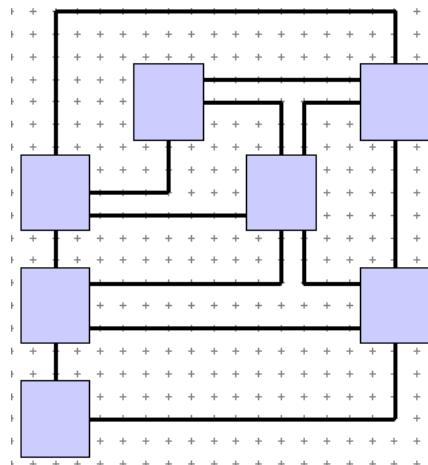
### Custom Minimum Distance to Nodes ([see API](#))

Determines the distance between any edge segment and any node side. The edge router strictly adheres to the set value. Note that this value normally is being automatically derived unless "Use Custom Minimum Distance Edge to Node" is set.

### Route on Grid ([see API](#))

If set, then all edge paths will be routed on grid lines from a predefined grid. If not set, then "free" routing will be applied to the edge paths.

**Figure 5.61. Grid routing**



Routing on grid lines with a grid spacing of 25 [pixel].

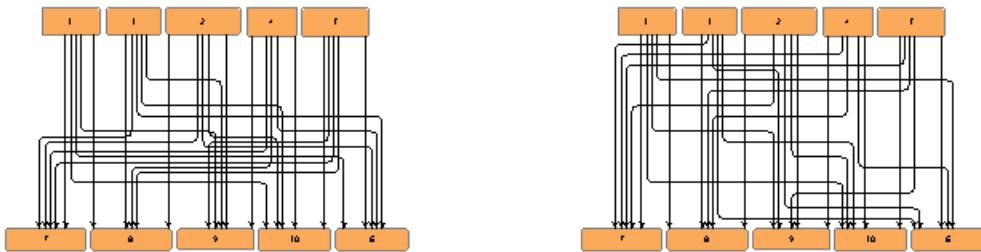
### Grid Spacing ([see API](#))

Determines the spacing of the grid lines where all edge paths will be routed upon. Grid spacing plays the same role for routing on grid lines as minimum distance between any two edge segments does for "free" routing. The edge router adheres to the set value as possible, but reduces the spacing value selectively, i.e., only for a currently processed edge, when there is too little space to find a path with the proper value.

### Space Driven Versus Center Driven Search ([see API](#))

Determines the ratio between two complementary weighting strategies when looking for an edge path, namely "center driven" and "space driven" weighting. The ratio is expressed with a value between 0.0 and 1.0. Values closer to 0.0 lead to edge paths that are more distributed over the available space. Values closer to 1.0 give more emphasis to paths near the "barycenter" of the given edge.

**Figure 5.62. Difference between "center driven" and "space driven" search strategy**



Ratio slider at 1.0 means 100% "center driven" search strategy when looking for an edge path.

Ratio slider at 0.0 means 100% "space driven" search strategy when looking for an edge path.

### Local Crossing Minimization (see API)

If not set, the number of crossings seen at a node's side can increase a lot. Since this option has a positive effect on diagram "readability," it is enabled by default.

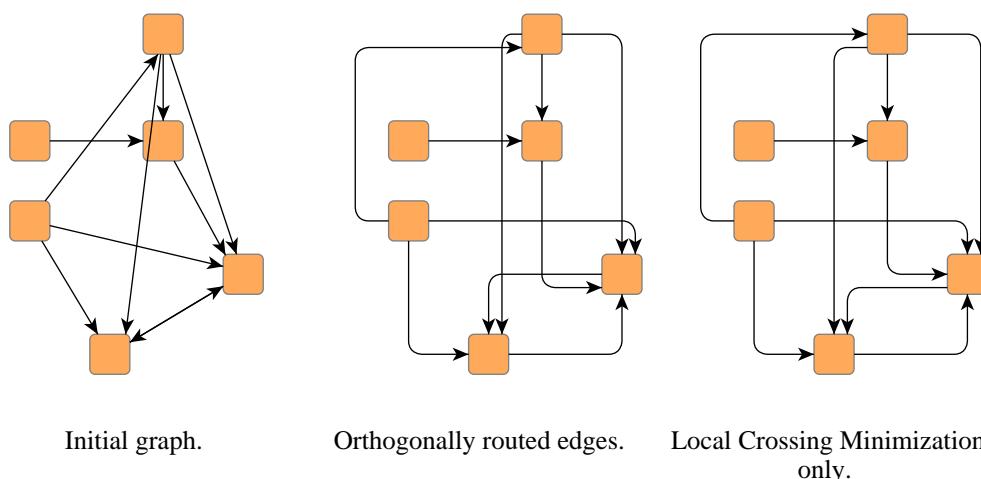
### Crossing Cost (see API)

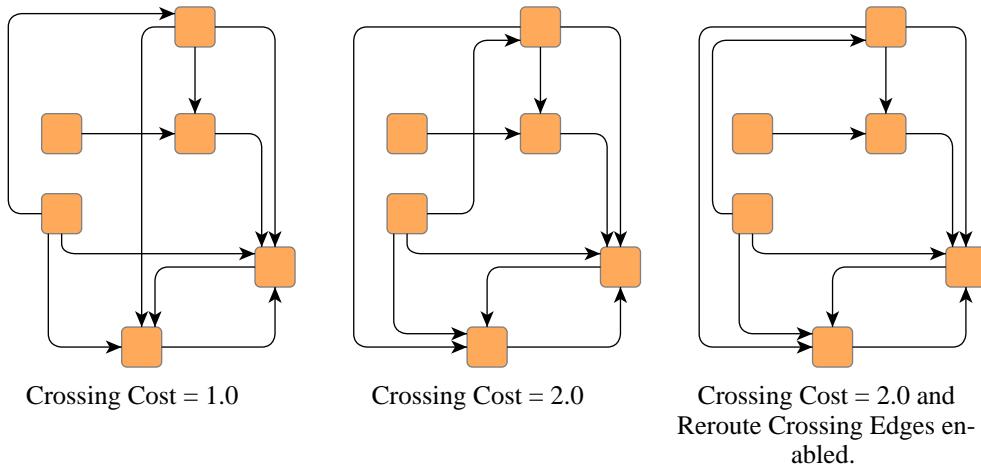
Determines a "penalty" for edge crossings. Basically, a penalty value of  $n$  means that an edge rather changes direction  $n$  times than cross an already routed edge path. In contrast to "Local Crossing Minimization" this optimization works globally, i.e., on an entire edge path. Good values for a crossing penalty lie in the range from 1.0 to 3.0. By default this value is set to 0.0, i.e., there is no penalty.

### Reroute Crossing Edges (see API)

If set, then edges with many crossings will be rerouted. This optimization makes only sense in combination with values greater than 0.0 for "Crossing Cost." By default, rerouting edges is disabled.

**Figure 5.63. Different optimization levels to minimize edge crossings**





## Advanced Routing Features

### Port Constraints

Orthogonal edge router obeys both types of port constraints, weak and strong. The port constraints are retrieved from data providers that are bound to the graph using the look-up keys `SOURCE_PORT_CONSTRAINT_KEY` and `TARGET_PORT_CONSTRAINT_KEY`, respectively.

### Port Candidates

In addition to the support provided for port constraints, orthogonal edge router also supports the concept of port candidates. The port candidates for the edges of a graph are retrieved from data providers that are bound to the graph using the look-up keys `SOURCE_PCLIST_DPKEY` and `TARGET_PCLIST_DPKEY`, respectively.

See the section called “Port Candidates” [61] for a detailed description of this concept.

### Incremental Routing

OrthogonalEdgeRouter supports incremental routing through the “Scope” feature. See the above description.

## Enhancing the Routing Process

Table 5.34, “Layout Stages” [151] lists a number of so-called layout stages that can be used to enhance the routing process of class OrthogonalEdgeRouter. Class `OrthogonalEdgeRouterModule.java` demonstrates how to set up and use these layout stages in conjunction with OrthogonalEdgeRouter.

**Table 5.34. Layout Stages**

Classname	Description
<code>EdgeGroupRouterStage</code>	Adds support for edge/port grouping, i.e., bus-style edge routing.
<code>GroupNodeRouterStage</code>	Adds support for routing so-called inter-edges, i.e., edges that cross group node boundaries in a hierarchically organized graph.
<code>PatchRouterStage</code>	Increases routing performance by decomposing the graph.
<code>ReducedSphereOfActionStage</code>	Increases routing performance when only a subset of the graph's edge set should be routed.

## Tutorial Demo Code

The following yFiles source code demo programs demonstrate how to use Ortho-

gonalEdgeRouter within an application.

- [EdgeRouterDemo.java](#)
- [MazeRouterDemo.java](#)

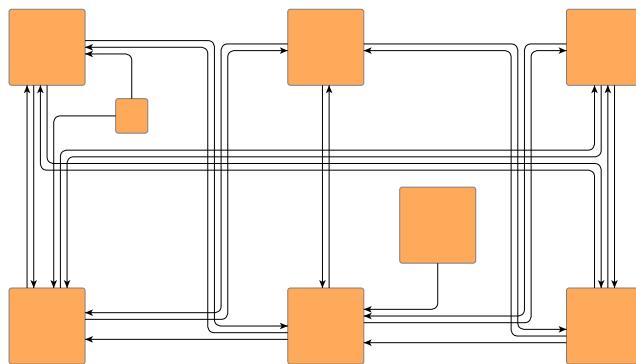
Additionally, in [UMLClassDiagramLayouterDemo.java](#) enhancing OrthogonalEdgeRouter with edge/port grouping by means of layout stage EdgeGroupRouterStage can be seen.

# Channel Edge Routing

[ChannelEdgeRouter](#) is an orthogonal edge routing algorithm that uses a two step approach for finding edge routes. First, a path finding phase assigns preliminary paths for all edges. In a second step, the distribution phase, these edge paths are then distributed so that edge segments take up the space between the nodes. The available space that an edge segment can use is also referred to as its "channel."

ChannelEdgeRouter is particularly well suited for routing bus-like structures, i.e., sets of parallel edges between common nodes. Compared to [OrthogonalEdgeRouter](#), the other yFiles orthogonal edge routing algorithm, edge segments can be very close to each other, and edges may also overlap with nodes when using the default pattern routing scheme for the path finding phase (see below). However, there are many situations where this algorithm will be faster.

**Figure 5.64. Sample edge routing produced by ChannelEdgeRouter**



By default, ChannelEdgeRouter delegates the path finding and distribution steps to specialized layout providers, namely classes [OrthogonalPatternEdgeRouter](#) and [OrthogonalSegmentDistributionStage](#), respectively. The strategies for both path finding and distribution can be easily customized.

## Supplemental Layout Data

Both the default path finding as well as the default distribution strategy of class ChannelEdgeRouter know a number of data provider keys which are used to retrieve supplemental layout data for a graph's elements. The data is bound to the graph by means of a data provider, which is registered using a given look-up key. [Table 5.35, “Data provider look-up keys”](#) [?] lists all look-up keys for ChannelEdgeRouter's default path finding and distribution strategies.

Binding supplemental layout data to a graph is described in the section called “Providing Supplemental Layout Data” [55].

**Table 5.35. Data provider look-up keys**

Key	Element Type	Value Type	Description
<a href="#">AFFECTED_EDGES</a>	Edge	boolean	For each edge a boolean value indicating whether it should be routed or not.
<a href="#">SOURCE_PORT_CONSTRAINT_KEY</a>	Edge	Port-Constraint	For each edge a PortConstraint object encoding its source end's port constraint.
<a href="#">TARGET_PORT_CONSTRAINT_KEY</a>	Edge	Port-Constraint	For each edge a PortConstraint object encoding its target end's port constraint.

## Routing Options

Key	Element Type	Value Type	Description
SOURCE_PCLIST_DPKKEY	Edge	Collection	For each edge a java.util.Collection of PortCandidate objects that encode the subset of desired anchor locations where the source port likes to connect to.
TAR-GET_PCLIST_DPKKEY	Edge	Collection	For each edge a java.util.Collection of PortCandidate objects that encode the subset of desired anchor locations where the target port likes to connect to.

## Routing Options

Class ChannelEdgeRouter itself has no special routing options beyond the setter methods listed in API Excerpt 5.27, “Setter methods for class ChannelEdgeRouter's strategies” [154] which allow to customize the strategies used for the two routing phases.

### API Excerpt 5.27. Setter methods for class ChannelEdgeRouter's strategies

```
void setPathFinderStrategy(Layouter pathFinderStrategy)
void setEdgeDistributionStrategy(Layouter edgeDistributionStrategy)
```

## Affected Edges

Determines the set of edges from the graph that should be processed. When only a subset should be routed, a data provider holding the selection state for each edge is looked up. The data provider is expected to be registered with the graph using key AFFECTED\_EDGES.

OrthogonalPatternEdgeRouter and OrthogonalSegmentDistributionStage, which are used as the default path finding and distribution strategy, respectively, both support this routing option. Also, they provide convenience methods that allow to map other already registered data providers to the AFFECTED\_EDGES key. See API Excerpt 5.28, “Setter method for mapping data providers” [154]

### API Excerpt 5.28. Setter method for mapping data providers

```
// Available in classes OrthogonalPatternEdgeRouter and
// OrthogonalSegmentDistributionStage.
void setAffectedEdgesDPKey(Object key)
```

## Advanced Routing Features

### Port Constraints

Both the default path finding as well as the default distribution strategy of class ChannelEdgeRouter obey both types of port constraints, weak and strong. The port constraints are retrieved from data providers that are bound to the graph using the look-up keys SOURCE\_PORT\_CONSTRAINT\_KEY and TARGET\_PORT\_CONSTRAINT\_KEY, respectively.

### Port Candidates

In addition to the support provided for port constraints, ChannelEdgeRouter's default path finding and distribution strategies also support the concept of port candidates. The port candidates for the edges of a graph are retrieved from data providers that are bound to the graph using the look-up keys SOURCE\_PCLIST\_DPKKEY and TARGET\_PCLIST\_DPKKEY, respectively.

See the section called “Port Candidates” [61] for a detailed description of this concept.

### Incremental Routing

ChannelEdgeRouter supports incremental routing through the “Affected Edges” feature. See the

above description.

## Related Classes

By default, [OrthogonalPatternEdgeRouter](#) is set as the path finding strategy for [ChannelEdgeRouter](#). It supports two different policies for finding edge paths, "free" routing, which is the default, and grid routing. Common to both routing policies is the general technique for finding an edge path: depending on the relative location of source and target node, an edge path is chosen from a set of possible routing "patterns." The paths from this set are associated with costs, and [OrthogonalPatternEdgeRouter](#) chooses the routing pattern that causes the least costs.

The methods listed in [API Excerpt 5.29, “Cost-related setter methods from class OrthogonalPatternEdgeRouter” \[155\]](#) can be used to specify the costs for specific situations, for example, edge segments that overlap. Note that, despite the different cost factors, the actual routing chosen by [OrthogonalPatternEdgeRouter](#) may overlap with nodes, for example, when an edge's channels offer not enough space for a proper non-overlapping route.

### API Excerpt 5.29. Cost-related setter methods from class OrthogonalPatternEdgeRouter

```
void setNodeCrossingCost(double nodeCrossingCost)
void setEdgeCrossingCost(double edgeCrossingCost)
void setEdgeOverlapCost(double edgeOverlapCost)
```

As an alternative to [OrthogonalPatternEdgeRouter](#), static inner class [ChannelEdgeRouter.OrthogonalShortestPathPathFinder](#), a specialized variant of [class OrthogonalEdgeRouter](#), can be used as the path finding strategy for [ChannelEdgeRouter](#).

## Tutorial Demo Code

Usage of class [ChannelEdgeRouter](#) can be observed in the following yFiles source code demo programs:

- [EdgeRouterDemo.java](#)
- [MazeRouterDemo.java](#)

Configuration of the channel edge routing algorithm can also be observed in [ChannelEdgeRouterModule.java](#).

# Automatic Label Placement

Both types of graph elements, i.e., nodes and edges, provide at least one label, which can be used to show a descriptive text that marks the respective graph element. The label's purpose is served best when it is both readable and also near its element, where readability most importantly results from the label being placed so that it does not overlap with other graph elements.

The yFiles library offers advanced labeling algorithms that automatically generate label arrangements that, whenever feasible, completely eliminate all overlaps, or minimize the number thereof otherwise. Labeling algorithms can be applied in two different scenarios: general labeling and integrated labeling.

## Labeling Scenarios

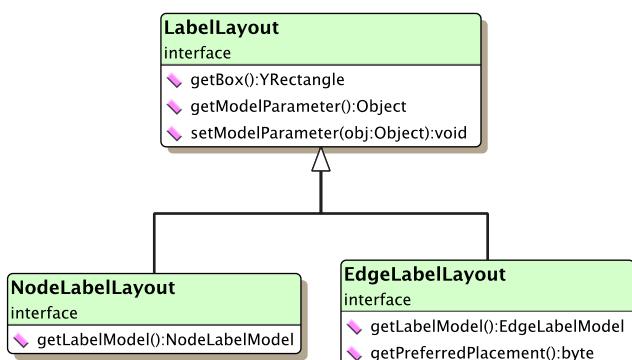
**General labeling** is independent of any layout algorithms and can be run "stand-alone" and at any time. It generates label placements without altering node positions, node sizes, or edge paths. **Integrated labeling**, in contrast, is directly provided by certain major layout algorithms as an integrated part of the layout calculation. This approach allows, e.g., to place labels before any edge path is generated, or change the position of nodes to make room for a big label.

Integrated labeling's advantage is that, by design, it prevents label overlaps completely, in other words, it can guarantee the absence of overlaps. General labeling cannot give such a guarantee, since it has to deal with a fixed environment. However, the graph's dimensions will remain constant or grow only minimally with general labeling.

## Concepts

In essence, labels are nothing more than boxes with given width and height that should be placed in such a manner that conflicts with other graph elements or labels are avoided. Interface `LabelLayout` defines the basis of all label implementations accordingly, see [Figure 5.65, “Basic interface for label implementations” \[156\]](#)

**Figure 5.65. Basic interface for label implementations**



With the means provided by the yFiles Layout distribution, label boxes are set up explicitly using `NodeLabelLayout` and `EdgeLabelLayout` implementations. [Example 5.27, “Creating label boxes” \[156\]](#) demonstrates the usage of class `EdgeLabelLayoutImpl` to define label boxes for an edge.

### Note

Users of the yFiles Viewer distribution use the functionality provided by class `YLabel` and its descendants to set up node and edge labels.

### Example 5.27. Creating label boxes

```
// Set up two boxes that represent edge labels for an edge 'e'.
```

```
// Usually, the size of a label will be determined by calculating the bounding
// box of its text, which would also account for the text's font (size, etc.).
EdgeLabelLayoutImpl ell1 = new EdgeLabelLayoutImpl();
ell1.setBox(new YRectangle(0, 0, 80, 20));

EdgeLabelLayoutImpl ell2 = new EdgeLabelLayoutImpl();
ell2.setBox(new YRectangle(0, 0, 40, 20));

// Associate the label boxes with the edge.
EdgeLabelLayout[] ells = new EdgeLabelLayout[]{ell1, ell2};
graph.setLabelLayout(e, ells);
```

## Label Models

Both node label and edge label have associated a so-called "label model," that defines a set of valid positions where the label can be placed relative to its respective graph element. In the context of automatic label placement, all valid positions are at the same time "candidates" among which a labeling algorithm can choose the best match to ultimately place the label.

### Note

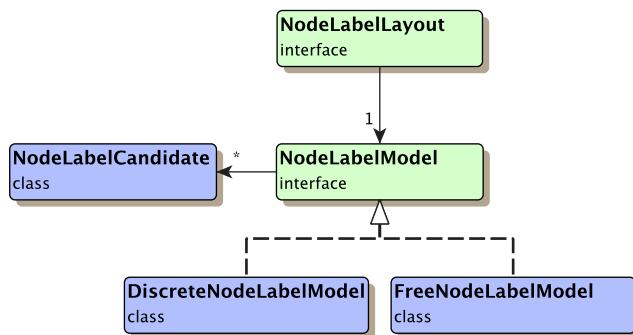
In terms of minimization of overlaps, more candidates mean better prospects for the outcome of a general labeling algorithm.

[Table 5.36, “Predefined node label models” \[157\]](#) lists the available node label models, their class hierarchy is presented in [Figure 5.66, “Predefined node label models” \[157\]](#)

**Table 5.36. Predefined node label models**

Class Name	Description
DiscreteNodeLabelModel	Defines a maximum of eight positions outside the node's bounding box, and also a maximum of nine positions inside.
FreeNodeLabelModel	Supports label placement at any given offset relative to the node's upper left corner. In effect, "free" defines a single fixed position for a label. Static inner class <a href="#">FreeNodeLabelModel.ModelParameter</a> is used to describe the label's position.

**Figure 5.66. Predefined node label models**



For general labeling, DiscreteNodeLabelModel with its up to nine valid positions (inside a node's area) is a much better choice for a node label model than FreeNodeLabelModel. However, FreeNodeLabelModel allows for an arbitrary label position, while the positions of DiscreteNodeLabelModel are all predefined.

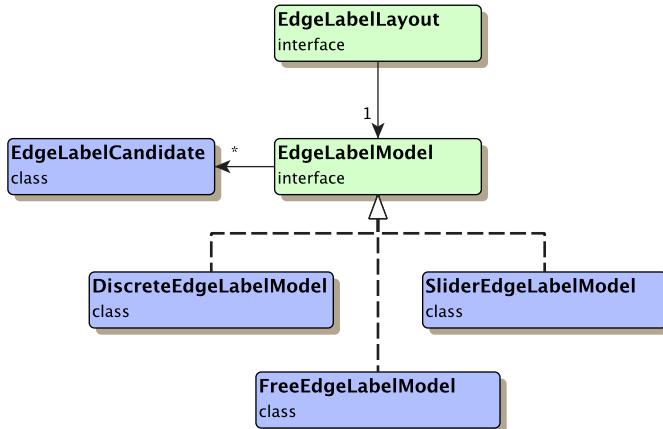
[Table 5.37, “Predefined edge label models” \[157\]](#) lists the available edge label models, their class hierarchy is presented in [Figure 5.67, “Predefined edge label models” \[158\]](#)

**Table 5.37. Predefined edge label models**

## General Labeling

Class Name	Description
DiscreteEdgeLabelModel	Defines a maximum of six positions along the edge's path, and also a maximum of three positions directly on the edge.
FreeEdgeLabelModel	Supports label placement at any given offset relative to the edge's intersection point with its source node. In effect, "free" defines a single fixed position for a label. Static inner class <code>FreeEdgeLabelModel.ModelParameter</code> is used to describe the label's position.
SliderEdgeLabelModel	Supports continuous label positions along the edge's path as well as directly on it. Static inner class <code>SliderEdgeLabelModel.ModelParameter</code> is used to describe the label's position.

**Figure 5.67. Predefined edge label models**



For general labeling, SliderEdgeLabelModel with its dynamic positions is the best choice for an edge label model, followed by DiscreteEdgeLabelModel. Both offer more variety than FreeEdgeLabelModel. However, FreeEdgeLabelModel allows for an arbitrary label position, while the positions of SliderEdgeLabelModel and DiscreteEdgeLabelModel are all directly along or on the edge path, respectively.

Integrated labeling, in contrast, favors FreeEdgeLabelModel, followed by SliderEdgeLabelModel, since it does not work with the candidates initially. Instead, it first computes an optimal location for the label, and then tries to find the best matching label position for that location in the given label model.

Associating an edge label with one of the predefined edge label models is shown in [Example 5.28, “Setting edge label models” \[158\]](#)

### Example 5.28. Setting edge label models

```

// Set center slider edge label model, where the label is placed directly on
// the edge's path.
e111.setEdgeLabelModel(
    new SliderEdgeLabelModel(SliderEdgeLabelModel.CENTER_SLIDER));

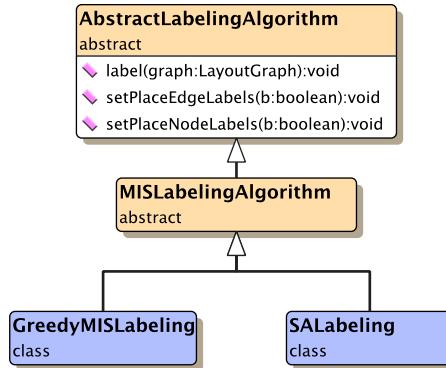
// Set side slider edge label model, where the label is placed along the edge's
// path.
e112.setEdgeLabelModel(
    new SliderEdgeLabelModel(SliderEdgeLabelModel.SIDE_SLIDER));
  
```

## General Labeling

A general labeling algorithm computes label positions for the labels from a given graph so that they, ideally, do not overlap with each other or with graph elements. It does so without altering nodes or edges in any way. As an option, the set of labels can be restricted to node labels only or

to edge labels only. [Figure 5.68, “Class hierarchy for labeling algorithms” \[159\]](#) shows the class hierarchy of the general labeling algorithms that are provided in package `y.layout.labeling`.

**Figure 5.68. Class hierarchy for labeling algorithms**



The labeling algorithm uses the model that is associated with a label to determine the available candidate positions. From the set of candidates it then chooses one that best matches the calculated position. The label model's model parameter is then used to encode this position.

### Important

The outcome of a labeling algorithm is a single model parameter that directly derives from its respective label model. The model parameter expresses the calculated label position within a given model, and is the only means to get an actual location for the label.

[Example 5.29, “Getting a label's position from its model parameter” \[159\]](#) shows how to turn the information from a model parameter into a proper location for an actual label.

### Example 5.29. Getting a label's position from its model parameter

```

/*
 * Returns the calculated location of the edge label.
 * Note that the labeling machinery returns the edge label's position as a
 * parameter of the model that belongs to the label. This model parameter can
 * be used to retrieve the actual location of the label as shown in this
 * method.
 */
YPoint getEdgeLabelLocation(LayoutGraph graph, Edge e, EdgeLabelLayout ell)
{
    YPoint ellp = ell.getLabelModel().getLabelPlacement(
        ell.getBox(),
        graph.getEdgeLayout(e),
        graph.getNodeLayout(e.source()),
        graph.getNodeLayout(e.target()),
        ell.getModelParameter());
    return ellp;
}
  
```

The base class for all major layout algorithms, class `CanonicMultiStageLayouter`, allows to conveniently enable/disable general labeling for each layout run. The labeling algorithm is then invoked after the actual layout algorithm has finished. Additionally, the labeling algorithm itself can also be set. [API Excerpt 5.30, “Labeling-related methods from class CanonicMultiStageLayouter” \[159\]](#) lists the methods provided by class `CanonicMultiStageLayouter` that are related to labeling.

### API Excerpt 5.30. Labeling-related methods from class CanonicMultiStageLayouter

```
// Getter/setter for enabling/disabling general as well as integrated labeling.
boolean isLabelLayouterEnabled()
void setLabelLayouterEnabled(boolean enabled)

// Getter/setter for the actual labeling algorithm to be used.
LayoutStage getLabelLayouter()
void setLabelLayouter(LayoutStage labeler)
```

## Integrated Labeling

Integrated labeling denotes automatic edge label placement as an integrated part of a layout algorithm. All edge labels of a graph are arranged in such a manner that there are no overlaps of edge labels with each other or with graph elements.

Table 5.38, “Layout support for integrated labeling” [160] lists the major layout algorithms that provide support for integrated labeling.

**Table 5.38. Layout support for integrated labeling**

Layout Style	Classname	Note
Hierarchical	<a href="#">HierarchicLayouter</a> , <a href="#">HierarchicGroupLayouter</a> , <a href="#">IncrementalHierarchicLayouter</a>	With class IncrementalHierarchicLayouter, integrated labeling can conveniently be enabled using appropriate configuration methods. See the descriptions of <a href="#">classic hierarchical layout</a> and <a href="#">incremental hierarchical layout</a> for more information.
Orthogonal	<a href="#">OrthogonalLayouter</a> , <a href="#">DirectedOrthogonalLayouter</a>	See the descriptions of <a href="#">orthogonal layout</a> and <a href="#">directed orthogonal layout</a> for more information.
Tree	<a href="#">TreeLayouter</a> , <a href="#">GenericTreeLayouter</a>	Integrated labeling can conveniently be enabled using appropriate configuration methods. See the descriptions of <a href="#">directed tree layout</a> and <a href="#">generic tree layout</a> for more information.

Integrated labeling relies on the services of class [LabelLayoutTranslator](#) which converts label layout information so that it is understood by a layout algorithm. Example 5.30, “Using class [LabelLayoutTranslator](#)” [160] shows how to configure integrated labeling in the absence of an appropriate configuration method. Layout algorithms that provide such a configuration method automatically ensure proper use of [LabelLayoutTranslator](#).

### Example 5.30. Using class [LabelLayoutTranslator](#)

```
void useIntegratedLabeling(Layouter layouter)
{
    // Use integrated labeling support, and let class LabelLayoutTranslator do
    // the conversion of information held by an edge label's EdgeLabelLayout
    // object to data provider-based information that is understood by the
    // algorithm.
    layouter.setLabelLayouter(new LabelLayoutTranslator());
    layouter.setLabelLayouterEnabled(true);
}
```

## Tutorial Demo Code

The tutorial demo application [LayoutWithoutAView.java](#) shows the setup for running a labeling algorithm without the presence of the yFiles library package [y.view](#). [LabelingDemo.java](#) shows the setup with classes [NodeLabel](#) and [EdgeLabel](#) available.

---

# Chapter 6. Displaying and Editing Graphs

This chapter shows how to render a graph in a so-called "view." It presents the classes that are responsible for the visual representation of graph elements and explains their contribution to the "look and feel" of an application. Furthermore, it shows how user interaction with both graph elements and the view containing a graph is handled.

## Model-View-Controller Paradigm

Package `y.view` bundles the yFiles classes that provide user interface (UI) functionality. UI functionality comprises two main aspects, namely the representation of a graph structure to the user and the interaction of a user with the graph structure.

The classes from package `y.view` reflect the Model-View-Controller (MVC) paradigm, i.e., conceptually they can be divided into three types, each having specific responsibilities:

- Model classes: hold data specific to the application
- View classes: present information to the user
- Controller classes: tie the model to the view

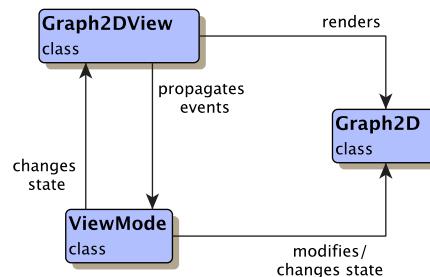
In terms of MVC then, class `Graph2D` together with classes `NodeRealizer` and `EdgeRealizer` make up the model which holds the data and its state. Class `Graph2DView` defines the view which presents the information to the user. Finally, class `ViewMode` and its descendants tie together model and view.

The respective parts of MVC are further described in the following sections:

- Model: [Graph Structure Enhancements](#) and [Bringing Graph Elements to Life: The Realizer Concept](#)
- View: [View Implementations](#)
- Controller: [User Interaction](#)

**Figure 6.1, “Model-View-Controller paradigm in package `y.view`”** [161] depicts the interdependencies of the three parts. The view (a `Graph2DView` instance) renders the model (an object of type `Graph2D`); it also propagates user interface events to the controller (class `ViewMode` and its descendants). According to these events, the controller then changes state on either view or model.

**Figure 6.1. Model-View-Controller paradigm in package `y.view`**

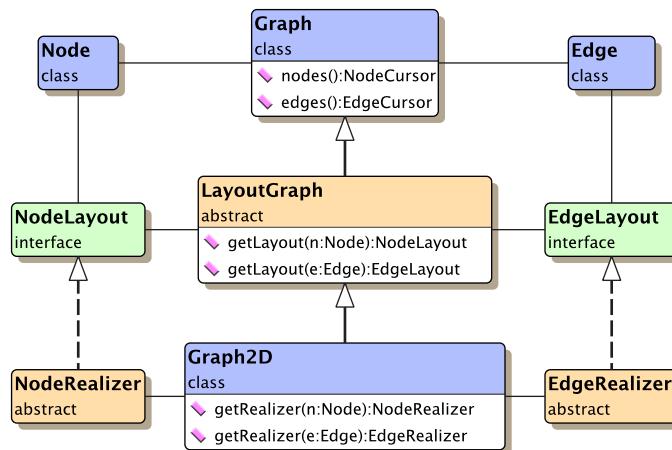


# Graph Structure Enhancements

The graph implementation in package `y.view` is provided by class `Graph2D`. It enhances the graph structure with support for the visual presentation of graph elements inside graphical user interface (GUI) components. Indispensable basis for nearly all visualization aspects is positional and dimensional information which is added by the graph structure implementation in package `y.layout`.

Abstract class `LayoutGraph` provides positional and dimensional information for all graph elements. In particular, this means width and height for a node and also coordinates for its center and upper left corner. For edges, more precisely edge paths, positional information means coordinates for both starting point and end point, as well as the control points in-between these two. [Figure 6.2, “Hierarchy of graph structure implementations” \[162\]](#) depicts the hierarchy of the `yFiles` graph structure classes.

**Figure 6.2. Hierarchy of graph structure implementations**



## Class Graph2D

Class `Graph2D` is one of the main contributors to the model part of the MVC pattern present in package `y.view`. It offers a variety of methods to conveniently modify the state of single graph elements or arbitrary collections thereof.

The other major contributors to the model part besides `Graph2D`, classes `NodeRealizer` and `EdgeRealizer` hold and manage all necessary graph element state information, and are therefore mandatory to guarantee the proper working of the user experience/the visual aspects of the graph structure.

By means of `Graph2D`'s "default realizer" mechanism, any created graph element is automatically bound to a fresh instance of its respective realizer type. The actual default node realizer and edge realizer classes that are used for this mechanism can be set in advance. Alternatively, realizers can also be associated explicitly with single graph elements, thus resetting the automatic binding.

### Note

Initially, classes `ShapeNodeRealizer` and `PolyLineEdgeRealizer` are used as the default node respectively edge realizers for the "default realizer" mechanism of class `Graph2D`.

Node realizers and edge realizers are described in all detail in [the section called “Bringing Graph Elements to Life: The Realizer Concept” \[164\]](#). There, both alternatives for associating realizers to graph elements are described, and the methods listed in [API Excerpt 6.1, “Realizer-related methods from class Graph2D” \[163\]](#) are also explained.

## API Excerpt 6.1. Realizer-related methods from class Graph2D

```
// Getter and setter methods for the default realizer mechanism.
EdgeRealizer getDefaultEdgeRealizer()
NodeRealizer getDefaultNodeRealizer()

void setDefaultEdgeRealizer(EdgeRealizer er)
void setDefaultNodeRealizer(NodeRealizer nr)

// Getter and setter methods for explicit realizers.
EdgeRealizer getRealizer(Edge e)
NodeRealizer getRealizer(Node v)

void setRealizer(Edge e, EdgeRealizer er)
void setRealizer(Node v, NodeRealizer nr)
```

To link the view part and the model, Graph2D naturally provides methods to register view objects with it, and to manage all registered views. The methods for view handling are presented in API Excerpt 6.2, “View-related methods from class Graph2D” [163]. Most prominently, method `updateViews()` is used to inform all registered views to update themselves, i.e., to repaint the representation of the graph.

## API Excerpt 6.2. View-related methods from class Graph2D

```
// Getter and setter methods.
View getCurrentView()
YCursor getViews()
void registerView(View view)
void removeView(View view)
void setCurrentView(View view)

void updateViews()
```

# Advanced Topics

## Copying a Graph2D

The generic mechanism for copying a graph structure as described in the section called “Copying a Graph” [23] can also be used to copy Graph2D objects. Instead of a GraphCopyFactory instance, however, other copy factory implementations, which are capable of copying the node realizers and edge realizers associated with a Graph2D, need to be used.

Classes `Graph2DCopyFactory` and `Graph2DCopyFactory.HierarchicGraph2DCopyFactory` are predefined copy factories that handle realizers. The latter class is returned when calling the `getGraphCopyFactory` method on a Graph2D. It is a hierarchy-aware graph copy factory that wraps a `Graph2DCopyFactory` instance and in addition to the services provided by its delegate also automatically copies all hierarchy information.

Example 6.1, “Creating a GraphCopier for a “flat” Graph2D” [163] shows how to create a GraphCopier that uses the graph copy factory for “flat” graphs instead of the default hierarchy-aware one.

### Example 6.1. Creating a GraphCopier for a “flat” Graph2D

```
// 'graph' is of type y.view.Graph2D.

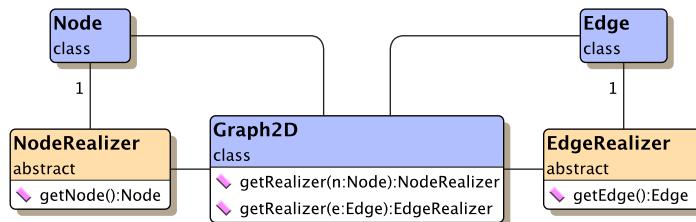
// Create a new GraphCopier that uses a graph copy factory for "flat" graphs.
GraphCopier gc = new GraphCopier(new Graph2DCopyFactory());
```

# **Bringing Graph Elements to Life: The Realizer Concept**

Graph elements are first and foremost structural elements. By themselves, they do not have a visual representation. Instead, the yFiles library utilizes so-called "realizers" to visualize the nodes and edges of a graph. Among other user interface aspects, a realizer defines the way a graph element is rendered. To this end, it has to be associated with, i.e., bound to, either node or edge, and thereafter it provides and manages all visual aspects of this graph element.

For either type of graph element there is a specialized realizer class available: class `NodeRealizer` for nodes, and class `EdgeRealizer` for edges. [Figure 6.3, “Node and edge realizers” \[164\]](#) shows the class hierarchy for the realizer types. Both classes are abstract and serve as a basis for actual implementations. For most general cases there already exist a number of predefined node and edge realizers.

**Figure 6.3. Node and edge realizers**



## **Binding Realizers**

The most convenient way to have realizers be bound to nodes and edges at creation time is to register so-called "default realizers" with a `Graph2D` object using the methods from [API Excerpt 6.3, “Support for default realizers with class Graph2D” \[164\]](#). These realizers serve as factories for realizer creation by creating copies of themselves.

### **API Excerpt 6.3. Support for default realizers with class Graph2D**

```

// Getter methods from y.view.Graph2D.
EdgeRealizer getDefaultEdgeRealizer()
NodeRealizer getDefaultNodeRealizer()

// Setter methods from y.view.Graph2D.
void setDefaultNodeRealizer(NodeRealizer r)
void setDefaultEdgeRealizer(EdgeRealizer r)
  
```

Graph elements that are created using any of the methods from [API Excerpt 6.4, “Graph element creation with default realizer bindings” \[164\]](#) are then automatically bound to the respective realizer type, i.e., to the copy of a default realizer.

### **API Excerpt 6.4. Graph element creation with default realizer bindings**

```

// Node creation resorting to default realizer binding.
// (All methods from y.view.Graph2D.)
Node createNode()
Node createNode(double x, double y)
Node createNode(double x, double y, double w, double h, String label)
Node createNode(double x, double y, String label)

// Edge creation resorting to default realizer binding.
// (Both methods from y.view.Graph2D.)
Edge createEdge(Node v, Edge e1, Node w, Edge e2, int d1, int d2)
Edge createEdge(Node v, Node w)
  
```

Alternatively, realizers can be bound to already existing nodes and edges on a per-element basis using the methods listed in [API Excerpt 6.5, “Support for explicit realizers with class Graph2D” \[165\]](#) In effect, these methods are the most powerful to associate realizers.

### API Excerpt 6.5. Support for explicit realizers with class Graph2D

```
// Getter methods from y.view.Graph2D.  
EdgeRealizer getRealizer(Edge e)  
NodeRealizer getRealizer(Node v)  
  
// Setter methods from y.view.Graph2D.  
void setRealizer(Edge e, EdgeRealizer r)  
void setRealizer(Node v, NodeRealizer r)
```

The default realizer binding at element creation time can also be circumvented using the methods from [API Excerpt 6.6, “Graph element creation with explicit realizer bindings” \[165\]](#)

### API Excerpt 6.6. Graph element creation with explicit realizer bindings

```
// Node creation with explicit realizer binding.  
// (Method from y.view.Graph2D.)  
Node createNode(NodeRealizer r)  
  
// Edge creation with explicit realizer binding.  
// (Both methods from y.view.Graph2D.)  
Edge createEdge(Node v, Edge e1,  
                Node w, Edge e2, int d1, int d2, EdgeRealizer r)  
Edge createEdge(Node v, Node w, EdgeRealizer r)
```

## General Features

Both node and edge realizers provide a number of common features that cover a variety of different aspects. The features include, e.g., support for multiple labels with a graph element, setting an element's graphical attributes, and handling its visibility and selection state. Other features, with a more technical focus, include, e.g., support for a special "less-detail" rendering mode and also for hit tests.

A realizer holds and manages an arbitrary number of labels. Its label support comprises methods to add or remove a label, to control a label's text, and to get the number of held labels. The realizers also provide factory methods to conveniently create the proper label type, i.e., either `NodeLabel` or `EdgeLabel` (see [API Excerpt 6.7, “Factory methods to create labels” \[165\]](#)). Furthermore, painting the labels lies in the responsibility of a realizer, too.

The label classes themselves are described in more detail in the section called “[Label Support](#)” [\[?\]](#). Among other things, the different positioning models for nodes and edges are explained there too.

### API Excerpt 6.7. Factory methods to create labels

```
// Defined in class NodeRealizer.  
NodeLabel createNodeLabel()  
  
// Defined in class EdgeRealizer.  
EdgeLabel createEdgeLabel()
```

Hit tests subsume a variety of related topics. These are:

- Testing whether a mouse event's coordinates are inside the bounds of a graph element.
- Calculating a graph element's bounding box.
- Calculating a graph element's compound bounding box that also includes the bounding boxes of all its labels.
- Determining whether a graph element is contained in a given rectangular area.
- Determining intersection of a graph element's bounding box and a given rectangular area.

[API Excerpt 6.8, “Hit-testing methods” \[166\]](#) lists those methods that are common to both realizer types. See the sections on the respective realizer types for more.

### API Excerpt 6.8. Hit-testing methods

```
// Defined in class NodeRealizer.
void calcUnionRect(Rectangle2D r)

boolean contains(double x, double y)
boolean intersects(double x, double y, double w, double h)

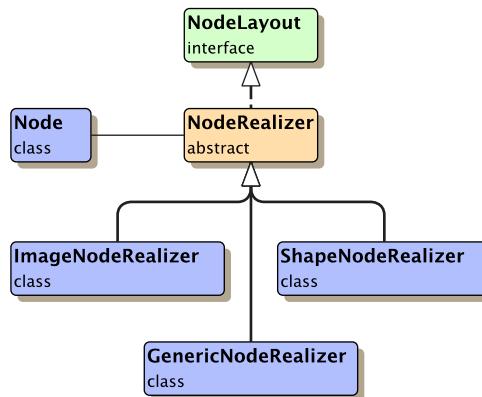
// Defined in class EdgeRealizer.
void calcUnionRect(Rectangle2D r)

boolean contains(double x, double y)
boolean intersects(Rectangle2D r)
```

## Node Realizers

Node realizers are responsible for graphically rendering nodes in a view. Abstract class `NodeRealizer` provides the basis for actual implementations. [Figure 6.4, “Node realizer hierarchy” \[166\]](#) shows the class hierarchy of the node realizer implementations.

**Figure 6.4. Node realizer hierarchy**



[Table 6.1, “Predefined node realizer implementations” \[166\]](#) lists all predefined node realizer implementations from package `y.view`.

**Table 6.1. Predefined node realizer implementations**

Classname	Description
<code>GenericNodeRealizer</code>	A generic basis for convenient new-style customization of all aspects of node representation and behavior. Out of the box, this <code>NodeRealizer</code> provides only a simple default node representation.
<code>ImageNodeRealizer</code>	A node representation that can draw an image.
<code>ShapeNodeRealizer</code>	A node representation that can draw a variety of geometric shapes.

Actual `NodeRealizer` implementations have to provide bodies for the methods listed in [API Excerpt 6.9, “Abstract methods from class NodeRealizer” \[167\]](#). These methods define the implementation’s replication behavior, and how the node’s visual representation is rendered. (Special considerations that have to be observed with realizer replication are described in [Writing Customized Realizers](#).)

### API Excerpt 6.9. Abstract methods from class NodeRealizer

```
abstract NodeRealizer createCopy(NodeRealizer nr)
protected abstract void paintNode(Graphics2D gfx)
```

API Excerpt 6.10, “Rendering methods from class NodeRealizer” [167] shows the methods that are invoked to render the visual representation of a node. The default “less-detail” implementation provided by class NodeRealizer draws the node as a simple rectangle. Note that all pre-defined node realizers resort to this default implementation.

### API Excerpt 6.10. Rendering methods from class NodeRealizer

```
void paint(Graphics2D gfx)
void repaint()
protected abstract void paintNode(Graphics2D gfx)

void paintHotSpots(Graphics2D gfx)
void paintText(Graphics2D gfx)

// Special less-detail rendering.
void paintsSloppy(Graphics2D gfx)
```

Support for hit tests is offered by the methods from API Excerpt 6.11, “Hit-testing methods” [?]. The default implementations of these methods take the bounding box of the node as the basis for their calculations. In contrast, note that method `contains` from class ShapeNodeRealizer performs shape-specific hit test computation. Also, note that method `calcUnionRect` calculates the enclosing rectangle of the node together with all its labels. Almost certainly, the result is larger than the bounding box of the node alone.

### API Excerpt 6.11. Hit-testing methods

```
// Bounding boxes.
Rectangle2D.Double getBoundingBox()
void calcUnionRect(Rectangle2D r)

// Hit-testing.
byte hotSpotHit(double x, double y)
boolean contains(double x, double y)
boolean isInBox(double x, double y, double w, double h)

// Intersection.
boolean findBBIIntersection(double ix, double iy, double ox, double oy,
                             Point2D result)
boolean findIntersection(double ix, double iy, double ox, double oy,
                        Point2D result)
boolean intersects(double x, double y, double w, double h)
```

## Class ShapeNodeRealizer

A NodeRealizer implementation where one of several possible geometric shapes is used for the visual representation of an associated node. Table 6.2, “Geometric shapes of class ShapeNodeRealizer” [167] shows a list of all shape types together with their type constants as defined in class ShapeNodeRealizer.

Note that some of the geometric shapes behave differently, e.g.:

- Shape type ELLIPSE is drawn with anti-aliasing turned on by default.
- Shape type RECT\_3D does not support transparency. Also, setting the line color has no effect, since there is no border line drawn.

**Table 6.2. Geometric shapes of class ShapeNodeRealizer**

Constant Name	Shape
ELLIPSE	
RECT	
TRIANGLE	
PARALLELOGRAM	
HEXAGON	
DIAMOND	
OCTAGON	
ROUND_RECT	
RECT_3D	
TRAPEZOID	
TRAPEZOID_2	

The shape type can be controlled using the methods listed in [API Excerpt 6.12, “Shape-related methods from class ShapeNodeRealizer” \[168\]](#)

#### API Excerpt 6.12. Shape-related methods from class ShapeNodeRealizer

```
// Getter and setter methods.
byte getShapeType()
void setShapeType(byte type)
```

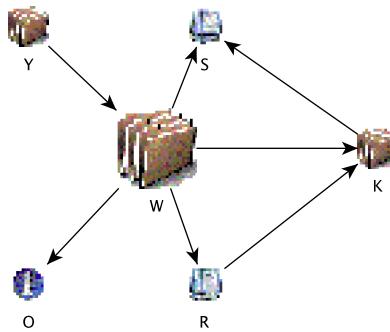
For tutorial demo code see:

- [StateNodeRealizer.java](#)
- [UMLClassNodeRealizer.java](#)

#### Class ImageNodeRealizer

A NodeRealizer implementation where an image is used for the visual representation of an associated node. The image scales accordingly when the node gets resized. [Figure 6.5, “Nodes represented by ImageNodeRealizer” \[168\]](#) shows a graph where ImageNodeRealizer is used.

#### Figure 6.5. Nodes represented by ImageNodeRealizer



Class [ImageNodeRealizer](#) provides methods to specify the image using an URL or to give it directly. (Note that images that are specified by an URL have to have GIF or JPG image file format.) [API Excerpt 6.13, “Image-related methods from class ImageNodeRealizer” \[169\]](#) lists these methods.

### API Excerpt 6.13. Image-related methods from class ImageNodeRealizer

```
// Getter methods.
Image getImage()
URL getImageURL()

// Setter methods.
void setImage(Image image)
void setImageURL(URL imageURL)
```

The alpha transparency of an image can optionally be used to modify the image's hit-testing behavior. Pixels of an image that are more translucent than opaque are considered irrelevant for the hit test. The setter method shown in [API Excerpt 6.14, “Alpha transparency support for images” \[169\]](#) can be used to determine whether alpha transparency should be considered.

### API Excerpt 6.14. Alpha transparency support for images

```
// Getter/setter methods from y.view.ImageNodeRealizer.
boolean isAlphaImageUsed()
void setAlphaImageUsed(boolean use)
```

Furthermore, the class holds an image cache where all URL-specified images are stored so that they can be retrieved from this cache. The image cache can conveniently be filled using a number of static methods. [API Excerpt 6.15, “Image cache methods from class ImageNodeRealizer” \[169\]](#) lists the image cache-related methods from class ImageNodeRealizer.

### API Excerpt 6.15. Image cache methods from class ImageNodeRealizer

```
// Getter methods.
static Vector availableImageURLs()
static Image getImage(URL url)
static Hashtable getImageTable()

// Setter methods.
static Image addImage(URL imageURL)
static void addImages(String userHomeRes, String classPathRes)
static void addImages(URL url)
```

### Note

The URL that has been used to specify a node's image is also used when the graph containing the node is written to file. To achieve uniqueness, however, the URL is made

absolute.

## Class GenericNodeRealizer

This NodeRealizer implementation primarily serves as a generic basis for node realizer customization. It defines a set of static inner interfaces which allow fine-grained control over all aspects of a node realizer's visual representation and behavior. Instead of applying subclassing and overriding, custom logic for a specialized realizer can be conveniently expressed by providing implementations for only a selection or for all of these interfaces.

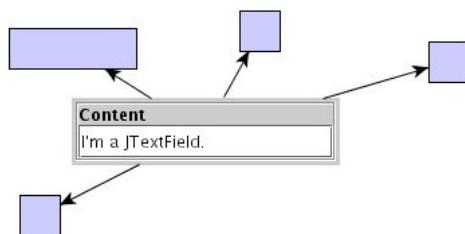
Class [GenericNodeRealizer](#) offers all methods from its superclass [NodeRealizer](#), the actual work, though, is delegated to the following interfaces:

- [GenericNodeRealizer.Painter](#) to render the node's visual representation
- [GenericNodeRealizer.HotSpotPainter](#) to add the resize knobs whenever a node is selected
- [GenericNodeRealizer.HotSpotHitTest](#) to handle hit testing on the node's resize knobs
- [GenericNodeRealizer.ContainsTest](#) to perform hit tests
- [GenericNodeRealizer.IntersectionTest](#) to perform intersection testing
- [GenericNodeRealizer.UnionRectCalculator](#) to calculate the node's bounds
- [GenericNodeRealizer.BoundsChangedHandler](#) to handle changes of the node's bounds
- [GenericNodeRealizer.PortCandidateListProvider](#) to provide coordinates for possible port locations
- [GenericNodeRealizer.GenericMouseInputEditorProvider](#) to provide [MouseInputEditor](#) instances which handle mouse-based interaction
- [GenericNodeRealizer.UserDataHandler](#) to govern both copying and serializing of user-defined data

Together, a set of actual implementations for these interfaces forms a so-called "configuration" that defines the look and feel of one node type. Different configurations can be used to define a variety of node types.

For example, using class [NodeCellRendererPainter](#), which is an implementation of interface [GenericNodeRealizer.Painter](#), a configuration that supports rendering of Swing UI components as seen in [Figure 6.6, “Swing UI component rendered by GenericNodeRealizer” \[170\]](#) can be defined.

**Figure 6.6. Swing UI component rendered by GenericNodeRealizer**



The tutorial demo application [SwingRendererDemo.java](#) shows how such a configuration can be defined. (See also [Swing User Interface Components as Node Realizers](#) for more information on the Swing UI components rendering support provided by the yFiles library.)

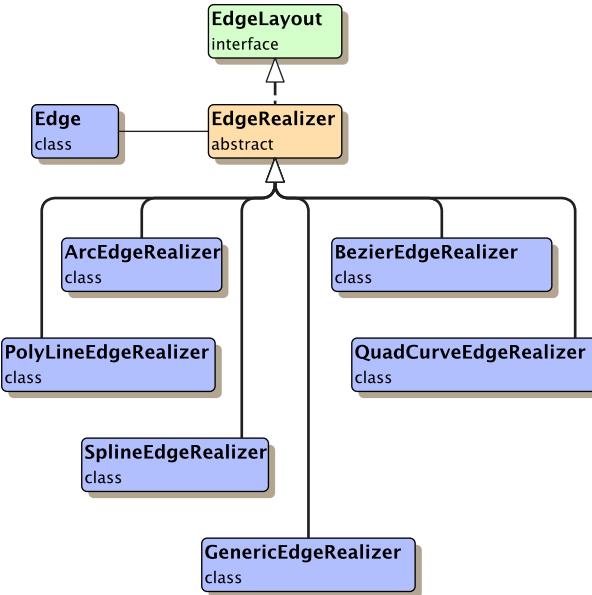
Out of the box, [GenericNodeRealizer](#) uses its default configuration, which results in a simple rectangular node representation similar to that of [ShapeNodeRealizer](#) as shown above. To take full advantage of the concept that is behind class [GenericNodeRealizer](#), custom configurations must be defined and registered. The section [Writing Customized Realizers](#) discusses the necessary steps.

## Edge Realizers

Edge realizers are responsible for graphically rendering edges in a view. Abstract class [EdgeR-](#)

realizer provides the basis for actual implementations. [Figure 6.7, “Edge realizer hierarchy” \[171\]](#) shows the class hierarchy of the edge realizer implementations.

**Figure 6.7. Edge realizer hierarchy**



[Table 6.3, “Predefined edge realizer implementations” \[171\]](#) lists all predefined edge realizer implementations from package `y.view`.

**Table 6.3. Predefined edge realizer implementations**

Classname	Description
<code>GenericEdgeRealizer</code>	A generic basis for convenient new-style customization of all aspects of edge representation and behavior. Out of the box, this <code>EdgeRealizer</code> provides only a simple default edge representation.
<code>ArcEdgeRealizer</code>	An edge representation that draws a simple arc.
<code>BezierEdgeRealizer</code>	An edge representation that draws a spline curve where the existing bends are interpreted as control points for the curve. No control point actually lies on the edge path.
<code>PolyLineEdgeRealizer</code>	An edge representation that draws direct line connections from bend to bend.
<code>QuadCurveEdgeRealizer</code>	An edge representation that draws a spline curve where the existing bends are interpreted as control points for the curve. No control point actually lies on the edge path.
<code>SplineEdgeRealizer</code>	An edge representation that draws a spline curve where the existing bends are interpreted as control points for the curve. All control points lie on the edge path.

Actual `EdgeRealizer` implementations have to provide bodies for the methods listed in [API Excerpt 6.16, “Abstract methods of class `EdgeRealizer`” \[172\]](#). These methods determine bend handling, edge path calculation, and also the implementation’s replication behavior. (Special considerations that have to be observed with realizer replication are described in [Writing Customized Realizers](#).)

### Note

"Bend" and "control point" are two terms for the same thing; class `Bend` defines the visual counterpart for a control point in an edge path.

### API Excerpt 6.16. Abstract methods of class EdgeRealizer

```
// Bend-related methods.
abstract Bend createBend(double x, double y, Bend refBend, int dir)
abstract Bend insertBend(double x, double y)
abstract void reInsertBend(Bend bend, Bend refBend, int dir)
abstract Bend removeBend(Bend bend)

// Path calculation method. Returns the intersection coordinates at both source
// node and target node.
protected abstract byte calculatePath(Point2D sourceIntersectionOut,
                                         Point2D targetIntersectionOut)

// Replication method.
abstract EdgeRealizer createCopy(EdgeRealizer er)
```

[API Excerpt 6.17, “Rendering methods from class EdgeRealizer” \[172\]](#) shows the methods that are invoked to render the visual representation of a edge. The default “less-detail” implementation provided by class EdgeRealizer draws the edge as a simple poly-line path without any decorations and labels. Note that all predefined edge realizers resort to this default implementation.

### API Excerpt 6.17. Rendering methods from class EdgeRealizer

```
void paint(Graphics2D gfx)
void repaint()

protected void paintArrows(Graphics2D gfx)
protected void paintHighlightedBends(Graphics2D gfx)
protected void paintLabels(Graphics2D gfx)
protected void paintPorts(Graphics2D gfx)

// Special less-detail rendering.
void paintSloppy(Graphics2D gfx)
```

Support for hit tests is offered by the methods from [API Excerpt 6.18, “Hit-testing methods” \[?\]](#). Note that the intersection methods are mainly used to determine whether an edge lies inside a given rectangular area. In contrast, the hit-testing methods check whether a mouse event's coordinates lie on the edge's path.

### API Excerpt 6.18. Hit-testing methods

```
// Bounding box.
void calcUnionRect(Rectangle2D r)

// Performing hit tests.
boolean contains(double x, double y)
int containsSeg(double x, double y)

// Intersection.
boolean intersects(Rectangle2D r)
boolean pathIntersects(Rectangle2D r, boolean considerLabels)
```

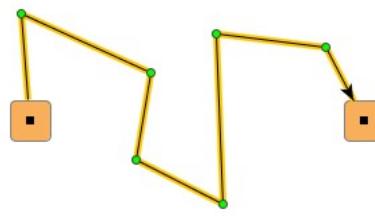
## Class PolyLineEdgeRealizer

An EdgeRealizer implementation where the associated edge is drawn

- as a straight line segment connecting its actual end points (if the edge does not have any bends), or
- as a sequence of straight line segments that forms a path connecting end points and the bend(s) in-between (if the edge has at least one bend).

[Figure 6.8, “Edge represented by PolyLineEdgeRealizer” \[173\]](#) shows an example of a poly-line edge that has six bends and hence seven edge segments. (The straight line segments of an edge are called “edge segments.”)

**Figure 6.8. Edge represented by PolyLineEdgeRealizer**



Class `PolyLineEdgeRealizer` provides a bend-smoothing feature which can be controlled using the methods from API Excerpt 6.19, “Bend-smoothing methods from class `PolyLineEdgeRealizer`” [173]

#### API Excerpt 6.19. Bend-smoothing methods from class `PolyLineEdgeRealizer`

```
// Getter and setter method for bend-smoothing.
boolean getSmoothedBends()
void setSmoothedBends(boolean b)
```

A special rendering behavior is applied for self-loops with at most one bend:

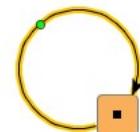
- If the edge has no bends, then there is one inserted at a default location relative to the self-loop's node. The edge is then drawn as a circle with a default radius as depicted in Figure 6.9, “Automatic self-loop rendering behavior with class `PolyLineEdgeRealizer`” [173] (left).
- If the edge already has a bend, it is used to determine the circle's diameter. The edge is then drawn as shown in Figure 6.9, “Automatic self-loop rendering behavior with class `PolyLineEdgeRealizer`” [173](right).

Self-loops that have more than one bend are drawn normally.

**Figure 6.9. Automatic self-loop rendering behavior with class `PolyLineEdgeRealizer`**

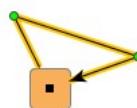


Bend-less self-loop.



Single-bend self-loop.

**Figure 6.10. Standard self-loop rendering behavior**



Standard rendering for a self-loop with more than one bend.

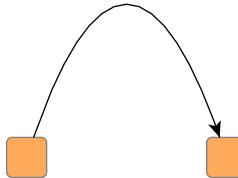
#### Note

The applied "less-detail" rendering does neither smoothed bends nor special self-loop rendering.

### Class ArcEdgeRealizer

An EdgeRealizer implementation where the associated edge is drawn as a simple arc connecting its end nodes. The arc is drawn using a single bend that always lies on the perpendicular bisector of the straight line segment between the arc's actual end points. [Figure 6.11, “Edge represented by ArcEdgeRealizer” \[174\]](#) shows an arc example.

**Figure 6.11. Edge represented by ArcEdgeRealizer**



As a consequence of the single-bend design all excessive bends of an edge are actually deleted when it is associated an instance of type `ArcEdgeRealizer`. Also, once an edge is associated an `ArcEdgeRealizer` instance, the number of bends cannot be changed. [API Excerpt 6.20, “No-op methods from class ArcEdgeRealizer” \[174\]](#) lists the bend-handling methods that have empty bodies in class `ArcEdgeRealizer`.

### API Excerpt 6.20. No-op methods from class ArcEdgeRealizer

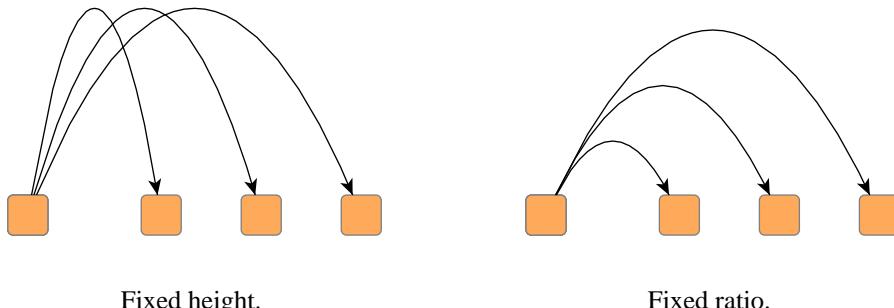
```
// No-op methods. (Returning 'null' or having empty body.)
Bend createBend(double x, double y, Bend refBend, int dir)
Bend insertBend(double x, double y)
void reInsertBend(Bend bend, Bend refBend, int dir)
Bend removeBend(Bend b)
```

Class `ArcEdgeRealizer` supports two different arc behaviors: fixed height and fixed ratio (height to distance). The behavior defines the way the arc is drawn when the position of one of the arc's end nodes changes.

Fixed height means that the height of the perpendicular of the straight line segment between the arc's actual end points remains fixed regardless of the distance between the end points. In contrast, fixed ratio means that the ratio of height to distance remains fixed.

[Figure 6.12, “ArcEdgeRealizer behaviors” \[174\]](#) demonstrates the difference in behavior when the distance between the arc's end points is reduced. Fixed height behavior retains the arc's height, while fixed ratio yields a scaled version of the original arc.

**Figure 6.12. ArcEdgeRealizer behaviors**



[API Excerpt 6.21, “Behavior-related methods from class ArcEdgeRealizer” \[175\]](#) lists the methods provided by class ArcEdgeRealizer that can be used in relation with arc behavior.

### API Excerpt 6.21. Behavior-related methods from class ArcEdgeRealizer

```
// Getting/setting the arc behavior.
byte getArcType()
void setArcType(byte t)

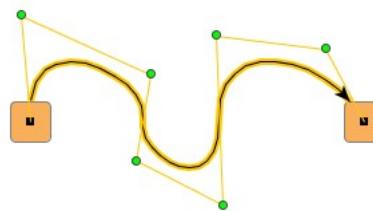
// Getter and setter methods for either arc behavior.
float getHeight()
float getRatio()
void setHeight(float h)
void setRatio(float r)
```

## Class BezierEdgeRealizer

An EdgeRealizer implementation where the associated edge is drawn as a bezier spline curve. The edge's bends are interpreted as control points for the curve, which, by construction, are not part of the curve path.

[Figure 6.13, “Edge represented by BezierEdgeRealizer” \[175\]](#) shows an example of a bezier spline curve rendered by class `BezierEdgeRealizer`. The edge is selected to point out the effect of the original bends that are interpreted as control points. (Observe also, that the selection indication for a bezier spline curve applies special rendering to present the control points.)

**Figure 6.13. Edge represented by BezierEdgeRealizer**



## Class SplineEdgeRealizer

An EdgeRealizer implementation where the associated edge is drawn as a cubic spline curve. The edge's bends are interpreted as control points for the curve, which, by construction, are all part of the curve path.

[Figure 6.14, “Edge represented by SplineEdgeRealizer” \[175\]](#) shows an example of a cubic spline curve rendered by class `SplineEdgeRealizer`. The edge is selected to point out the effect of the original bends that are interpreted as control points.

**Figure 6.14. Edge represented by SplineEdgeRealizer**



## Class QuadCurveEdgeRealizer

An EdgeRealizer implementation where the associated edge is drawn as a quadratic spline curve. The edge's bends are interpreted as control points for the curve, which, by construction, are not part of the curve path.

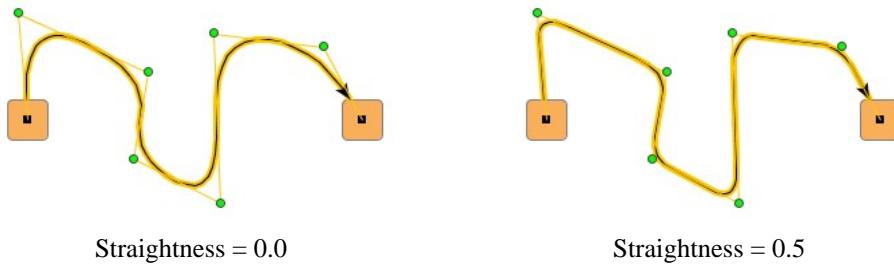
Class [QuadCurveEdgeRealizer](#) supports a straightness factor to customize the extent at which the curve follows the given straight line path between control points. Having a range of 0.0 to 1.0, higher straightness factor means closer matching of curve and given straight line path. [API Excerpt 6.22, “Straightness methods from class QuadCurveEdgeRealizer” \[176\]](#) lists the methods to control the straightness settings for a quadratic spline curve.

### API Excerpt 6.22. Straightness methods from class QuadCurveEdgeRealizer

```
// Getter and setter methods.
double getStraightness()
void setStraightness(double straightness)
```

[Figure 6.15, “Edges represented by QuadCurveEdgeRealizer” \[176\]](#) demonstrates two possible quadratic spline curves rendered by class [QuadCurveEdgeRealizer](#). The edges are selected to point out the effect of the original bends that are interpreted as control points. (Observe also, that the selection indication for a quadratic spline curve applies special rendering to present the control points.)

**Figure 6.15. Edges represented by QuadCurveEdgeRealizer**



## Class GenericEdgeRealizer

This EdgeRealizer implementation primarily serves as a generic basis for edge realizer customization. It defines a set of static inner interfaces which allow fine-grained control over all aspects of an edge realizer's visual representation and behavior. Instead of applying subclassing and overriding, custom logic for a specialized realizer can be conveniently expressed by providing implementations for only a selection or for all of these interfaces.

Class [GenericEdgeRealizer](#) offers all methods from its superclass [EdgeRealizer](#), the actual work, though, is delegated to the following interfaces:

- [GenericEdgeRealizer.Painter](#) to render the edge's visual representation
- [GenericEdgeRealizer.BendHandler](#) to handle the bends of the edge
- [GenericEdgeRealizer.BendPainter](#) to render the visual representation of an edge's bends
- [GenericEdgeRealizer.PortPainter](#) to render the visual representation of an edge's ports
- [GenericEdgeRealizer.PathCalculator](#) to calculate an edge's path
- [GenericEdgeRealizer.ContainsTest](#) to perform hit tests
- [GenericEdgeRealizer.IntersectionTest](#) to perform intersection testing
- [GenericEdgeRealizer.UnionRectCalculator](#) to calculate the edge's bounds
- [GenericEdgeRealizer.GenericMouseInputEditorProvider](#) to provide [MouseInputEditor](#) instances for mouse-based interaction
- [GenericEdgeRealizer.UserDataHandler](#) to govern copying and serializing of user-defined data

Together, a set of actual implementations for these interfaces forms a so-called "configuration" that defines the look and feel of one edge type. Different configurations can be used to define a variety of edge types.

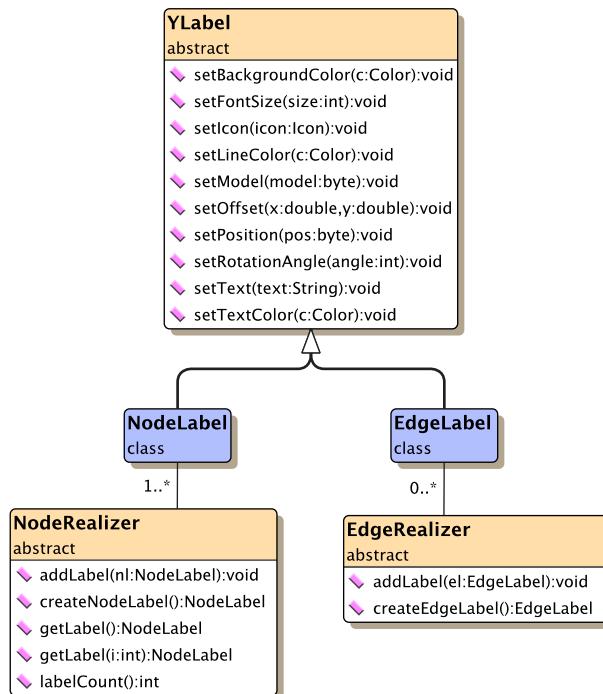
Out of the box, this edge realizer uses its default configuration, which results in a simple polyline edge representation similar to that of [PolyLineEdgeRealizer](#) as shown above. To take full advantage of the concept that is behind class `GenericEdgeRealizer`, custom configurations must be defined and registered. The section [Writing Customized Realizers](#) discusses the necessary steps.

# Realizer-Related Features

## Label Support

The yFiles label capabilities cover a variety of standard features common with today's word processing systems. A label can have multiple lines, the label's text can be aligned in several ways, and font attributes like type, size, and color can be controlled. Also, using an instance of type `java.awt.Insets`, a label's text can be placed with custom margins within its bounding box. Other graphical attributes, including background and border color, can be set, too.

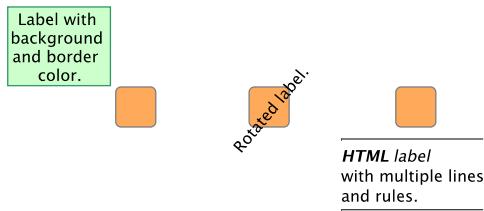
**Figure 6.16. Hierarchy of the yFiles label classes**



Look and feel of both node labels and edge labels are defined by so-called "configurations," i.e., sets of interface implementations that cover all aspects of label functionality. Classes `NodeLabel` and `EdgeLabel` each make available two predefined configurations, the default label configuration, which is encapsulated in class `DefaultLabelConfiguration`, and a second configuration that provides automatic line-wrapping of a label's text. The latter one is accessible with these classes using the name "CroppingLabel".

In addition to text alignment and font attributes, the default label configuration provides further useful features. On the basis of the used font the label's size is automatically determined each time a new text is set, respectively one of the label's properties changes. Instead of text, or in addition to its text, a label can also show an icon, and for complete freedom in design it can even render HTML code. Furthermore, the entire label can be rotated about its center. [Figure 6.17, "Some features of the default label configuration"](#) [178] depicts some label variations.

**Figure 6.17. Some features of the default label configuration**



HTML rendering is enabled by default and is triggered as soon as the label's text starts with <html>. Note that the rendering capabilities are constrained to the basic HTML support provided by Swing. [Example 6.2, “Creating some labels” \[179\]](#) shows code to generate various kinds of labels.

### Example 6.2. Creating some labels

```
// 'nr1' to 'nr3' are of type y.view.NodeRealizer.

NodeLabel nl;
// First node.
nl = nr1.createNodeLabel();
nr1.setLabel(nl);

// Set the label model and position.
nl.setModel(NodeLabel.CORNERS);
nl.setPosition(NodeLabel.NW);

// Note the line breaks.
nl.setText("Label with \nbackground \nand border \nncolor.");
nl.setBackgroundColor(new Color(204, 255, 204));
nl.setLineColor(new Color(51, 153, 102));

// Second node.
nl = nr2.createNodeLabel();
nr2.setLabel(nl);

nl.setText("Rotated label.");
nl.setRotationAngle(310);

// Third node.
nl = nr3.createNodeLabel();
nr3.setLabel(nl);

// Set the label model and position.
nl.setModel(NodeLabel.SANDWICH);
nl.setPosition(NodeLabel.S);

// HTML rendering is on by default.
nl.setText("<html><hr>" +
"<em><b>HTML</b> label</em><br>" +
"with multiple lines<br>" +
"and rules." +
"<hr></html>");
```

Figure 6.18, “Label with icon instead of text” [179] is created by the code shown in [Example 6.3, “Creating a label with an icon” \[179\]](#)

### Figure 6.18. Label with icon instead of text



### Example 6.3. Creating a label with an icon

```
// 'nr' is of type y.view.NodeRealizer.
NodeLabel nl = nr.createNodeLabel();
nr.setLabel(nl);

// Set the label model and position.
nl.setModel(NodeLabel.SANDWICH);
nl.setPosition(NodeLabel.N);

// Well-known icon that comes in handy as an example...
nl.setIcon(MetalIconFactory.getFileChooserHomeFolderIcon());
```

### Tip

Using an image icon as the content of a node label permits a node that is rendered by a `ShapeNodeRealizer` to show both shape and image. In contrast to using an `ImageNodeRealizer`, however, the image will retain its size, i.e., it will not scale with the node.

Using the additional line of code from [Example 6.4, “Setting custom margins” \[180\]](#) the label's bounding box is effectively enlarged to create the specified margins around the text. See [Figure 6.19, “Label text with custom margins” \[180\]](#) for the result.

### Example 6.4. Setting custom margins

```
// Increase the left margin.
nl.setInsets(new Insets(10, 70, 10, 10));
```

**Figure 6.19. Label text with custom margins**



Using the "CroppingLabel" configuration, which enables a label's text to be automatically line-wrapped, it is easily possible to have a label's visual presentation truncated to fit into the node's size, for example. [Figure 6.20, “Label text line-wrapping and cropping” \[180\]](#) shows both the default behavior when a node label has a long text, and the results when the text is automatically line-wrapped to fit either the width or the width and height of its node. In the latter case, when the line-wrapped label text does not fit, the displayed text is truncated and an ellipsis character replaces the part that is not shown.

**Figure 6.20. Label text line-wrapping and cropping**



[Example 6.5, “Different sizing policies for the label text” \[181\]](#) shows the code that sets up the auto-sizing policies depicted in [Figure 6.20, “Label text line-wrapping and cropping” \[180\]](#).

### Note

The `CroppingLabel` configuration currently has no support for HTML rendering or

setting an icon with a label.

### Example 6.5. Different sizing policies for the label text

```
// 'graph' is of type y.view.Graph2D.

Set configurations = NodeLabel.getFactory().getAvailableConfigurations();
for (int i = 0; i < 3; i++) {
    Node n = graph.createNode(200 * (i + 1), 100);
    NodeLabel nl = graph.getRealizer(n).getLabel();

    nl.setFontName("Tahoma");
    nl.setFontSize(14);
    nl.setAlignment(YLabel.ALIGN_LEFT);
    nl.setText("The quick brown fox jumps over the lazy dog.");

    switch (i) {
        case 1:
            if (configurations.contains("CroppingLabel")) {
                nl.setConfiguration("CroppingLabel");
                nl.setAutoSizePolicy(NodeLabel.AUTOSIZE_NODE_WIDTH);
            }
            break;
        case 2:
            if (configurations.contains("CroppingLabel")) {
                nl.setConfiguration("CroppingLabel");
                nl.setAutoSizePolicy(YLabel.AUTOSIZE_NONE);
                nl.setContentSize(60, 60);
            }
            break;
        default:
            nl.setAutoSizePolicy(YLabel.AUTOSIZE_CONTENT);
    }
}
```

### Customizing Label Behavior

Abstract class `YLabel` defines a set of inner interfaces which allow fine-grained control over all aspects of a label's visual representation and behavior. Custom label logic can be conveniently expressed by providing implementations for only a selection or for all of these interfaces:

- `YLabel.Painter` renders the label's visual representation and provides the locations for the label's icon and text
- `YLabel.Layout` calculates the actual size of a label's content and provides hit-testing support
- `YLabel.BoundsProvider` is used to enlarge a given rectangle so that the label's bounds is fully contained
- `YLabel.UserDataHandler` governs both copying and serializing of user-defined data

Together, a set of actual implementations for these interfaces forms a so-called "configuration" that defines the look and feel of a label. Different configurations can be used to define a variety of label types.

Management of label configurations is done by static inner class `YLabel.Factory`. A reference to this class can be get using the `NodeLabel` and `EdgeLabel` class method shown in [API Excerpt 6.23, “Static method to get the factory for configuration management” \[181\]](#)

### API Excerpt 6.23. Static method to get the factory for configuration management

```
// Getter method in classes NodeLabel and EdgeLabel.
static YLabel.Factory getFactory()
```

To handle user-defined data, `YLabel` defines the `YLabel.UserDataHandler` inner interface. This interface is used to delegate all work that relates to copying or (de)serializing of all user-defined data of a label. Class `SimpleUserDataHandler` can be used as a default implementation for this interface. It is capable of dealing with arbitrary user-defined data objects that implement interfaces `java.lang.Cloneable` (for copying) and `java.io.Serializable` (for serialization and deserialization).

Class `DefaultLabelConfiguration` serves as a default implementation for all `YLabel` interfaces. It provides the default label behavior and can be used as a convenient base for customized configurations. [Example 6.6, “Customizing the default node label configuration” \[182\]](#) shows how to create a specialized node label configuration using another `YLabel.Painter` implementation.

### Example 6.6. Customizing the default node label configuration

```
// Get the factory to register custom styles/configurations.
YLabel.Factory factory = NodeLabel.getFactory();

// Retrieve a map that holds the default NodeLabel configuration.
// The implementations contained therein can be replaced one by one in order to
// create custom configurations...
Map implementationsMap = factory.createDefaultConfigurationMap();

// Customize the painting with a custom YLabel.Painter implementation.
implementationsMap.put(YLabel.Painter.class, new MyPainter());

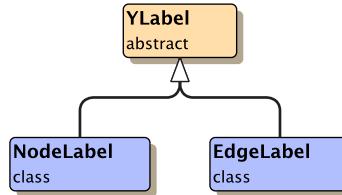
// Add this configuration to the factory.
factory.addConfiguration("Bubble", implementationsMap);
```

Tutorial demo application [YLabelConfigurationDemo.java](#) presents in detail how a custom node label configuration is created.

## Label Models

The aforementioned general features are offered by abstract class `YLabel`. Classes `NodeLabel` and `EdgeLabel` provide further label functionality that is specific to the respective type of graph element. [Figure 6.21, “Label classes hierarchy” \[182\]](#) presents the class hierarchy for the label functionality from package `y.view`.

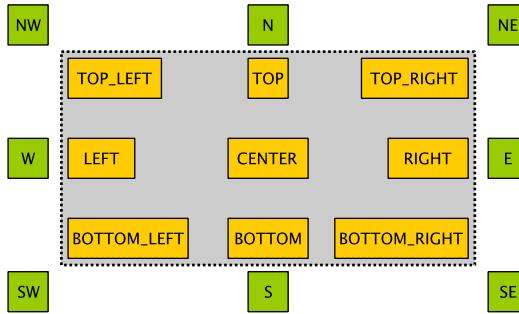
**Figure 6.21. Label classes hierarchy**



A label naturally belongs to a specific graph element, and accordingly should be presented near this element. To control the label's position relative to its element, so-called "label models" are used.

Class `NodeLabel` provides a number of label models with predefined positions and additionally a "free" model, where the label's position is solely defined by an offset. [Figure 6.22, “Possible node label positions” \[182\]](#) depicts all predefined positions for a node label.

**Figure 6.22. Possible node label positions**



It is important to note that the set of valid positions varies with the chosen model, i.e., not all possible positions are valid for each model. [Table 6.4, “Valid label positions for each model” \[?\]](#) lists the sets of valid positions for each node label model.

**Table 6.4. Valid label positions for each model**

Model Name	Valid Positions
INTERNAL	TOP_LEFT, TOP, TOP_RIGHT, LEFT, CENTER, RIGHT, BOTTOM_LEFT, BOTTOM, BOTTOM_RIGHT
CORNERS	NW, NE, SW, SE
SIDES	N, W, E, S
EIGHT_POS	All positions from CORNERS and also all from SIDES.
SANDWICH	N, S
FREE	ANYWHERE

The offset with the "free" label model is the distance vector between the upper left corners of node and label. Setting a label model and choosing one of the predefined label positions is demonstrated in [Example 6.7, “Setting a node label model” \[183\]](#). If the chosen position is not in the set of valid positions for a label model, then a default position is taken instead.

### Example 6.7. Setting a node label model

```
// 'nr' is of type y.view.NodeRealizer.

// Create the label and add it to the node realizer.
NodeLabel nl = nr.createNodeLabel();
nr.setLabel(nl);

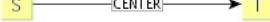
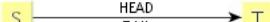
// Set a node label model and choose one of the predefined label positions.
// 'CORNERS' has four positions at the corners of a node, outside its bounds.
nl.setModel(NodeLabel.CORNERS);
// 'SE' means the lower right corner.
nl.setPosition(NodeLabel.SE);
```

Observe that the sets of valid positions with label models CORNERS, SANDWICH, and SIDES are proper subsets of those allowed with label model EIGHT\_POS. Choosing one of these subsets explicitly restricts the number of available label positions, i.e., for the most flexibility EIGHT\_POS is the best choice.

Class `EdgeLabel` provides several label models with predefined positions, and some "slider" models where the set of valid positions is dynamic and varies with the length of the edge path. Additionally, there is also a "free" model available, where the label's position is defined by an offset. [Table 6.5, “Edge label models” \[183\]](#) lists all edge label models and shows the predefined label positions, the dynamic positions of the "slider" models, and also examples for offset positions with the "free" model.

**Table 6.5. Edge label models**

## Related Classes

Model Name	Valid Positions
CENTERED	CENTER
	
TWO_POS	HEAD, TAIL
	
THREE_CENTER	SCENTR, CENTER, TCENTR
	
SIX_POS	SHEAD, HEAD, THEAD, STAIL, TAIL, TTAIL
	
CENTER_SLIDER	DYNAMIC
	
SIDE_SLIDER	DYNAMIC
	
FREE	ANYWHERE
	

The dynamic positions of the "slider" models are specified by a ratio value that ranges from 0 . 0 to 1 . 0. It determines the label's placement along the edge path, with increasing value meaning positions farther from the source node. The offset with the "free" label model is the distance vector between the intersection point of the first edge segment with the source node and the upper left corner of the label. Setting a label model and choosing a label position is demonstrated in [Example 6.8, “Setting an edge label model” \[184\]](#). If the chosen position is not in the set of valid positions for a label model, then a default position is taken instead.

### Example 6.8. Setting an edge label model

```
// 'er' is of type y.view.EdgeRealizer.

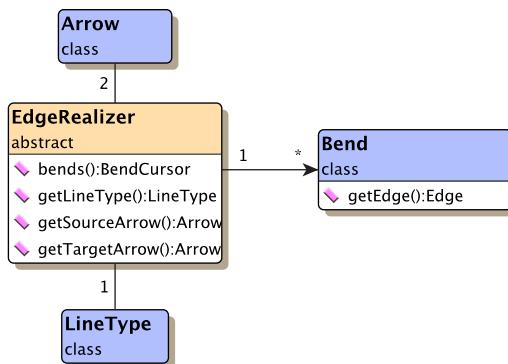
// Create the label and add it to the edge realizer.
EdgeLabel el = er.createEdgeLabel();
er.addLabel(el);

// Set an edge label model. 'CENTER_SLIDER' has a dynamic number of label
// positions that lie on the edge's path.
el.setModel(EdgeLabel.CENTER_SLIDER);
el.setPosition(EdgeLabel.DYNAMIC);
// Choose a label position near the source node.
el.setRatio(0.3);
```

## Related Classes

In addition to the EdgeRealizer object associated there are further classes involved in rendering an edge's visual representation. For each of the edge's bends the EdgeRealizer object holds a corresponding object of type **Bend**. The special graphical decoration that can be placed at either end of an edge is governed by **class Arrow**. And the line graphic of the edge itself can be controlled using **class LineType**.

**Figure 6.23.** EdgeRealizer-related classes



[API Excerpt 6.24, “Arrow-related methods from class EdgeRealizer” \[185\]](#) shows the methods provided by class EdgeRealizer to handle arrows.

### API Excerpt 6.24. Arrow-related methods from class EdgeRealizer

```

// Getter methods.
Arrow getArrow()
Arrow getSourceArrow()
Arrow getTargetArrow()

// Setter methods.
void setArrow(Arrow arrow)
void setSourceArrow(Arrow arrow)
void setTargetArrow(Arrow arrow)
  
```

## Class Arrow

Class **Arrow** renders the graphical decorations that can be placed at an edge's ends. Normally, this decoration is some kind of arrowhead that is used to indicate the edge's direction. [Table 6.6, “Arrow shapes of class Arrow” \[185\]](#) depicts the predefined arrow shapes.

Note that omitting the decoration is done using another arrow shape. Arrow type **NONE**, which is not shown below, "renders" an empty arrow shape.

**Table 6.6. Arrow shapes of class Arrow**

Arrow Type	Arrow Shape
DELTA	
DIAMOND	
SHORT	

## Related Classes

Arrow Type	Arrow Shape
STANDARD	
WHITE_DELTA	
WHITE_DIAMOND	

Class Arrow supports fine-grained control of arrowhead rendering. It is possible to specify an arrowhead's distance to the border of a node, and also the position of the edge path's end relative to an arrowhead's tip.

In addition to the predefined arrow shapes it is also possible to register custom ones using the methods listed in [API Excerpt 6.25, “Support for custom arrow shapes” \[186\]](#). The most freedom defining a customized arrow(head) is provided when using graphical objects that implement interface [Drawable](#).

### API Excerpt 6.25. Support for custom arrow shapes

```
// Getter and setter methods for custom arrow shapes.
static Arrow addCustomArrow(String name, Drawable drawable)
static Arrow addCustomArrow(String name, Shape shape, Color fillColor)
static Arrow addCustomArrow(String name, Shape shape, Color fillColor,
                           Stroke lineStroke, Color lineColor)
static Arrow getCustomArrow(String name)

// Arrow shapes using customized distance and arrow length.
static Arrow addCustomArrow(String name, Drawable drawable,
                           double arrowLength, double clipLength)
static Arrow addCustomArrow(String name, Shape shape, Color fillColor,
                           Stroke lineStroke, Color lineColor,
                           double arrowLength, double clipLength)

String getCustomName()
```

## Class Bend

Class [Bend](#) is to bends (also known as control points) what classes [NodeRealizer](#) and [EdgeRealizer](#) are to nodes and edges. It handles all user interface (UI)-related aspects, i.e., it renders a bend's graphical representation, and also provides a number of features including, e.g., control of the selection state and support for hit tests.

Note that bends, i.e., control points, determine the path of an edge, but are not necessarily part of the path itself. Especially the classes [BezierEdgeRealizer](#) and [SplineEdgeRealizer](#) exhibit this difference.

The most defining characteristic of a bend is its location. [API Excerpt 6.26, “Positional methods from class Bend” \[186\]](#) lists the methods that can be used to control it.

### API Excerpt 6.26. Positional methods from class Bend

```
// Getter methods.
double getX()
double getY()

// Setter methods.
void setLocation(double newX, double newY)
void moveBy(double dx, double dy)
```

## Class LineType

Class [LineType](#) is responsible for rendering the line graphic of an edge. It is used for both polyline edges as well as curves. [Table 6.7, “Line types of class LineType” \[187\]](#) shows the predefined

line types. Note that the line types are available in different widths.

**Table 6.7. Line types of class LineType**

Constant Name	Line Type
DASHED_STYLE	---
DOTTED_STYLE	.....
LINE_STYLE	---

Different line types can be get directly using their respective names, e.g., DASHED\_1, DOTTED\_2, LINE\_4, etc. Alternatively, LineType also offers the static methods from [API Excerpt 6.27, “Static getter methods from class LineType” \[187\]](#) to get (one of) the available line types.

### API Excerpt 6.27. Static getter methods from class LineType

```
static Vector availableLineTypes()
static LineType getLineType(int width, byte style)
```

In addition to the predefined line types it is also possible to create custom ones using the method from [API Excerpt 6.28, “Support for creation of custom line types” \[187\]](#)

### API Excerpt 6.28. Support for creation of custom line types

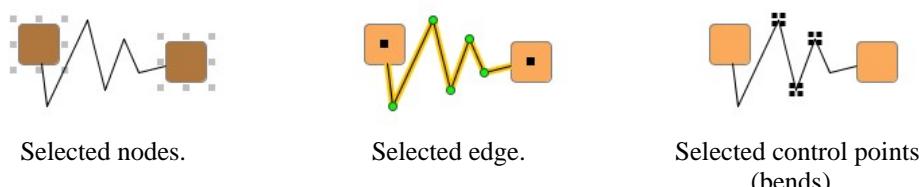
```
static LineType createLineType(float width,
                               int cap, int join,
                               float miterlimit,
                               float[] dash, float dashPhase)
```

## Selection State

The indication whether a graph element is selected or not contributes a great deal to the user experience and is of central importance to the notion of a user interacting with the graph structure by means of a graphical user interface.

The predefined NodeRealizer and EdgeRealizer implementations define special rendering logic for the selection indication of its associated graph elements. A selected node, e.g., also shows eight small knobs around its outline that can be used to change the node's width and height. [Figure 6.24, “Selection indication for nodes, edges, and bends” \[187\]](#) shows the default results for selected nodes, a selected edge, and selected bends. Note that bends have different appearance depending on the context.

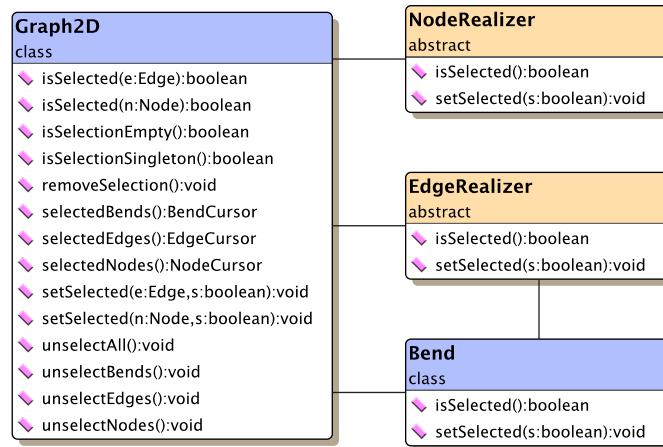
**Figure 6.24. Selection indication for nodes, edges, and bends**



Both realizer types offer getter and setter methods to control the selection state of their associated graph element. Likewise, similar methods in class Bend can be used to control the selection

state of an edge's control points. Class Graph2D has a variety of additional convenience methods to change, respectively query, the selection state of arbitrary collections of graph elements. [Figure 6.25, “Methods for modifying the selection state” \[188\]](#) presents an overview of the classes and their methods dealing with selection state. Further, more sophisticated methods to alter the selection state are offered by abstract class Selections.

**Figure 6.25. Methods for modifying the selection state**



Changes in the selection state of graph elements can easily be listened to using the familiar events and listener concept. Interface [Graph2DSelectionListener](#) conveys an event of type [Graph2DSelectionEvent](#) which can be queried for specific details. See the section called “Events and Listeners” [\[221\]](#) for further description.

# Writing Customized Realizers

The predefined node realizer and edge realizer implementations of package `y.view` already provide advanced out-of-the-box functionality (please see the sections on [node realizers](#) and [edge realizers](#), respectively). To modify specific aspects of either visual representation or behavior of a realizer, an available realizer implementation can be used as the starting point for customization.

Both node realizer and edge realizer implementations support two different customization schemes:

- customization by [subclassing and overriding](#)
- customization by [providing interface implementations](#)

The latter customization scheme is applied by classes `GenericNodeRealizer` and `GenericEdgeRealizer`, while the former is applied by all remaining node realizer and edge realizer implementations.

## Subclassing and Overriding

Creating a custom variant of either node realizer or edge realizer is easily accomplished by extending one of:

- `ShapeNodeRealizer`
- `ImageNodeRealizer`

(for a node realizer specialization) or one of:

- `PolyLineEdgeRealizer`
- `ArcEdgeRealizer`
- `BezierEdgeRealizer`
- `SplineEdgeRealizer`
- `QuadCurveEdgeRealizer`

(for an edge realizer specialization).

A specialized node realizer, for example, defines a new `paintNode` implementation, maybe also a new `calcUnionRect` implementation. In case there is a new shape created in a `ShapeNodeRealizer` descendant, the methods `setSize` and `setLocation` eventually would have to be changed, too.

Generally, to have a customized realizer behave properly, its responsibilities have to be retained. These include:

- [realizer replication](#)
- [bounds calculation](#)
- [YGF serialization and deserialization](#) (if the customized realizer should be written to and read from YGF file format)

## Realizer Replication

Replication means the mechanism that is utilized when a new realizer instance for a graph element in a `Graph2D` object is created. The default realizer types registered with the `Graph2D` object serve as factories for realizer creation, i.e., they create copies of themselves via the methods shown in [API Excerpt 6.29, “Realizer methods for replication”](#) [189]

### API Excerpt 6.29. Realizer methods for replication

```
// NodeRealizer replication methods.  
NodeRealizer createCopy()  
abstract NodeRealizer createCopy(NodeRealizer nr)  
  
// EdgeRealizer replication methods.
```

```
EdgeRealizer createCopy()
abstract EdgeRealizer createCopy(EdgeRealizer er)
```

Proper replication behavior for a realizer implementation can easily be achieved using, e.g., a copy constructor as demonstrated in [Example 6.9, “Sample replication code” \[190\]](#)

### Example 6.9. Sample replication code

```
class MyCustomNodeRealizer extends ShapeNodeRealizer
{
    // Custom attribute.
    private int myFancyAttribute;

    // Copy constructor.
    public MyCustomNodeRealizer(NodeRealizer nr)
    {
        super(nr);
        // If the given node realizer is of this type, then apply copy semantics.
        if (nr instanceof MyCustomNodeRealizer)
        {
            MyCustomNodeRealizer mcnr = (MyCustomNodeRealizer)nr;
            // Copy the values of custom attributes.
            myFancyAttribute = mcnr.myFancyAttribute;
        }
    }

    public NodeRealizer createCopy(NodeRealizer nr)
    {
        // Return a new node realizer that has specific type.
        return new MyCustomNodeRealizer(nr);
    }
}
```

## Bounds Calculation

The bounds calculation of a realizer plays an important role for the graphical user interface. Among other things, it is necessary for:

- determining those parts of a view that have to be repainted
- calculating the correct bounding box of an entire graph so that all elements are included
- hit-testing a graph element

Hit-testing a node, for example, also relates to determining the intersection points of all its connecting edges. The methods from class `EdgeRealizer` that return an edge's intersection points in turn invoke the `contains` method of both its source and target node's `NodeRealizer`. The correct edge clipping calculation thus relies on the proper implementation of the node realizer's hit-testing method.

[Example 6.10, “Customized ShapeNodeRealizer” \[190\]](#) presents a customized node realizer that draws an additional shadow underneath the used shape. Since the shadow leads to an increase in both width and height of the resulting graphic, the bounds calculation as performed by method `calcUnionRect` is modified to accommodate that fact. Also, the node's resize knobs (a.k.a. "hot spots") are relocated, and their hit-testing logic is changed, too.

Note, however, that this implementation does not change the general hit-testing logic from its superclass, i.e., the shadow is not relevant for any mouse events.

### Example 6.10. Customized ShapeNodeRealizer

```
class DropShadowedShapeNodeRealizer extends ShapeNodeRealizer
{
    public void paintNode(Graphics2D gfx){
        gfx.translate(4, 4);
        gfx.setColor(Color.darkGray);
        gfx.fill(shape);

        gfx.translate(-4, -4);
        // Invoke the superclass's paint method.
        super.paintNode(gfx);
    }
}
```

```
// Enlarge the realizer's dimensions to include the shadow.
public void calcUnionRect(Rectangle2D rect){
    super.calcUnionRect(rect);
    rect.add(x + width + 4, y + height + 4);
}

// Paint the resize knobs outside the shadowed area.
public void paintHotspots(Graphics2D gfx){
    width += 4; height += 4;
    super.paintHotspots(gfx);
    width -= 4; height -= 4;
}

public byte hotspotHit(double hx, double hy){
    width += 4; height += 4;
    byte hit = super.hotspotHit(hx, hy);
    width -= 4; height -= 4;

    return hit;
}
}
```

## YGF Serialization and Deserialization

Realizer (de)serialization is important when the respective class should be written to/read from a file in Y Graph Format (YGF). The methods that provide the (de)serialization support are listed in [API Excerpt 6.30, “Realizer methods for YGF \(de\)serialization” \[191\]](#)

### API Excerpt 6.30. Realizer methods for YGF (de)serialization

```
// YGF (de)serialization methods.
void read(ObjectInputStream in)
void write(ObjectOutputStream out)
```

Links to tutorial demo code that shows customized realizer implementations and examples for proper realizer (de)serialization can be found in [the section called “Tutorial Demo Code” \[258\]](#)

## Providing Interface Implementations

Classes `GenericNodeRealizer` and `GenericEdgeRealizer` each define an appropriate set of static inner interfaces which allow fine-grained control over all aspects of a node's or an edge's visual representation and behavior. Custom logic for a specialized realizer can be conveniently expressed by providing implementations for only a selection or for all of these interfaces.

Together, a set of actual implementations for these interfaces forms a so-called "configuration" that defines the look and feel of one node type or edge type, respectively. Different configurations can be used to define a variety of types.

Management of realizer configurations is done by static inner classes `GenericNodeRealizer.Factory` and `GenericEdgeRealizer.Factory`. A reference to either of these classes can be get using the class methods listed in [API Excerpt 6.31, “Static methods to get the factory for configuration management” \[191\]](#)

### API Excerpt 6.31. Static methods to get the factory for configuration management

```
// Getter method of class GenericNodeRealizer.
static GenericNodeRealizer.Factory getFactory()

// Getter method of class GenericEdgeRealizer.
static GenericEdgeRealizer.Factory getFactory()
```

In contrast to realizer customization using [subclassing and overriding](#), both `GenericNodeRealizer` and `GenericEdgeRealizer` already support proper realizer replication and YGF (de)serialization. In addition, support for (de)serialization from/to GML and XGML file formats is also supported.

To handle user-defined data, the generic realizer classes define static inner interfaces `GenericNodeRealizer.UserDataHandler` and `GenericEdgeRealizer.UserDataHandler`. These interfaces

are used to delegate all work that relates to copying or (de)serializing of all user-defined data of a realizer object. Class `SimpleUserDataHandler` can be used as a default implementation for both these interfaces. It is capable of dealing with arbitrary user-defined data objects that implement interfaces `java.lang.Cloneable` (for copying) and `java.io.Serializable` (for serialization and deserialization).

Static inner interfaces `GenericNodeRealizer.GenericMouseInputEditorProvider` and `GenericEdgeRealizer.GenericMouseInputEditorProvider` serve as entry points for custom `MouseInputEditor` implementations that can be associated with configurations for either kind of generic realizer. See also the section called “Interface `MouseInputEditor`” [218]

### Class `GenericNodeRealizer`

`GenericNodeRealizer` offers all methods from its superclass `NodeRealizer`, the actual work, though, is delegated to a number of static inner interfaces, namely:

- `GenericNodeRealizer.Painter` to render the node's visual representation
- `GenericNodeRealizer.HotSpotPainter` to add the resize knobs whenever a node is selected
- `GenericNodeRealizer.HotSpotHitTest` to handle hit testing on the node's resize knobs
- `GenericNodeRealizer.ContainsTest` to perform hit tests
- `GenericNodeRealizer.IntersectionTest` to perform intersection testing
- `GenericNodeRealizer.UnionRectCalculator` to calculate the node's bounds
- `GenericNodeRealizer.BoundsChangedHandler` to handle changes of the node's bounds
- `GenericNodeRealizer.PortCandidateListProvider` to provide coordinates for possible port locations
- `GenericNodeRealizer.GenericMouseInputEditorProvider` to provide `MouseInputEditor` instances which handle mouse-based interaction
- `GenericNodeRealizer.UserDataHandler` to govern both copying and serializing of user-defined data

Abstract classes `AbstractCustomNodePainter` and `AbstractCustomHotSpotPainter` serve as convenient base implementations for interfaces `GenericNodeRealizer.Painter` and `GenericNodeRealizer.HotSpotPainter`/`GenericNodeRealizer.HotSpotHitTest`, respectively.

Additionally, class `NodeCellRendererPainter` is a specialized implementation for interface `GenericNodeRealizer.Painter` that provides support for using Swing UI components for the visual representation of node realizers.

[Example 6.11, “Creating a `GenericNodeRealizer` configuration” \[192\]](#) shows how to create a configuration for a node type that renders an ellipse.

### Example 6.11. Creating a `GenericNodeRealizer` configuration

```
String createEllipseConfiguration(Map implementationsMap)
{
    // Create and register an implementation for interfaces
    // GenericNodeRealizer.Painter and GenericNodeRealizer.ContainsTest.
    // (This implementation renders and hit-tests an ellipse.)
    MyCustomPainter painter = new MyCustomPainter(new Ellipse2D.Double());
    implementationsMap.put(GenericNodeRealizer.Painter.class, painter);
    implementationsMap.put(GenericNodeRealizer.ContainsTest.class, painter);

    // Create and register an implementation for interfaces
    // GenericNodeRealizer.HotSpotPainter and GenericNodeRealizer.HotSpotHitTest.
    // ('165' is the decimal value for a bit mask that yields hot spots at the
    // corners of the bounding box of the ellipse.)
    MyCustomHotSpotPainter chsp =
        new MyCustomHotSpotPainter(165, new Ellipse2D.Double(), null);
    implementationsMap.put(GenericNodeRealizer.HotSpotPainter.class, chsp);
    implementationsMap.put(GenericNodeRealizer.HotSpotHitTest.class, chsp);

    return "Ellipse";
}
```

### Note

If a configuration lacks implementations for any of the aforementioned interfaces, then appropriate default implementations from class `NodeRealizer` are invoked.

[Example 6.12, “Adding a GenericNodeRealizer configuration” \[193\]](#) demonstrates the steps to add a new node realizer configuration to the factory that manages the available configurations. Note that the factory can be conveniently used to create a map holding the default configuration for `GenericNodeRealizer`.

### Example 6.12. Adding a GenericNodeRealizer configuration

```
void addConfiguration()
{
    // Get the factory to register different configurations/styles.
    GenericNodeRealizer.Factory factory = GenericNodeRealizer.getFactory();

    // Retrieve a map that holds the default GenericNodeRealizer configuration.
    // The implementations contained therein can be replaced one by one in order
    // to create custom configurations...
    Map implementationsMap = factory.createDefaultConfigurationMap();

    // Create and store a configuration in the map.
    String desc = createEllipseConfiguration(implementationsMap);
    // Add the configuration to the factory.
    factory.addConfiguration(desc, implementationsMap);
}
```

Finally, [Example 6.13, “Using a specific GenericNodeRealizer configuration” \[193\]](#) shows how a specific configuration is used with `GenericNodeRealizer`.

### Example 6.13. Using a specific GenericNodeRealizer configuration

```
void setDefaultNodeRealizerForView(Graph2DView view)
{
    // Add a configuration to the pool of GenericNodeRealizer configurations.
    addConfiguration();

    // Create a node realizer.
    GenericNodeRealizer gnr = new GenericNodeRealizer();
    // Initialize this node realizer to apply the configuration just added...
    gnr.setConfiguration("Ellipse");

    // Set the node realizer as the default node realizer for the given view.
    view.getGraph2D().setDefaultNodeRealizer(gnr);
}
```

### Note

If `GenericNodeRealizer` is used with an empty configuration, then appropriate methods from class `NodeRealizer` are invoked, which results in a simple rectangular node representation.

The tutorial demo application [GenericNodeRealizerDemo.java](#) presents the customization of `GenericNodeRealizer` in detail.

### Class GenericEdgeRealizer

`GenericEdgeRealizer` offers all methods from its superclass `EdgeRealizer`, the actual work, though, is delegated to a number of static inner interfaces, namely:

- [GenericEdgeRealizer.Painter](#) to render the edge's visual representation
- [GenericEdgeRealizer.BendHandler](#) to handle the bends of the edge
- [GenericEdgeRealizer.BendPainter](#) to render the visual representation of an edge's bends
- [GenericEdgeRealizer.PortPainter](#) to render the visual representation of an edge's ports
- [GenericEdgeRealizer.PathCalculator](#) to calculate an edge's path
- [GenericEdgeRealizer.ContainsTest](#) to perform hit tests
- [GenericEdgeRealizer.IntersectionTest](#) to perform intersection testing

- `GenericEdgeRealizer.UnionRectCalculator` to calculate the edge's bounds
- `GenericEdgeRealizer.GenericMouseInputEditorProvider` to provide `MouseInputEditor` instances for mouse-based interaction
- `GenericEdgeRealizer.UserDataHandler` to govern both copying and serializing of user-defined data

Classes `SimpleBendHandler` and `PolyLinePathCalculator` serve as default implementations for interfaces `GenericEdgeRealizer.BendHandler` and `GenericEdgeRealizer.PathCalculator`, respectively. They are also part of the default configuration for `GenericEdgeRealizer` which renders a simple poly-line edge path.

Additionally, class `GenericEdgePainter` serves as a convenient base implementation for interface `GenericEdgeRealizer.Painter` that can be used for further customization.

### Important

An actual edge type configuration must always provide implementations for static inner interfaces `GenericEdgeRealizer.BendHandler` and `GenericEdgeRealizer.PathCalculator` to form a valid configuration.

The tutorial demo application `GenericEdgeRealizerDemo.java` presents the customization of `GenericEdgeRealizer` in detail.

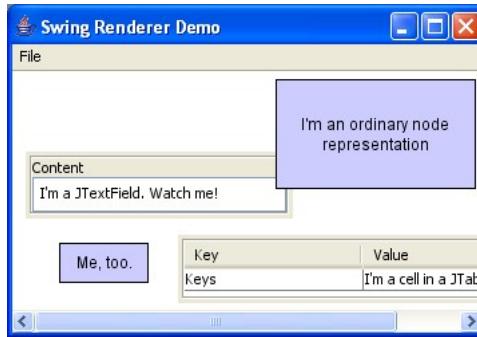
## Swing User Interface Components as Node Realizers

The yFiles library enables the use of Java Swing User Interface (UI) components to render nodes and to create proper editors for node-related data which are presented as "live" objects within a Graph2DView. Similar to the concept and use of Java Swing interfaces

- TableCellRenderer and TableCellEditor or
- TreeCellRenderer and TreeCellEditor

both visual representation of nodes and editing of related data can be accomplished by means of implementations for interfaces [NodeCellRenderer](#) and [NodeCellEditor](#), respectively. Each interface provides a callback method to return a JComponent which presents Swing look and feel. [Figure 6.26, “JComponents in the view” \[195\]](#) shows different JComponents within a view at zoom level 100%.

**Figure 6.26. JComponents in the view**



## Rendering

Using Swing UI components for the visual representation of a node can be accomplished by providing an implementation for interface [NodeCellRenderer](#). This interface behaves similar to Java Swing interfaces such as TableCellRenderer or TreeCellRenderer, and serves as a callback for the logic that is responsible for painting the node realizer. [API Excerpt 6.32, “Callback method of interface NodeCellRenderer” \[195\]](#) presents the method that is defined by NodeCellRenderer.

### API Excerpt 6.32. Callback method of interface NodeCellRenderer

```
// Parameter 'value' holds the value of the cell to be rendered, i.e., the
// actual node-related data that is presented in the JComponent.
JComponent getNodeCellRendererComponent(Graph2DView view, NodeRealizer context,
                                         Object value, boolean isSelected)
```

Class [NodeCellRendererPainter](#) provides support for rendering Swing UI components with [GenericNodeRealizer](#). It is an implementation for interface [GenericNodeRealizer.Painter](#) which is responsible for painting [generic node realizers](#). A NodeCellRendererPainter instance can be instantiated using a NodeCellRenderer implementation and be added to a GenericNodeRealizer configuration. The given NodeCellRenderer implementation is then invoked to return the JComponent that serves as the visual representation for a node.

Defining a GenericNodeRealizer configuration that uses NodeCellRendererPainter is outlined in [Example 6.14, “Defining a GenericNodeRealizer configuration that supports rendering of UI components” \[195\]](#). Tutorial demo application [SwingRendererDemo.java](#) shows how to define a corresponding NodeCellRenderer that returns a JComponent.

### Example 6.14. Defining a GenericNodeRealizer configuration that supports rendering of UI components

```

void createJTextFieldNodeRealizer()
{
    // Create a customized NodeCellRenderer instance that defines the visual
    // representation of a node.
    NodeCellRenderer myNCR = new MyJTextFieldNodeCellRenderer();

    // Define a GenericNodeRealizer configuration that uses 'myNCR' as a callback
    // for class NodeCellRendererPainter.
    // This class provides support for rendering Swing UI components.
    GenericNodeRealizer.Factory factory = GenericNodeRealizer.getFactory();
    Map map = factory.createDefaultConfigurationMap();
    map.put(
        GenericNodeRealizer.Painter.class,
        new NodeCellRendererPainter(myNCR, NodeCellRendererPainter.USER_DATA_MAP));
    map.put(
        GenericNodeRealizer.UserDataHandler.class,
        new SimpleUserDataHandler(SimpleUserDataHandler.REFERENCE_ON_FAILURE));

    // Register the configuration using the given name.
    factory.addConfiguration("JTextField", map);
}

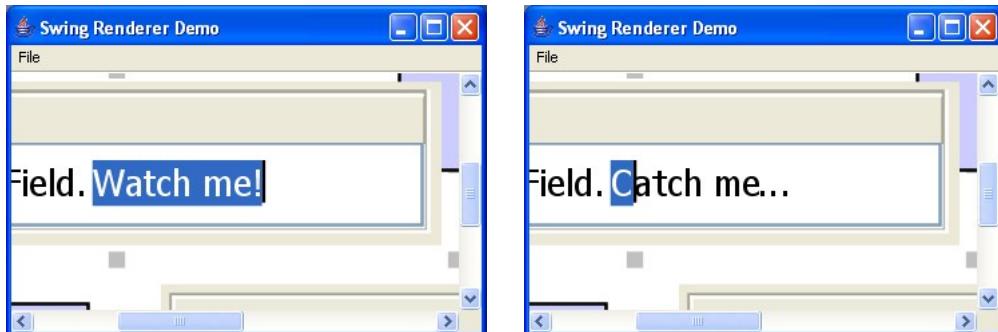
```

The UI components rendered in the view provide the full range of mouse-based user interaction that is also supported for "ordinary" nodes, i.e., they can be selected, moved around, or scaled using their resize knobs, for example. They are also subject to changes of the view port such as zooming in or out.

## Editor Functionality

In addition to using Swing UI components for the visual representation of nodes by means of a NodeCellRenderer implementation (as shown above), the components can also be used as proper editors. As a special feature, these Swing-based editors remain editable even when zoomed. [Figure 6.27, “Editing a zoomed JComponent” \[196\]](#) shows a zoomed-in JTextField before and after editing its value.

**Figure 6.27. Editing a zoomed JComponent**



Enhancing a node realizer that renders a JComponent with editor functionality can be accomplished by providing an implementation for interface [NodeCellEditor](#). The section [Node-Related Editing Capabilities](#) describes this interface and its role for user interaction with UI components.

Class [CellEditorMode](#) is the view mode that knows all nodes that have a Swing-based editor associated. It recognizes whenever a user attempts to use such an editor and initiates node editing. See the section called [“Class CellEditorMode” \[215\]](#) for a detailed description of this view mode.

## Tutorial Demo Code

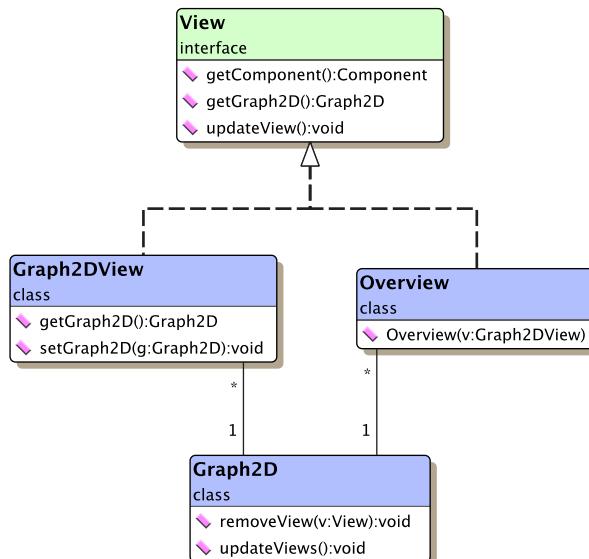
The tutorial demo application [SwingRendererDemo.java](#) shows how Java Swing UI components can be used for both rendering the visual representation of nodes and additionally providing editors for node-related data.

## View Implementations

A view, as described by the Model-View-Controller (MVC) paradigm, is responsible for presenting the model to the user. Also, events generated by the user are propagated to the proper controller classes that handle user interaction.

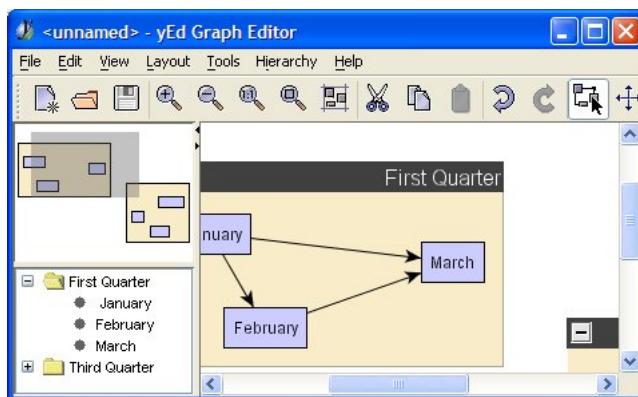
Interface [View](#) sets the general frame for actual yFiles view implementations. It provides for a getter method for the contained Graph2D instance, and for methods that are called to update, i.e., (re)paint, either the view's entire content or only (a specified rectangular) part of it. [Figure 6.28, “Class hierarchy for view implementations”](#) [197] gives an overview of the predefined views and their connection to the graph structure.

**Figure 6.28. Class hierarchy for view implementations**



[Figure 6.29, “Major view implementations”](#) [197] shows the major view implementations provided by package [y.view](#) in an application context. Class [Graph2DView](#) fills the right side of the application frame, an associated instance of class [Overview](#) is in the upper left.

**Figure 6.29. Major view implementations**



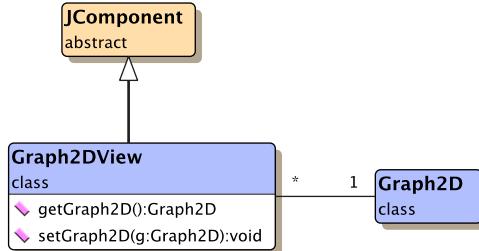
## Class Graph2DView

The canvas on which a graph is painted is an instance of type [Graph2DView](#). It is plainly called

"the view," or is referred to as "the view port." (The latter term is most notably used whenever the view's dimension is of particular importance.) The view is responsible for invoking all necessary actions to paint the elements of the contained graph whenever it is asked to.

Though a graph can be displayed by multiple views at the same time, a view can contain and display at most one graph. Figure 6.30, “[Class hierarchy for class Graph2DView](#)” [198] shows the inheritance structure for class Graph2DView.

**Figure 6.30. Class hierarchy for class Graph2DView**



To access the view's actual canvas component, e.g., to use it as a drop target in a drag-and-drop operation, class Graph2DView provides the method from API Excerpt 6.33, “Getting access to the view's canvas component” [198]

### API Excerpt 6.33. Getting access to the view's canvas component

```
// Returns the view's canvas component.  
JComponent getCanvasComponent()
```

## World and View Coordinates

There are different coordinate spaces used in the model itself and for its presentation in the view, the so-called "world coordinates" and the view coordinates.

The common thing to both world and view coordinate space is the way coordinates increase with each direction. Coordinates on computer screens most commonly increase from left to right and from top to bottom. Note that this is diametrical to the normal Cartesian coordinate system known from school where coordinates increase from left to right and from bottom to top.

The view coordinates always start with  $(0, 0)$  for the upper left corner and increase to the right and to the bottom. The coordinates of the lower right corner accordingly equals to  $(\text{view\_width}-1, \text{view\_height}-1)$ , i.e., the view coordinates directly reflect the view port's size.

The world coordinates, in contrast, are those valid in the model, i.e., they denote the proper locations that are set for a node's center or the control points of an edge.

## Note

One characteristic of world coordinates is that they can have negative values, while view coordinates are always positive.

The mapping from one coordinate space to the other happens automatically and is achieved using the notion of the so-called "view point" and scaling. The view point determines the world coordinates that get directly mapped to the upper left corner of the view port, i.e., it defines where the view's origin lies in the world coordinates's two-dimensional space.

An additional scaling factor that can be applied to the world coordinates determines the zoom level with which the view scales the model's presentation. [API Excerpt 6.34, “Methods to control view point and scaling” \[198\]](#) shows the available methods to control view point and scaling.

#### **API Excerpt 6.34. Methods to control view point and scaling**

```
// Getter methods.
double getZoom()
Point getViewPoint()
Point2D getViewPoint2D()

// Setter methods.
void setViewPoint(int x, int y)
void setViewPoint2D(double x, double y)
void setZoom(double zoom)
```

[Example 6.15, “Coordinate conversion methods” \[199\]](#) shows the coordinate conversion using the view point and scaling in full detail. Note that the listed methods can just as well be called explicitly for any coordinate conversions required.

### Example 6.15. Coordinate conversion methods

```
// From world to view coordinates...
int toViewCoordX(double x){ return (int)((x - viewPoint.x) * zoom); }
int toViewCoordY(double y){ return (int)((y - viewPoint.y) * zoom); }

// ... and vice-versa.
double toWorldCoordX(int x){ return x/zoom + viewPoint.x; }
double toWorldCoordY(int y){ return y/zoom + viewPoint.y; }
```

## View Port and Clipping

Class Graph2DView offers several methods to conveniently manipulate the view port and reflect any view port changes to the model. The most notable of these methods control the view's zoom level, its center, or its size and location (i.e., the view point).

Also, controlling the view's behavior to automatically adapt to the bounding box of the graph is supported. [API Excerpt 6.35, “Clipping modification methods” \[199\]](#) lists all methods for changing the view's clipping.

### API Excerpt 6.35. Clipping modification methods

```
// Getter methods.
Point2D getCenter()
Rectangle getVisibleRect()
Rectangle getWorldRect()
double getZoom()
boolean getFitContentOnResize()
Point getViewPoint()
Point2D getViewPoint2D()

// Setter methods.
void setCenter(double x, double y)
void setFitContentOnResize(boolean b)
void setViewPoint(int x, int y)
void setViewPoint2D(double x, double y)
void setWorldRect(int x, int y, int w, int h)
void setZoom(double zoom)

// Miscellaneous.
void fitContent()
void fitRectangle(Rectangle r)
void fitWorldRect()
void focusView(double newZoom, Point2D newCenter, boolean animated)
void updateWorldRect()
void zoomToArea(double x, double y, double width, double height)
```

## Scroll Bars

[API Excerpt 6.36, “Scroll bar methods” \[200\]](#) lists the getter and setter methods to control the policy settings of class Graph2DView's already built-in scroll bars. By default, they obey the policy settings HORIZONTAL\_SCROLLBAR\_AS\_NEEDED and VERTICAL\_SCROLLBAR\_AS\_NEEDED as defined in interface javax.swing.ScrollPaneConstants.

Note that using JScrollPane objects with Graph2DView is neither necessary nor supported, i.e., with respect to scroll bars class Graph2DView differs in behavior compared to normal Java

Swing components.

### API Excerpt 6.36. Scroll bar methods

```
// Getter methods.
int getHorizontalScrollBarPolicy()
int getVerticalScrollBarPolicy()

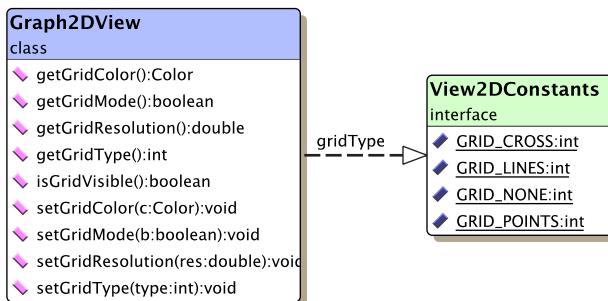
// Setter methods.
void setHorizontalScrollBarPolicy(int policy)
void setScrollBarPolicy(int vsbp, int hsbp)
void setVerticalScrollBarPolicy(int policy)
```

Furthermore, Graph2DView knows the `UnitIncrementPixels` client property which can be used to specify the unit increment for the scroll bars. The integral value for this property is interpreted in view coordinate space.

## Grid

Class Graph2DView optionally supports rendering of a grid in the background of a graph. (See below for details on the [rendering order](#) that is applied for the visual representation of a graph.)

**Figure 6.31. Graph2DView's grid support**



The grid's parameter as well as all its graphical attributes can be fully customized using the methods from [API Excerpt 6.37, “Grid-related methods” \[200\]](#)

When the view renders a grid in the background of a graph, it is in so-called "grid mode." The predefined yFiles view modes (see [the section called “User Interaction” \[208\]](#)) query the view for this exact state (using `Graph2DView`'s `getGridMode()` method) and then apply special logic that snaps coordinates to proper grid points.

### API Excerpt 6.37. Grid-related methods

```
// Getter methods.
Color getGridColor()
boolean getGridMode()
double getGridResolution()
int getGridType()
boolean isGridVisible()

// Setter methods.
void gridCursorActive(boolean active)
void setGridColor(Color color)
void setGridCursorPosition(double x, double y)
void setGridMode(boolean b)
void setGridResolution(double res)
void setGridType(int type)
void setGridVisible(boolean b)
```

## Selection Box

The look of the selection box that is drawn when a user selects multiple graph elements with a

mouse drag gesture can be conveniently customized using a set of client properties that are recognized by class Graph2DView. [Table 6.8, “Selection box-related client properties of class Graph2DView” \[201\]](#) lists the clients properties related to selection box rendering.

**Table 6.8. Selection box-related client properties of class Graph2DView**

Client Property	Value Type	Description
selection-box.rectangularshape	RectangularShape	A <code>java.awt.geom.RectangularShape</code> object that provides the shape representation of the selection box.
selection-box.fillcolor	Color	A (transparent) <code>java.awt.Color</code> that is used to fill the selection box.
selection-box.linecolor	Color	A <code>java.awt.Color</code> that is used to draw the border of the selection box.
selection-box.stroke	Stroke	A <code>java.awt.Stroke</code> object that is used to draw the border of the selection box.

Alternatively, selection box rendering can be completely re-defined by means of the method shown in [API Excerpt 6.38, “Selection box-related methods” \[201\]](#)

### API Excerpt 6.38. Selection box-related methods

```
protected void paintSelectionBox(Graphics2D gfx, Rectangle2D box)
```

## Rendering Order

The visual representation of a graph that is rendered to a view, is composed of five different layers where graph elements and further graphical objects are placed. From background to foreground these layers are:

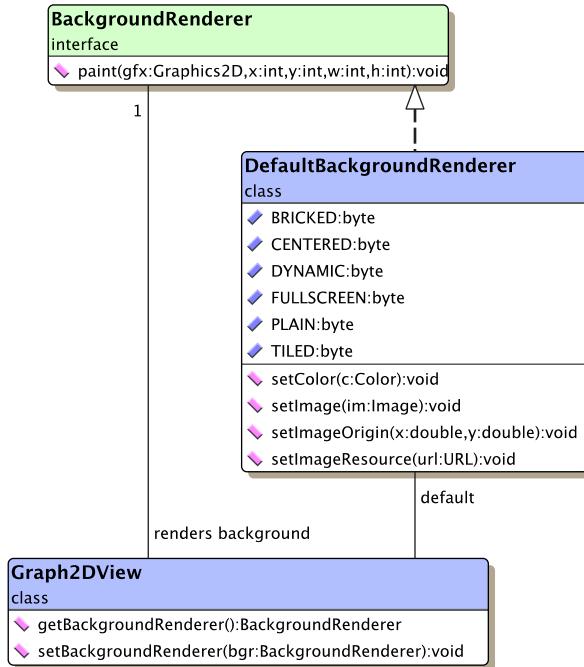
1. Background graphics, i.e., arbitrary images, patterns, or colors (governed by implementations of interface [BackgroundRenderer](#))
2. Grid (done by Graph2DView itself)
3. Background drawables
4. Graph elements (governed by implementations of interface [Graph2DRenderer](#))
5. Foreground drawables

Class [DefaultBackgroundRenderer](#) is the default implementation of interface `BackgroundRenderer`. It handles the rendering of any background graphics, e.g., images, patterns, or colors.

**Figure 6.32. DefaultBackgroundRenderer class hierarchy**

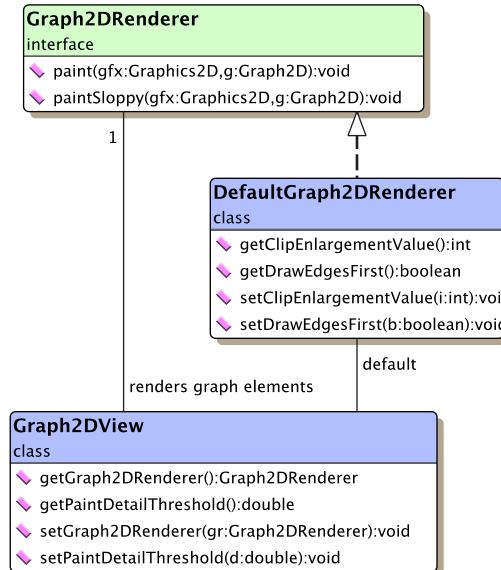
## Class Graph2DView

---



Class **DefaultGraph2DRenderer** is the default implementation of interface **Graph2DRender**. Its responsibility is to invoke the realizer that is bound to a graph element and make this realizer actually render the graph element.

**Figure 6.33. DefaultGraph2DRenderer class hierarchy**



Depending on the current zoom level and the value set using method `setPaintDetailThreshold` it calls either the realizer's normal rendering logic or the method that does less-detail rendering.

Less-detail rendering is invoked whenever the current zoom level is below the value set for the paint detail threshold. In effect, it means that a graph element is painted omitting any details that

would clutter the display, i.e., rendering is confined to those things that make up the outline of a graph element. The descriptions in the section called “Bringing Graph Elements to Life: The Realizer Concept” [164] have further explanations on the specific less-detail rendering for the predefined node and edge realizers.

By default, the sequence that is used to draw graph elements is first all nodes, then all edges. However, class DefaultGraph2DRenderer also allows to revert this behavior, i.e., optionally edges can be drawn before nodes.

Within each set of graph elements their respective sequence from first to last also determines their drawing order, so that the last node from the graph's node set, for example, is rendered atop all other nodes. Using the methods from class [Graph](#) as listed in [API Excerpt 6.39, “Changing the sequence of graph elements”](#) [203] the rendering order for both types of graph elements can easily be adjusted by changing the sequence of their respective set.

### API Excerpt 6.39. Changing the sequence of graph elements

```
// Moving graph elements to the front of their set.
void moveToFirst(Edge e)
void moveToFirst(Node v)

// Moving graph elements to the back of their set.
void moveToLast(Edge e)
void moveToLast(Node v)
```

### Important

For hierarchically organized graphs, DefaultGraph2DRenderer does not use the sequence of nodes directly, but instead queries the HierarchyManager instance that is registered with a root graph to determine the actual rendering order of nodes.

To reflect node order changes also for hierarchically organized graphs, class [HierarchyManager](#) offers corresponding methods to change the sequence of nodes within their respective parent entity (either root graph, group node, or folder node; see also [Chapter 7, Graph Hierarchies](#) [235] for the descriptions of these terms and concepts). As a consequence, all Graph operations that affect the node order have to be accompanied by corresponding HierarchyManager operations for the particular node. See [Table 6.9, “Corresponding node order methods”](#) [203]

**Table 6.9. Corresponding node order methods**

Graph	HierarchyManager
<a href="#">moveToFirst(Node)</a>	<a href="#">moveToFirst(Node)</a>
<a href="#">moveToLast(Node)</a>	<a href="#">moveToLast(Node)</a>

Fine-grained control over the drawing order of graph elements can be achieved by using DefaultGraph2DRenderer's support for so-called "layered painting." It provides the means to render nodes and edges in arbitrary order.

Layered painting introduces the notion of "layers," where graph elements can reside in for rendering. The layers are numbered and they are rendered in ascending order, i.e., graph elements in a layer with a small index are rendered before those in a layer with a higher index. Within a layer there can be both nodes and edges; also there can be an arbitrary number of graph elements of either type in a layer.

The rendering order for the graph elements within a layer is governed by the overall settings of DefaultGraph2DRenderer. In particular, this includes whether edges should be drawn before nodes or vice-versa.

[API Excerpt 6.40, “Layered painting support in DefaultGraph2DRenderer”](#) [203] lists the methods for enabling layered painting and also the callback methods for specifying the actual layer that a node or an edge should reside in for rendering.

### API Excerpt 6.40. Layered painting support in DefaultGraph2DRenderer

```
// Getter and setter methods for layered painting.
```

```
boolean isLayeredPainting()
void setLayeredPainting(boolean layeredPainting)

// Callback methods for specifying a graph element's layer.
protected int getLayer(Graph2D graph, Node node)
protected int getLayer(Graph2D graph, Edge edge)
```

Further functionality of class DefaultGraph2DRenderer is described in [the section called “Class DefaultGraph2DRenderer” \[205\]](#)

Another option provided is to enable/disable anti-aliased rendering for the entire view port, i.e., for all graphical objects therein. [API Excerpt 6.41, “Paint detail and anti-alias methods of Graph2DView” \[204\]](#) summarizes the methods of Graph2DView that can be used to control the paint detail threshold respectively anti-aliasing.

### API Excerpt 6.41. Paint detail and anti-alias methods of Graph2DView

```
// Getter methods.
double getPaintDetailThreshold()
boolean isAntialiasedPainting()

// Setter methods.
void setPaintDetailThreshold(double thresh)
void setAntialiasedPainting(boolean antialiasedPainting)
```

Foreground and background drawables are additional graphical objects of type `Drawable` that can be added to the visual representation of a graph. The [discussion of interface Drawable shows more details on this type](#). [API Excerpt 6.42, “Methods dealing with objects of type Drawable” \[?\]](#) lists Graph2DView methods that are used to add and remove drawables.

### API Excerpt 6.42. Methods dealing with objects of type Drawable

```
// Adding an object of type y.view.Drawable.
void addBackgroundDrawable.Drawable d)
void addDrawable.Drawable d)

// Removing an object of type y.view.Drawable.
void removeBackgroundDrawable.Drawable d)
void removeDrawable.Drawable d)
```

## Tutorial Demo Code

The following applications serve as an introduction to class Graph2DView's features:

- [SimpleDemo.java](#)
- [BuildGraphDemo.java](#)
- [ViewActionDemo.java](#)
- [BackgroundDemo.java](#) focusses on aspects of the rendering order

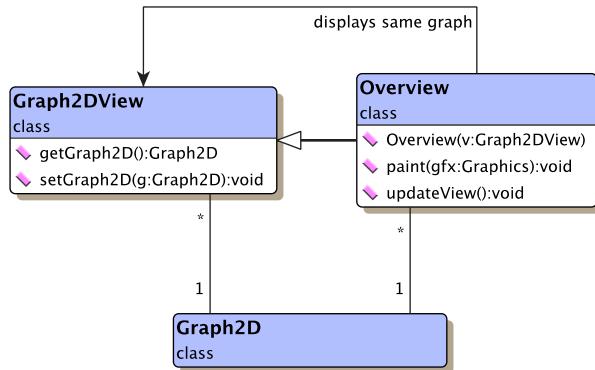
## Class Overview

Class [Overview](#) is a specialization of Graph2DView that always shows a graph in its entirety. It has an instance of its superclass associated and displays the same graph as this instance.

Additionally, a visual feedback for the current clipping area of the associated Graph2DView instance is superposed. This superposed rectangle can be moved using the mouse, thereby moving the clipping area accordingly.

Class [Overview](#) allows to customize the actual appearance of this visual indication and also many aspects of user interaction. For example, the area either inside or outside of the rectangle can be blurred, different scrolling behaviors can be set, or mouse wheel support to directly change the zoom level of the associated Graph2DView instance can be enabled.

### Figure 6.34. Class hierarchy for class Overview



## Note

To reduce drawing complexity, class **Overview** does not display the graph in the same manner as the associated **Graph2DView** instance. Instead, the graph is displayed with far less details, e.g., the labels are missing, there are no arrows at the ends of edges and nodes are always drawn as rectangles.

## Related Classes and Interfaces

### Class DefaultGraph2DRenderer

Class **DefaultGraph2DRenderer** is the default implementation of interface **Graph2DRenderer**. Its responsibility is to invoke the realizer that is bound to a graph element and make this realizer actually render the graph element inside the view. (See also the section called “[Rendering Order](#)” [201] for a description of how **DefaultGraph2DRenderer** interacts with the view.)

**DefaultGraph2DRenderer** optionally allows to use the services of class **BridgeCalculator** to insert so-called “bridges” into edge paths. Bridges are a means to resolve the visual ambiguity induced by intersecting edge paths. Each segment of an edge path that intersects with at least one other segment (from either the same or another edge path) which has already been rendered, can be augmented with a bridge in one of a variety of different styles.

Alternatively, instead of using the drawing order induced by the edge set to determine which segment to augment, the set of edge segments that are subject to augmentation can be restricted with respect to segment orientation. Edge segments that have either predominant horizontal or vertical orientation can be chosen to always lie atop of edge segments having contrary orientation. [Table 6.10, “Bridge styles of class BridgeCalculator”](#) [205] presents several of the bridge styles available with **BridgeCalculator**.

**Table 6.10. Bridge styles of class BridgeCalculator**

Constant Name	Bridge Style
CROSSING_STYLE_GAP	
CROSSING_STYLE_ARC	
CROSSING_STYLE_SQUARE	
CROSSING_STYLE_TWOSIDES	

## Note

By default, augmenting edge paths with bridges is only supported for straight-line segments of poly-line edge paths, i.e., in particular for edge realizers of type PolyLineEdgeRealizer.

**Example 6.16, “Using class BridgeCalculator with DefaultGraph2DRenderer”** [206] shows how to add a BridgeCalculator instance to a view’s DefaultGraph2DRenderer object.

### Example 6.16. Using class BridgeCalculator with DefaultGraph2DRenderer

```
void enableBridgesForEdgePaths(Graph2DView view)
{
    Graph2DRenderer gr = view.getGraph2DRenderer();
    if (gr instanceof DefaultGraph2DRenderer)
    {
        DefaultGraph2DRenderer dgr = (DefaultGraph2DRenderer)gr;
        // If there is no BridgeCalculator instance set, ...
        if (dgr.getBridgeCalculator() == null)
            // ... then register a newly created one that uses default settings.
        dgr.setBridgeCalculator(new BridgeCalculator());
    }
}
```

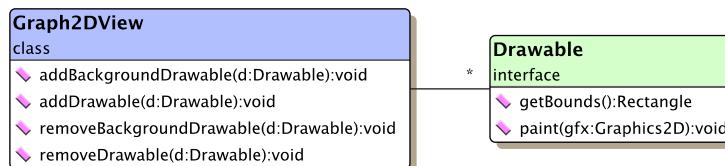
Tutorial demo application [BridgeDemo.java](#) demonstrates how the bridging functionality of DefaultGraph2DRenderer is enabled to resolve intersecting edge paths. [BridgeEdgeRealizer-Demo.java](#) shows how class BridgeCalculator can be used in conjunction with an implementation of class GenericEdgeRealizer.

## Interface Drawable

Interface **Drawable** sets the general frame for graphical objects that should be added to the visual presentation of a graph and be automatically (re)painted whenever necessary. Class **Graph2DView** has support to add Drawable implementations (or "drawables" for short) either to the foreground or the background of a graph. Similar support is provided to remove drawables.

Foreground drawables will be drawn above all graph elements, while background drawables will be drawn underneath.

**Figure 6.35. Interface Drawable**



As with graph elements it is important for a drawable that it provides information about its bounds. The proper dimension and location of all objects that have to be rendered are essential for correctly (re)painting a view's content.

Table 6.11, “Drawable implementations” [206] gives a brief overview on convenience implementations for interface Drawable.

**Table 6.11. Drawable implementations**

Classname	Description
<a href="#">CompositeDrawable</a>	Defines a new drawable by composing already existing ones.
<a href="#">ShapeDrawable</a>	Allows to use an existing <code>java.awt.Shape</code> object to define its visual representation.

The tutorial demo application [DrawablesDemo.java](#) shows the implementation of drawables and how they are added to the graph.

### Class Graph2DViewMouseWheelZoomListener

Class [Graph2DViewMouseWheelZoomListener](#) is a [java.awt.event.MouseWheelListener](#) implementation that adds convenient mouse wheel zooming to a view. [Example 6.17, “Adding mouse wheel zooming to a view” \[207\]](#) outlines how mouse wheel zooming can be enabled.

#### Example 6.17. Adding mouse wheel zooming to a view

```
public Graph2DViewMouseWheelZoomListener addMouseWheelZooming(Graph2DView view)
{
    // Create a default mouse wheel zoom listener.
    Graph2DViewMouseWheelZoomListener mwzl = new Graph2DViewMouseWheelZoomListener();
    // Register it with the given view's canvas component.
    view.getCanvasComponent().addMouseWheelListener(mwzl);

    return mwzl;
}
```

# User Interaction

In terms of the Model-View-Controller (MVC) paradigm, the controller part is the tie between the model and the view. It handles the user interaction and accordingly changes state of both model and view.

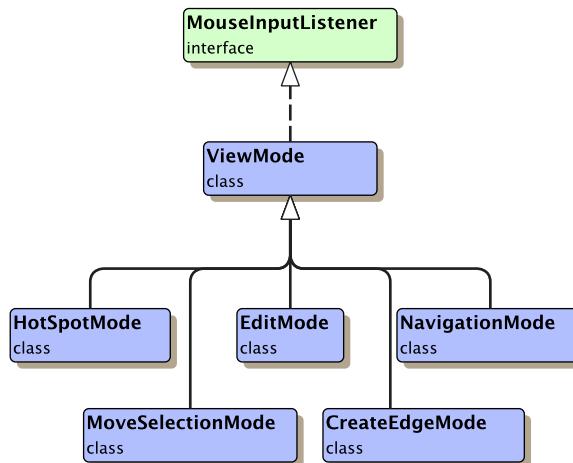
## Mouse-Driven Actions

All mouse-based user interaction is handled by class `ViewMode` and its descendants. See below for a [detailed description of class `ViewMode`](#). The term "mouse-based" summarizes simple mouse clicks as well as more complex gestures that denote, e.g.:

- creation of graph elements
- interaction with graph elements, e.g., selecting some nodes, moving a selection, resizing a node, etc.
- manipulation of the view port, e.g., moving the clipping area

[Figure 6.36, “ViewMode class hierarchy” \[208\]](#) shows the hierarchy of class `ViewMode` featuring a subset of its descendants.

**Figure 6.36. ViewMode class hierarchy**



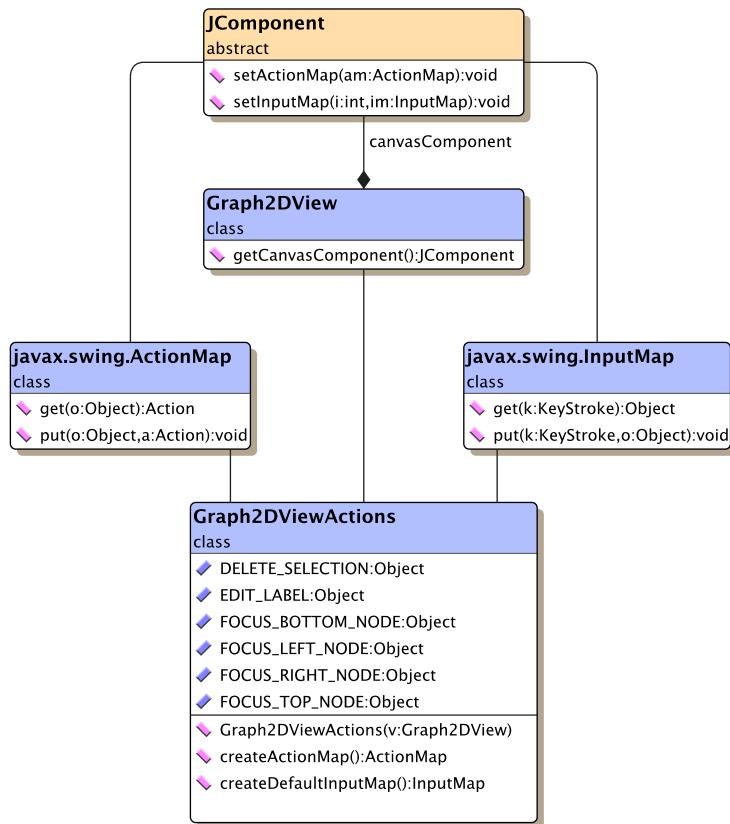
## Keyboard-Driven Actions

Editing a graph is most naturally done using the mouse, but keyboard input is equally important, e.g., to change a label's text, to delete selected graph elements, or to quickly select nodes or edges. Class [Graph2DViewActions](#) provides a set of default keyboard actions that can be conveniently installed with a `Graph2DView`. [Example 6.18, “Installing default keyboard actions with a Graph2DView” \[208\]](#) presents the single line needed to accomplish this.

### Example 6.18. Installing default keyboard actions with a `Graph2DView`

```
// 'myView' is of type y.view.Graph2DView.
new Graph2DViewActions().install(myView);
```

Customized keyboard actions can easily be registered with a `Graph2DView` object using the Java Swing classes `ActionMap` and `InputMap`. [Figure 6.37, “Using ActionMap and InputMap” \[209\]](#) depicts their interaction with the view and with `Graph2DViewActions`.

**Figure 6.37. Using ActionMap and InputMap**

The set of default keyboard actions provided by **Graph2DViewActions** can be conveniently retrieved readily encoded in **ActionMap** and **InputMap** objects, respectively. See [Example 6.19, “Registering keyboard actions” \[209\]](#) for the necessary steps to get the default keyboard actions, to add further actions to the set, and then register them altogether with the view.

### Example 6.19. Registering keyboard actions

```

public void registerSimpleZoomKeyboardActions(Graph2DView view)
{
    Graph2DViewActions actions = new Graph2DViewActions(view);
    // Create the maps to hold the actions.
    ActionMap aMap = actions.createActionMap();
    InputMap iMap = actions.createDefaultInputMap(aMap);

    // Add custom keyboard actions to the maps.
    // Connect 'zoom in' action to num pad '+' key.
    iMap.put(KeyStroke.getKeyStroke(KeyEvent.VK_ADD, 0), "ZOOM IN");
    aMap.put("ZOOM IN", new Zoom(1.2));

    // Connect 'zoom out' action to num pad '-' key.
    iMap.put(KeyStroke.getKeyStroke(KeyEvent.VK_SUBTRACT, 0), "ZOOM OUT");
    aMap.put("ZOOM OUT", new Zoom(0.8));

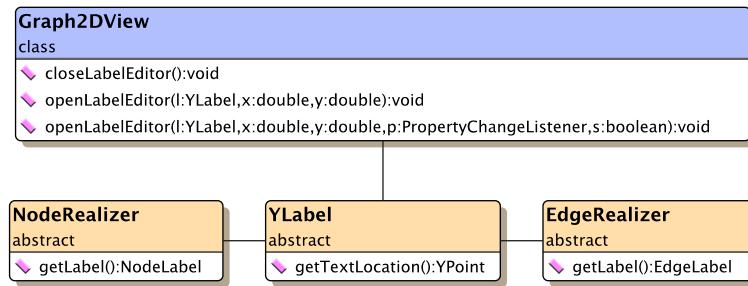
    // Register the maps with the canvas component of 'view.'
    view.getCanvasComponent().setActionMap(aMap);
    view.getCanvasComponent().setInputMap(JComponent.WHEN_FOCUSED, iMap);
}
  
```

## Label Editor

The built-in label editor directly provided by class **Graph2DView** can be opened in-place, i.e., at the label's location, or at any other suitable place in the view. It supports single-line as well as

multi-line labels.

**Figure 6.38. Graph2DView's built-in label editor**



Using the callback mechanism provided by interface `PropertyChangeListener` notifications for label change events are easily realized.

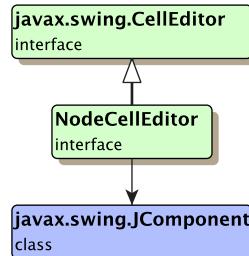
Method `getTextLocation` from abstract class `YLabel` delivers the proper location for the label editor to appear in-place, i.e., where the actual label resides. Note that the location where the label editor should open is given in world coordinates rather than view coordinates.

Among other things, `PopupModeDemo.java` demonstrates the usage of `Graph2DView`'s label editor.

## Node-Related Editing Capabilities

Interface `NodeCellEditor` enables the use of `JComponents` as editors for node-related data. It behaves similar to Java Swing interfaces such as `TableCellEditor` or `TreeCellEditor`, and serves as a callback for the view mode descendant class `CellEditorMode`.

**Figure 6.39. Class NodeCellEditor's context**



API Excerpt 6.43, “Callback method of interface `NodeCellEditor`” [210] presents the method that is defined by `NodeCellEditor`. It is called by `CellEditorMode` to initiate node editing using the returned `JComponent`.

### API Excerpt 6.43. Callback method of interface `NodeCellEditor`

```

// Parameter 'value' holds the value of the cell to be edited, i.e., the actual
// node-related data that is presented in the JComponent.
JComponent getNodeCellEditorComponent(Graph2DView view, NodeRealizer context,
                                       Object value, boolean isSelected)
  
```

See the section called “Class `CellEditorMode`” [215] for a description of the view mode that enables node editing using node cell editors. Tutorial demo application `SwingRendererDemo.java`

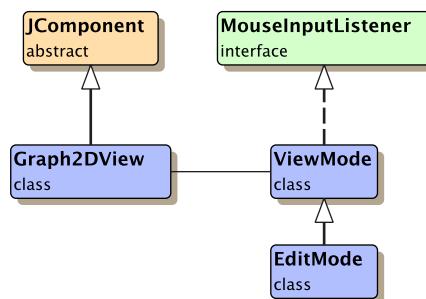
shows how NodeCellEditor implementations can be created using Java Swing's abstract class `AbstractCellEditor` as a convenient basis.

## ViewMode Workings

Class `ViewMode` provides the base functionality to handle mouse-based interaction that happens in a view. User-generated mouse events like, e.g., a mouse button press, release, or mouse drag with one of the buttons down are received and processed by appropriate methods.

Processing mouse events in class `ViewMode` has two faces to it. First of all, there are "low-level" methods from the implementation for Java Swing interface `MouseListener`. Then there are additional, but more high-level and application-oriented methods available. The former methods convey actual mouse events that use view coordinates, while the latter ones directly use coordinates from the model, i.e., they are already converted to world coordinates.

**Figure 6.40. ViewMode hierarchy**



View modes can be "chained," i.e., one view mode can yield control to another, which is referred to as a "child mode" in this context. Yielding control to a child mode means setting it as the active child mode, which is done using appropriate methods from class `ViewMode`. The child mode (or simply "child") then proceeds processing mouse events and after it is done returns control to its parent mode (its "parent"). Note that a view mode can have at most one active child mode at any time. [API Excerpt 6.44, "ViewMode methods to get the parent and child modes" \[211\]](#) lists the methods that can be used to get both the parent and the child of a view mode.

### Note

The term "child mode" is just another name for a view mode that is used in a specific context, namely view mode chaining. In particular, every child mode is a proper view mode that can be used on its own, i.e., without the need for a "parent mode."

### API Excerpt 6.44. ViewMode methods to get the parent and child modes

```

ViewMode getChild()
ViewMode getParent()
  
```

View modes are added to respectively removed from a view using `Graph2DView`'s methods shown in [API Excerpt 6.45, "ViewMode-related methods from class Graph2DView" \[211\]](#)

### API Excerpt 6.45. ViewMode-related methods from class Graph2DView

```

// Methods to add and remove a view mode to the canvas.
void addViewMode(ViewMode vm)
void removeViewMode(ViewMode vm)
  
```

To provide convenient notification support class `ViewMode` offers the methods presented in [API Excerpt 6.46, "Property change support in class ViewMode" \[212\]](#) to add implementations of in-

terface `PropertyChangeListener`. The listeners are notified whenever one of the view mode's properties changes, most notably when the view mode gets activated or deactivated or when it enters or leaves editing state. Other property changes that are also signaled include, e.g., a view mode's usage as a child mode or as a parent mode.

### API Excerpt 6.46. Property change support in class ViewMode

```
// Methods to add and remove property change listeners to a view mode.
void addPropertyChangeListener(PropertyChangeListener pcl)
void addPropertyChangeListener(String pName, PropertyChangeListener pcl)

void removePropertyChangeListener(PropertyChangeListener pcl)
void removePropertyChangeListener(String pName, PropertyChangeListener pcl)
```

A view mode's editing state characterizes whether the view mode currently processes some user interaction. This property is appropriately updated by all predefined view modes and should also by any customized variant. [API Excerpt 6.47, “Controlling a view mode's editing state” \[212\]](#) shows the methods that support getting and setting the editing state.

### API Excerpt 6.47. Controlling a view mode's editing state

```
// Getter and setter methods for a view mode's editing state.
boolean isEditing()
protected void setEditing(boolean editing)
```

Notifications about view mode property changes complement the notification mechanism that signals property changes relating to a graph's visual representation. Registering listeners with view modes provides qualified information about both beginning and ending of any mouse-based user interaction while listeners that receive `Graph2DEvent` objects are continuously notified during the interaction.

## Class EditMode

Among all `ViewMode` descendants class `EditMode` takes the most prominent role. It is the major mode that serves as the entry point for mouse events from the view. By default, it has a number of child modes associated, each of which specialized in handling mouse gestures within a specific context. [Table 6.12, “Specialized view modes associated with class EditMode” \[212\]](#) lists all of class `EditMode`'s child modes.

**Table 6.12. Specialized view modes associated with class EditMode**

Classname	Description
<code>CellEditorMode</code>	Can be registered to handle node editing. See also <a href="#">the section called “Class CellEditorMode” [215]</a>
<code>CreateEdgeMode</code>	Handles edge creation including the creation of control points. Edge creation does also cover aborting the process of creating an edge.
<code>HotSpotMode</code>	Handles node resizing.
<code>MouseInputMode</code>	Delegates mouse input to so-called mouse input editors using double precision mouse events. See <a href="#">the section called “Interface MouseInputEditor” [218]</a> for a description of the mouse input editor concept.
<code>MoveLabelMode</code>	Handles node and edge label moving and triggers the presentation of label position candidate boxes.
<code>MovePortMode</code>	Handles moving the port of an edge at the same node.
<code>MoveSelectionMode</code>	Handles moving a collection of selected graph elements.
<code>MoveViewPortMode</code>	Handles moving view port.
<code>OrthogonalMoveBendsMode</code>	Provides support for moving bends and edge segments in such a way that an edge's path is always orthogonal. This optional mode can be activated using method <code>setOrthogonalBendMove-</code>

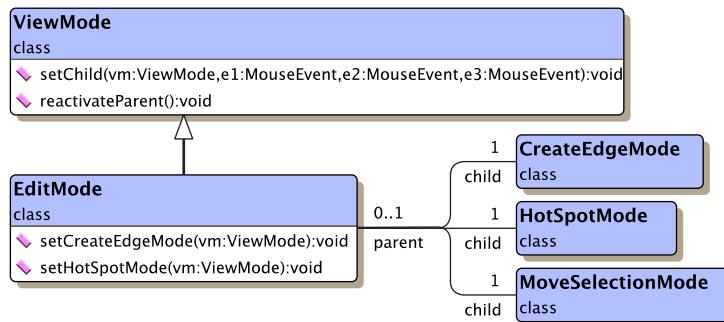
## Class EditMode

---

Classname	Description
	<code>mentEnabled(boolean).</code>
PopupMode	Handles context-sensitive menus.
SelectionBoxMode	Handles creating a rectangular box in the view and changes the selection state of graph elements contained in that box.

Depending on the given context, class `EditMode` delegates the processing to the appropriate child mode by chaining it, i.e., setting it as the active child mode.

**Figure 6.41. Child modes of class EditMode**



Chaining of another view mode is done as outlined in [Example 6.20, “Chaining another view mode” \[214\]](#) using one of the methods from [API Excerpt 6.48, “Child mode methods from class ViewMode” \[213\]](#). These methods are vital to chaining view modes, among other things, they are responsible for:

- Setting the parent mode's view as the active view for the child mode
- Providing the child with a reference to its parent
- Invoking the child mode's initialization method

### API Excerpt 6.48. Child mode methods from class ViewMode

```

// Available setChild methods.
void setChild(ViewMode child,
               MouseEvent pressEvent,
               MouseEvent dragEvent)
void setChild(ViewMode child,
               MouseEvent pressEvent,
               MouseEvent dragEvent,
               MouseEvent releaseEvent)
  
```

To provide the child mode with a well-defined state for its start, the most current mouse events are handed over in canonical order, i.e., press, drag, and release. All further mouse events are then received by both parent and child, but processed only by the child mode.

### Important

View modes rely on a contract regarding the mouse events they are given at startup. While the mouse release event, or both mouse drag and mouse release event can be null, there must always be given a valid mouse press event.

After processing is completed, the child mode returns control to its parent invoking its `reactivateParent()` method. Note that this method uses the reference to the parent that has been set before by the chaining mechanism. If no parent is set, then calling the method has no effect.

### Example 6.20. Chaining another view mode

```
public void mousePressedRight(double x, double y)
{
    // 'lastPressEvent,' 'lastDragEvent,' and 'lastReleaseEvent' are of type
    // java.awt.event.MouseEvent
    // NB: These are protected fields of class y.view.ViewMode

    // Create a new instance of the specialized view mode and set it the active
    // child of this view mode.
    if (canMoveSelection)
        // Hand over the most current last mouse events so that the child mode
        // starts with a well-defined state.
        setChild(new MoveSelectionMode(),
                 lastPressEvent, lastDragEvent, lastReleaseEvent);
}
```

### Support for Orthogonal Edge Paths

Class EditMode and its associated child modes provide special support for orthogonal edge paths. Specifically, the following edge-related mouse gestures offer specialized behavior for this kind of paths:

- edge creation (CreateEdgeMode)
- moving edge segments and creating new bends/edge segments (OrthogonalMoveBendsMode)
- moving the ports of an edge (MovePortMode)

Additionally, the following mouse gestures, which have an impact on edge paths, too, also offer specialized behavior:

- resizing nodes (HotSpotMode)
- moving selected nodes/bends (MoveSelectionMode)

[API Excerpt 6.49, “EditMode method related to orthogonal edge path support” \[214\]](#) lists the method that can be used to activate the specialized behavior for all edges of the graph across EditMode and its involved associated child modes (except CreateEdgeMode, see [API Excerpt 6.51, “Methods from CreateEdgeMode to support orthogonal edge paths” \[215\]](#)).

### API Excerpt 6.49. EditMode method related to orthogonal edge path support

```
// Allows to activate support for orthogonal edge paths for all edges across
// EditMode and its involved associated child modes.
void setOrthogonalEdgeRouting(boolean enable)
```

In contrast, when only a specific set of edges should have orthogonal edge path support activated, then either a data provider can be registered with the graph using the look-up key **ORTHOGONAL\_ROUTING\_DKEY**, or the callback method shown in [API Excerpt 6.50, “EditMode method related to orthogonal edge path support” \[214\]](#) can be overridden. Both the data provider and the callback method serve to indicate whether a given edge's path is orthogonal or not. This indication is conveyed using a boolean value.

### API Excerpt 6.50. EditMode method related to orthogonal edge path support

```
// Allows to specify the set of orthogonal edges.
protected boolean isOrthogonalRouting(Edge e)
```

Note that, except CreateEdgeMode, which can be set up using the methods listed in [API Ex-](#)

cerpt 6.51, “Methods from CreateEdgeMode to support orthogonal edge paths” [215]the involved child modes, too, access the data provider registered using the aforementioned look-up key.

### API Excerpt 6.51. Methods from CreateEdgeMode to support orthogonal edge paths

```
// CreateEdgeMode's methods to control orthogonal edge path support.  
boolean isOrthogonalEdgeCreation()  
void setOrthogonalEdgeCreation(boolean orthogonalEdgeCreation)
```

[OrthogonalEdgeViewModeDemo.java](#) demonstrates the support for orthogonal edge paths provided byEditMode and CreateEdgeMode.

## Class NavigationMode

Class [NavigationMode](#) provides a convenient alternative to move the view port so that it shows another part of the model. Instead of using the scroll bars, the clipping is changed by dragging the mouse around the view. While NavigationMode is active no direct mouse interaction with graph elements is possible, i.e., all mouse events involving the left button are consumed for changing the view port.

Note however, that NavigationMode's associated child mode (class [PopupMode](#)) still provides a way of indirectly manipulating graph elements via the items from the presented popup menu.

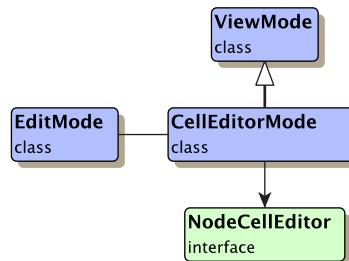
## Class CellEditorMode

Class [CellEditorMode](#) is a view mode descendant that bridges the gap between general mouse-based user interaction with elements in the view and specialized node-related editing capabilities that are provided by means of NodeCellEditor implementations. (See [Node-Related Editing Capabilities](#) for a detailed description of node cell editors.)

For example, if registered with a view'sEditMode instance, this specialized view mode is invoked whenever a so-called "edit node gesture" for a node that has a NodeCellEditor associated is recognized. (The default edit node gesture that is recognized by class EditMode is a mouse double-click on a node.) CellEditorMode then retrieves the actual NodeCellEditor implementation that is associated with the node, and initiates node cell editing by installing the JComponent which is returned by that node cell editor as the node's editor.

Figure 6.42, “Class CellEditorMode's context” [215]depicts the interrelations of CellEditorMode.

**Figure 6.42. Class CellEditorMode's context**



CellEditorMode is also responsible for mapping from nodes to both node cell editors and the objects that are used to hold the node-related data. API Excerpt 6.52, “Instantiation of CellEditorMode” [216]lists the constructors of class CellEditorMode. They can be used to instantiate this view mode so that it either

- shares a single NodeCellEditor implementation for all nodes of a graph or
- uses the given data provider to map from nodes to corresponding NodeCellEditor implementations.

## Class MagnifierViewMode

The DataMap parameter is used to provide the mapping from nodes to corresponding Objects that hold the node-related data which are presented in a JComponent.

### API Excerpt 6.52. Instantiation of CellEditorMode

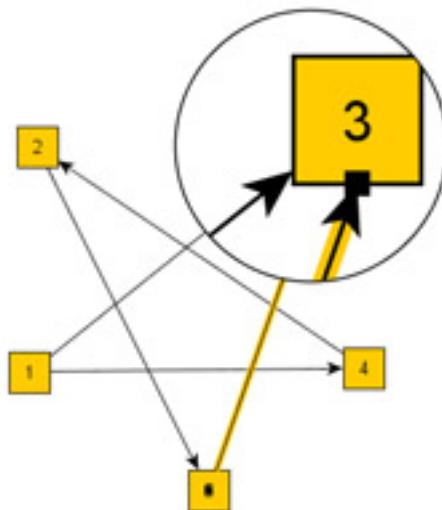
```
// A single NodeCellEditor implementation is shared for all nodes.  
CellEditorMode(NodeCellEditor nodeCellEditor, DataMap userObjectMap)  
  
// The data provider is queried for the NodeCellEditor implementation of a  
// given node.  
CellEditorMode(DataProvider nodeCellEditorProvider, DataMap userObjectMap)
```

The tutorial demo application [SwingRendererDemo.java](#) shows how CellEditorMode is used as a child mode of classEditMode to initiate node editing with different NodeCellEditor implementations.

## Class MagnifierViewMode

Class [MagnifierViewMode](#) is a view mode that provides a magnifying glass "tool" which enables convenient local zooming in a Graph2DView. The magnifying glass follows the mouse movement and supports mouse wheel zooming. Additionally, the radius of the circle where the zoom applies can be changed using the mouse wheel, too.

**Figure 6.43. Magnifying glass "tool"**



[Example 6.21, “Registering a MagnifierViewMode”](#) [216] demonstrates how to configure and install MagnifierViewMode concurrently with other view modes. Among other things, this means that interaction with graph elements as provided by these view modes is still possible while the magnifying glass is active.

### Example 6.21. Registering a MagnifierViewMode

```
// 'view' is of type y.view.Graph2DView.  
  
MagnifierViewMode magnifierMode = new MagnifierViewMode();  
magnifierMode.setMagnifierRadius(100);  
magnifierMode.setMagnifierZoomFactor(2.0);  
  
view.addViewMode(magnifierMode);
```

MagnifierViewMode is shown in tutorial demo application [MagnifierViewModeDemo.java](#).

## Related Classes and Interfaces

## Class HitInfo

Objects of type **HitInfo** collect and store hit-testing information on graph elements (i.e., nodes and edges), their labels, and further parts of graph elements like ports and bends. HitInfo objects use world coordinates and optionally return only the first element or all elements at the given coordinates. Of particular importance when only the first element should be returned is the hit-testing precedence which is applied when collecting the information. The default precedence is:

1. Ports
2. Bends
3. Edge labels
4. Edges
5. Nodes
6. Node labels

This precedence can easily be changed. Alternatively, using the constructors listed in [API Excerpt 6.53, “HitInfo support for hit-testing only a specified graph element type” \[217\]](#) hit-testing can also be performed for a specified graph element type, i.e., only for nodes or only for edges, for example. The type of graph element is specified using the constants defined in class HitInfo.

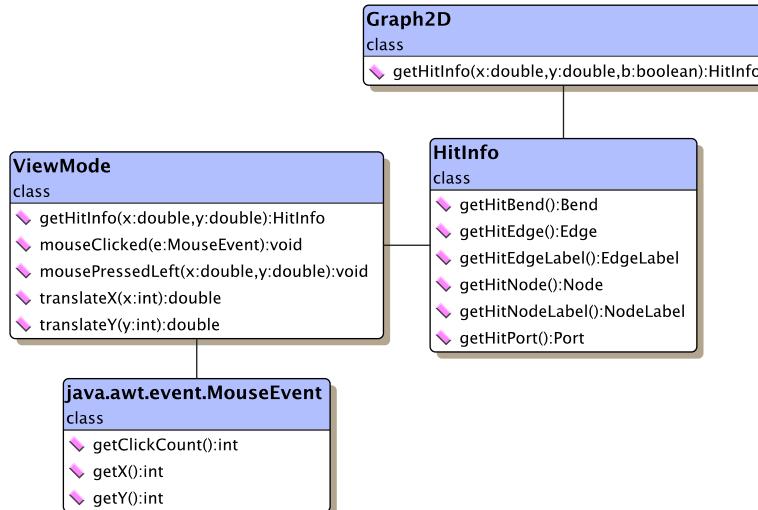
### API Excerpt 6.53. HitInfo support for hit-testing only a specified graph element type

```
// Returns only the first hit graph element of the specified type.
HitInfo(Graph2D graph, double x, double y, int type)

// Optionally returns all hit graph elements of the specified type.
HitInfo(Graph2D graph, double x, double y, int type, boolean firstHitOnly)
```

Using a HitInfo object is straightforward. Following a mouse event within a view, the event's coordinates are translated to world coordinates, with which a HitInfo object can be instantiated. This object then collects and stores all elements that are at the given coordinates. Single elements as well as sets of elements can afterwards be queried using HitInfo's getter methods. [Figure 6.44, “Class HitInfo's context” \[217\]](#) shows class HitInfo and its context.

**Figure 6.44. Class HitInfo's context**



Note that there can be defined independent hit-testing precedences with the graph and any view

mode, since both the view modes and class Graph2D offer a getter method that returns a proper HitInfo object. By default, the `getHitInfo` method defined in class ViewMode delegates all calls to the Graph2D, so that the HitInfo object returned by a view mode is that of the Graph2D that is shown in the view mode's view. To change the hit-testing precedence for a single view mode, the getter method can be customized as shown in [Example 6.22, “Customizing HitInfo precedence in the context of a single view mode” \[218\]](#)

### Example 6.22. Customizing HitInfo precedence in the context of a single view mode

```
public class MyEditMode extends EditMode {
    protected HitInfo getHitInfo(double x, double y) {
        // Instantiate a HitInfo object that collects only the first node at the
        // given coordinates.
        HitInfo hi = new HitInfo(getGraph2D(), x, y, HitInfo.NODE);
        // Set the most current HitInfo object to the one just created.
        setLastHitInfo(hi);
        // Return the HitInfo object.
        return hi;
    }
}
```

Class HitInfo is aware of nested graph hierarchies and accordingly uses the visiting order defined with the HierarchyManager object that governs the given graph.

### Interface MouseInputEditor

A mouse input editor, i.e., an implementation for interface `MouseInputEditor`, is a consumer of mouse events that happen in the view. It enables convenient mouse-based interaction with arbitrary elements such as node realizers and edge realizers, but also implementations of interface `Drawable`. A mouse input editor allows to receive and process mouse events with double precision world coordinates.

The mouse input mode that is active with a view's `EditMode` instance can be configured to forward mouse events using class `Mouse2DEvent` whenever the current mouse coordinates are above elements that have such a mouse input editor associated.

Abstract class `AbstractMouseInputEditor` serves as a base class implementation for interface `MouseInputEditor` that already provides convenient support for state change listeners. The tutorial demo applications

- [MouseInputDemo.java](#) and
- [ScrollingNodeRealizer.java](#)

show how `AbstractMouseInputEditor` can be used to create custom mouse event consumers in the context of `Drawable` and `NodeRealizer` implementations, respectively.

## Customizing an EditMode

The predefined `EditMode` offers default behavior that can be modified in various ways using, e.g., setter methods to allow or disallow certain mouse gestures, or call-back methods to completely change specific aspects of user interaction. Furthermore, class `EditMode`'s behavior can also be modified by exchanging any of its child modes with customized variants.

Among the things that can be disallowed, are, e.g., node and edge creation, node resizing, and bend creation. [Example 6.23, “Generating HTML tool tips” \[218\]](#) shows how alternative HTML tool tips can be realized by overriding the respective call-back methods.

### Example 6.23. Generating HTML tool tips

```
public class FancyHTMLToolTipsEditMode extends EditMode
{
    public FancyHTMLToolTipsEditMode()
    {
        // Enable tool tips for nodes and edges.
        showNodeTips(true);
        showEdgeTips(true);
    }
}
```

```

// Returns the HTML code for a label that nicely prints the number of an
// edge's bends.
protected String getEdgeTip(Edge e)
{
    EdgeRealizer er = view.getGraph2D().getRealizer(e);

    String text = "<html><table><tr>";
    text += "<td align=\"right\"><b>" + er.bendCount() + "</b></td>";
    text += "<td align=\"left\"><em>Bends</em></td>";
    text += "</tr></table></html>";

    return text;
}

// Returns the HTML code for a label that nicely formats the number of a
// node's incoming and outgoing edges.
protected String getNodeTip(Node n)
{
    String text = "<html><table><tr>";
    text += "<td rowspan=\"2\"><em>Edges</em></td>";
    text += "<td align=\"right\"><b>" + n.inDegree() + "</b></td>";
    text += "<td align=\"left\"><em>Incoming</em></td>";
    text += "</tr><tr>";
    text += "<td align=\"right\"><b>" + n.outDegree() + "</b></td>";
    text += "<td align=\"left\"><em>Outgoing</em></td>";
    text += "</tr></table></html>";

    return text;
}

```

**Example 6.24.** “Customizing edge creation” [219] is an excerpt from `CreateEditModeDemo.java`, a tutorial demo application that shows how to customize an `EditMode` by exchanging one of its specialized view modes.

### Example 6.24. Customizing edge creation

```

public class EvenOddCreateEditMode extends CreateEditMode
{
    public boolean isEven(Node n)
    {
        try {
            return (Integer.parseInt(getGraph2D().getLabelText(n)) % 2 == 0);
        }
        catch (Exception ex) {
            return true;
        }
    }

    // Accept the given node as source for an edge if its label is an even number.
    protected boolean acceptSourceNode(Node source, double x, double y)
    {
        return isEven(source);
    }

    // Accept the given node as source for an edge if its label is an odd number.
    protected boolean acceptTargetNode(Node target, double x, double y)
    {
        return !isEven(target);
    }
}

public class CustomizedEditModeDemo extends ViewActionDemo
{
    public CreateEditModeDemo()
    {
        // Register a customized CreateEditMode version.
        editMode.setCreateEditMode(new EvenOddCreateEditMode());
    }
}

```

A customized view mode that is also used for view mode chaining should be designed to return control to its parent in a well-behaved manner. More precisely, the call to reactivate a parent should occur after a mouse release event has been processed. The call should never occur such that the parent receives a mouse drag or mouse release event as the first event.

### Tutorial Demo Code

For further view mode customization examples see the tutorial demo applications:

## Customizing an EditMode

---

- [CreateEdgeModeDemo.java](#)
- [PopupModeDemo.java](#) and [InactiveLayerDemo.java](#)
- [MouseInputDemo.java](#)
- [PortsDemo.java](#)
- [PortCreateEdgeModeDemo.java](#)
- [StateNodeRealizerDemo.java](#) together with [StateNodeRealizer.java](#)

## Events and Listeners

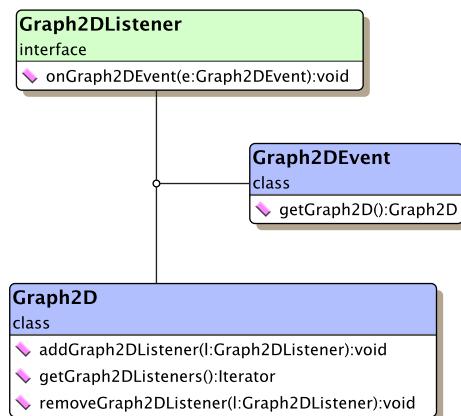
The listeners that are provided with package `y.view` allow for simple notification support with graphical property changes and also selection state modifications. They complement the event notification mechanism from package `y.base` which governs all [structural graph changes](#).

### Graphical Property Changes

Interface `Graph2DListener` notifies interested parties of Graph2D property changes. The information is conveyed using objects of type `Graph2DEvent`, and covers non-structural properties, i.e., label text changes, and realizer type changes.

To receive notifications, `Graph2DListener` implementations have to be properly registered with the `Graph2D` object. [Figure 6.45, “Context for using `Graph2DListener`” \[22\]](#) gives an overview on interface `Graph2DListener`'s role.

**Figure 6.45. Context for using `Graph2DListener`**



Note that the returned property name of `Graph2DEvent`'s `getPropertyName()` method follows the Java Beans naming convention for properties and applies to the event's subject which is returned by method `getSubject()`.

### Selection State Changes

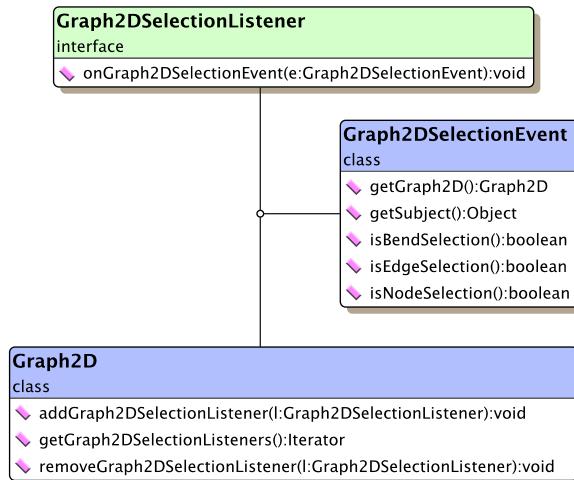
Interface `Graph2DSelectionListener` notifies interested parties of selection state changes with Graph2D elements. The information is conveyed using objects of type `Graph2DSelectionEvent`, which provide various methods to detect the specific type of elements.

To receive notifications, `Graph2DSelectionListener` implementations have to be properly registered with the `Graph2D` object. [Figure 6.46, “Context for using `Graph2DSelectionListener`” \[?\]](#) gives an overview on interface `Graph2DSelectionListener`'s role.

**Figure 6.46. Context for using `Graph2DSelectionListener`**

## Selection State Changes

---

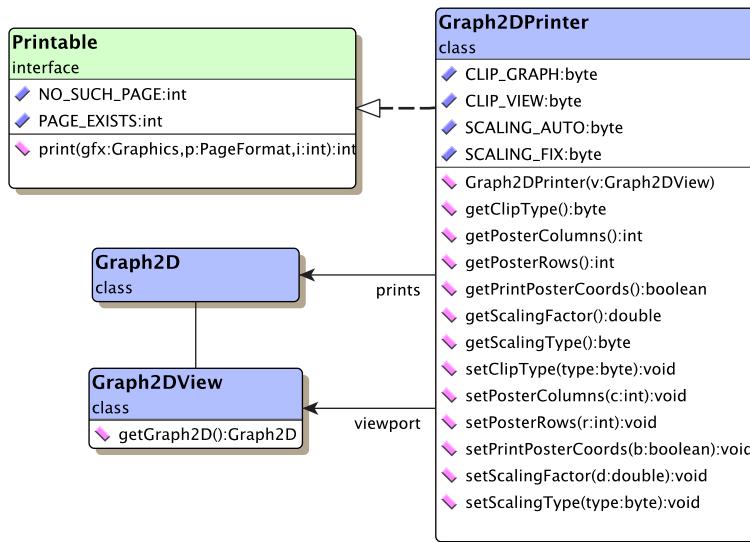


Note that changing the selection state of NodeRealizer and EdgeRealizer objects generates events only when they are bound to a graph element that resides in a Graph2D. The same holds true for changing the selection state of Bend objects. Events are generated only when the edge that is bound to its EdgeRealizer object resides in a Graph2D.

# Printing a Graph's Visual Representation

Class `Graph2DPrinter` provides all functionality that is necessary to print a graph. It supports printing the complete graph or only the part that is visible in the `Graph2DView` containing it. [Figure 6.47, “Printing with Graph2DPrinter” \[223\]](#) gives an overview on class `Graph2DPrinter`'s central role in the printing process.

**Figure 6.47. Printing with Graph2DPrinter**



## Scaling and Clipping

Besides clipping, `Graph2DPrinter` also allows scaling the graph's visual representation for printing. [API Excerpt 6.54, “Policy methods from class Graph2DPrinter” \[223\]](#) lists the methods that are used to control both clipping and scaling policy.

### API Excerpt 6.54. Policy methods from class Graph2DPrinter

```

// Clipping policy getter and setter methods.
byte getClipType()
void setClipType(byte clipType)

// Scaling policy getter and setter methods.
byte getScalingType()
void setScalingType(byte scalingType)
  
```

There are two scaling policies supported: either the number of desired pages in both horizontal and vertical direction is given, then the graph will be scaled automatically to fit the available space. Or, alternatively, the graph's scaling is chosen to be fixed, then the number of necessary pages will be inferred instead. Using either scaling policy the methods from [API Excerpt 6.55, “Methods from class Graph2DPrinter” \[223\]](#) can be used to control the necessary parameters.

### API Excerpt 6.55. Methods from class Graph2DPrinter

```

// Getter and setter for the number of pages.
int getPosterColumns()
void setPosterColumns(int columns)
int getPosterRows()
void setPosterRows(int rows)

// Getter and setter for the scaling factor.
double getScalingFactor()
void setScalingFactor(double scalingFactor)
  
```

## Title Bar and Footer Support

Adding a title bar to the top of a printed page can be accomplished by setting an implementation of interface `Graph2DPrinter.TitleDrawable` with a `Graph2DPrinter` using the corresponding method from API Excerpt 6.56, “Title bar and footer support” [224]. Likewise, by setting an implementation of interface `Graph2DPrinter.FooterDrawable` with a `Graph2DPrinter` a footer can be added to the bottom of a printed page.

Classes `Graph2DPrinter.DefaultTitleDrawable` and `Graph2DPrinter.DefaultFooterDrawable` are the default `TitleDrawable` and `FooterDrawable` implementations that are initially set with a `Graph2DPrinter` object. They offer attributes to control title and footer text, the fonts and text colors thereof, and also background colors for both title bar and footer.

### API Excerpt 6.56. Title bar and footer support

```
// Getter methods.  
Graph2DPrinter.TitleDrawable getTitleDrawable()  
Graph2DPrinter.FooterDrawable getFooterDrawable()  
  
// Setter methods for title bar and footer drawables.  
void setTitleDrawable(Graph2DPrinter.TitleDrawable title)  
void setFooterDrawable(Graph2DPrinter.FooterDrawable footer)
```

## Printing Preview

Class `PrintPreviewPanel` provides a print preview component that can be used in conjunction with `Graph2DPrinter` to create a preview of the printing.

## Creating a Dedicated Printing View

The functionality offered by class `Graph2DPrinter` can be mixed with the same technique used to do an image export of a graph. In particular, Example 6.25, “Opening a dedicated “printing view”” [224] demonstrates how a dedicated “printing” view is opened that shows a given rectangular part of the graph.

### Example 6.25. Opening a dedicated “printing” view

```
Graph2DView replaceCurrentWithPrintingView(Graph2D graph, Rectangle r)  
{  
    // Save the currently active view.  
    Graph2DView originalView = (Graph2DView)graph.getCurrentView();  
  
    // Create a new Graph2DView instance with the graph. This will be the  
    // dedicated view for printing.  
    Graph2DView printingView = new Graph2DView(graph);  
  
    // Set the dimension of the view to the given rectangle.  
    printingView.fitRectangle(r);  
    // Use the Graph2DRenderer instance of the currently active view. (Optional.)  
    printingView.setGraph2DRenderer(originalView.getGraph2DRenderer());  
  
    // Replace the currently active view containing the graph with the “printing”  
    // view.  
    graph.setCurrentView(printingView);  
  
    return originalView;  
}
```

Example 6.26, “Using class `ViewPortConfigurator` to set up a view for printing” [224] shows how this “printing” view is configured using class `ViewPortConfigurator`. Except for the clipping type all configurator settings are left to their respective default values, i.e., a zoom level of 100%, and an additional margin of 15 pixels around the view’s clipping.

### Example 6.26. Using class `ViewPortConfigurator` to set up a view for printing

```
void configurePrintingView(Graph2DView printingView)
{
    ViewPortConfigurator vpc = new ViewPortConfigurator();

    // Register the view with the configurator instance.
    vpc.setGraph2DView(printingView);

    // Only a part of the graph should be printed, hence set the clipping type
    // accordingly.
    vpc.setClipType(ViewPortConfigurator.CLIP_VIEW);

    // Configure the printing view using mainly default values, i.e., zoom level
    // 100%, and 15 pixel margin around the view's clipping.
    vpc.configure(printingView);
}
```

## Tutorial Demo Code

The usage of the print preview is explained in the tutorial demo [PrintPreviewDemo.java](#). The tutorial demo application [ViewActionDemo.java](#) shows the usage of Graph2DPrinter in general.

# Animations for Graph Elements

The yFiles graph visualization library enables high-quality animations and visual effects for graph elements, more precisely their visual representations. Node realizers, edge realizers, but also drawables can be subject for arbitrary effects. Predefined generic animations and effects are, for example:

- moving, resizing, and scaling of nodes
- color transformations with nodes and making nodes blink
- fading in and out, blurring, and explosions that can be used to "decorate" creation and/or removal of both nodes and edges

These effects and animations build upon the general animation framework defined in package `y.anim`, which enables processing of animations both in parallel and in sequence, allows repetitions of animations, and also pausing between animations.

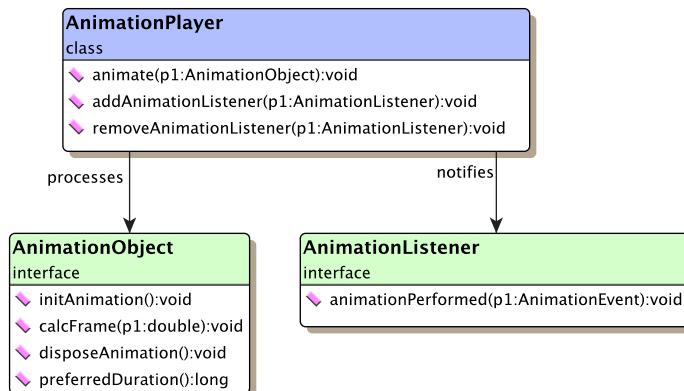
## Animation Framework

Package `y.anim` contains the classes that constitute a general animation framework. Class `AnimationPlayer` is the central instance that governs an animation process. It is given a so-called animation object, an implementation of interface `AnimationObject`, which encapsulates an actual animation respectively a compound of animations.

Effectively, an animation object provides a sequence of so-called "frames" that represent the animation's state as it evolves over time. `AnimationPlayer` progresses the overall animation process by continuously triggering the given animation object to calculate a single new frame that corresponds to the supplied (strictly monotonic) increasing time value. Whenever a new frame has been calculated, the animation player notifies all registered animation listeners, i.e., implementations of interface `AnimationListener`, to update their associated view so that the new frame is displayed.

`AnimationPlayer` also triggers initialization of an `AnimationObject` before the animation process and any necessary cleanup thereafter by calling appropriate methods. [Figure 6.48, “Animation framework central classes” \[226\]](#) shows the role of class `AnimationPlayer`.

**Figure 6.48. Animation framework central classes**



Setup for processing a given animation object and having the progress of the (compound) animation displayed in a view is demonstrated in [Example 6.27, “Playing a \(compound\) animation” \[?\]](#).

### Example 6.27. Playing a (compound) animation

```

void startAnimation(AnimationListener al, AnimationObject ao)
{
}
  
```

```
// Create a new AnimationPlayer that processes the given (compound) animation.
AnimationPlayer player = new AnimationPlayer();
// Register the given animation listener that is to display the (compound)
// animation. Usually, Graph2DView or Graph2DViewRepaintManager are provided
// to this end.
player.addAnimationListener(al);
// Play the given (compound) animation.
player.animate(ao);
}
```

An animation listener is responsible for updating its associated view that hosts an animation. It triggers a repainting of the view each time the animation progresses, i.e., each time a new frame has been calculated. Predefined implementations for interface `AnimationListener` are classes `Graph2DView` and `Graph2DViewRepaintManager`, both from package `y.view`.

When using `Graph2DView` directly as an animation listener, the entire view port is repainted to display the frames of an animation. `Graph2DViewRepaintManager`, in contrast, allows fine-grained control over the repaint area. It provides registration methods for realizers and drawables to advertise their participation in an upcoming animation. Subsequently, `Graph2DViewRepaintManager` invokes repaints on only the union of bounds of these realizers and drawables, which results in performance benefits especially for animations that are confined to a small region.

Animation objects are the actual provider of an animation. Their task is to generate a sequence of frames that represent the animation's state as it evolves over time. An animation's time is defined to run in the range 0.0 to 1.0 from animation start to animation end.

To create animation objects, the factory classes `AnimationFactory` and `ViewAnimationFactory` can be used. The former class returns animation objects that primarily address structural concerns, more precisely, it returns animation objects that can be used to define animations that:

- can be processed concurrently
- can be processed sequentially
- are processed repeatedly
- do nothing, i.e., that effectively introduce a pause

To achieve concurrent and sequential processing of a set of animations, interface `CompositeAnimationObject` provides the necessary methods to add or remove single animation objects to form compound animations. [API Excerpt 6.57, “Methods to define the structure of an animation” \[227\]](#) lists the methods of class `AnimationFactory` that can be used to define the structure of an animation process that is composed of multiple animation objects.

### **API Excerpt 6.57. Methods to define the structure of an animation**

```
// Processing animations concurrently.
static CompositeAnimationObject createConcurrency()
// Processing animations sequentially.
static CompositeAnimationObject createSequence()

// Repeating an animation.
static AnimationObject createRepetition(final AnimationObject ao,
                                         final int repetitions,
                                         final boolean initDisposeRepeatedly)

// Pausing an animation.
static AnimationObject createPause(final long preferredDuration)
```

Class `AnimationFactory` also provides a set of animation objects that can be used as "decorators" for other animations, namely these are ease in and ease out effects. An ease in effect applies a configurable acceleration at the beginning of a given animation, likewise, an ease out effect applies a configurable deceleration at the end. [Example 6.28, “Layout morphing using ease in and ease out effects” \[227\]](#) shows how both ease in and ease out effects can be used to smooth the morphing between two graph layouts.

### **Example 6.28. Layout morphing using ease in and ease out effects**

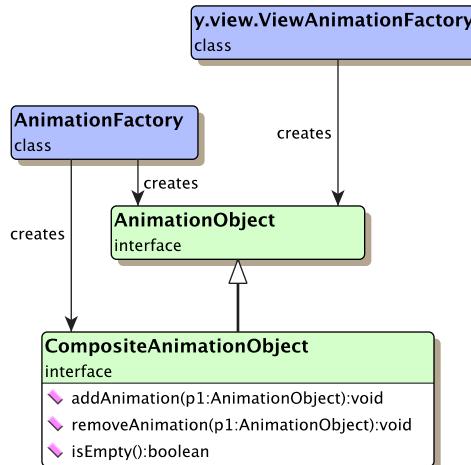
```
void morphLayoutUsingEaseInEaseOutDecoration(Graph2DView view, GraphLayout gl)
```

```
{
    // Create a regular LayoutMorpher and subsequently "decorate" it using ease
    // in and ease out effects.
    LayoutMorpher plainLM = new LayoutMorpher(view, gl);
    AnimationObject easedLM = AnimationFactory.createEasedAnimation(plainLM);

    // Play the animation, i.e., perform smooth layout morphing.
    AnimationPlayer player = new AnimationPlayer();
    player.setFps(240);
    player.addAnimationListener(view);
    player.animate(easedLM);
}
```

Class ViewAnimationFactory, in contrast, can be used to return a variety of high-quality animations and visual effects for graph elements, more precisely their visual representations. [Figure 6.49, “Classes that create animation objects” \[228\]](#) shows the classes that are involved in creating animation objects.

**Figure 6.49. Classes that create animation objects**



By default, ViewAnimationFactory is configured to return animations that trade animation quality for speed. Using the methods and constants listed in [API Excerpt 6.58, “Quality-related methods and constants of class ViewAnimationFactory” \[228\]](#) the quality for animations and effects can conveniently be controlled.

### API Excerpt 6.58. Quality-related methods and constants of class ViewAnimationFactory

```

// Animation quality constants.
static final AnimationQuality DEFAULT;
static final AnimationQuality HIGH_QUALITY;
static final AnimationQuality HIGH_PERFORMANCE;

// Getter/setter methods to control the quality of animations and effects.
AnimationQuality getQuality()
void getQuality(final AnimationQuality quality)
  
```

[Example 6.29, “Creating animations that repeat” \[228\]](#) shows how both classes AnimationFactory and ViewAnimationFactory can be used to create an animation object that combines structural aspect (repetition) and visual effect (blinking).

### Example 6.29. Creating animations that repeat

```

AnimationObject createRepeatedBlinking(NodeRealizer nr,
                                         Graph2DViewRepaintManager rm, int times)
  
```

```
{
    // Create the view animation factory and register the given repaint manager
    // as the responsible animation listener.
    ViewAnimationFactory vaf = new ViewAnimationFactory(rm);
    // The node realizer is automatically added to the repaint manager's list of
    // objects to update.
    // Note that the "blink" animation itself lasts 100 milliseconds.
    AnimationObject blink = vaf.blink(nr, 100);

    // Return an animation object that repeats the "blink" animation for the
    // specified number of times.
    return AnimationFactory.createRepetition(blink, times, true);
}
```

## Callback Mechanism With Generic Animations

Factory class `ViewAnimationFactory` can be used to create generic animations that allow for real-time customization using an advanced interaction scheme. For example, the method presented in [API Excerpt 6.59, “Generic transformation animation” \[229\]](#) makes use of interfaces `Value`, `Value2D`, and `ValueSettable` for a number of parameters.

### API Excerpt 6.59. Generic transformation animation

```
// Creates a generic animation that polls for the actual transformation that
// should be applied to the given node realizer.
AnimationObject transform(final NodeRealizer nr,
    // Each time the 'time' parameter callback provider (see below) is
    // modified, the transformation defined by (a subset of) these four
    // parameters can be customized.
    Value alpha, Value rotation,
    Value2D shear, Value2D scale,
    // Callback provider that is supplied with the current time.
    ValueSettable time,
    final long preferredDuration)
```

The parameter of type `ValueSettable` acts as a callback provider that is supplied with the animation object's current time. Whenever the value of this parameter is set, a client of such a generic transformation animation is free to update other parameters that together define the actual transformation which is to be applied to the given node realizer. Typically, a client can use this interaction scheme as outlined in [Example 6.30, “Using the interaction scheme for real-time customization” \[229\]](#)

### Example 6.30. Using the interaction scheme for real-time customization

```
// Real-time customization of a node realizer transformation.
// The given node realizer is linearly scaled to double its width and then
// return to normal width.
AnimationObject makeMyCustomTransformation(NodeRealizer nr,
                                         Graph2DViewRepaintManager manager)
{
    // Create the initial two-dimensional scale factor.
    MutableValue2D scale = DefaultMutableValue2D.create(1.0, 1.0);
    // Instantiate the callback object that is given to the generic
    // transformation animation.
    MyValueSettable time = new MyValueSettable(scale);

    // Create the view animation factory and configure it to create high-quality
    // effects.
    ViewAnimationFactory vaf = new ViewAnimationFactory(manager);
    vaf.setQuality(ViewAnimationFactory.HIGH_QUALITY);
    // Create an animation that only transforms the 'scale' value.
    return vaf.transform(nr, null, null, null, (Value2D)scale,
                        (ValueSettable)time, 1000);
}

// Callback logic that modifies the scale factor for a node realizer
// transformation.
class MyValueSettable implements ValueSettable
{
    // Holds the two-dimensional scale factor.
    MutableValue2D scale;
    public MyValueSettable(MutableValue2D scale) { this.scale = scale; }
    // Gets called by the animation that applies the generic transformation.
    void set(double v)
    {
        // Update the given scale factor with respect to the supplied value that
        // denotes the current time of the animation.
    }
}
```

```
    scale.setX((v < 0.5) ? (1.0 + 2 * v) : (3.0 - 2 * v));  
}
```

## Tutorial Demo Code

The tutorial demo applications

- [EaseInEaseOutDemo.java](#),
- [FadeInFadeOutDemo.java](#), and
- [AnimationEffectsDemo.java](#) (together with [AnimationEffectsDemoBase.java](#))

demonstrate usage of the animation framework and also many of the effects provided by factory class `ViewAnimationFactory`.

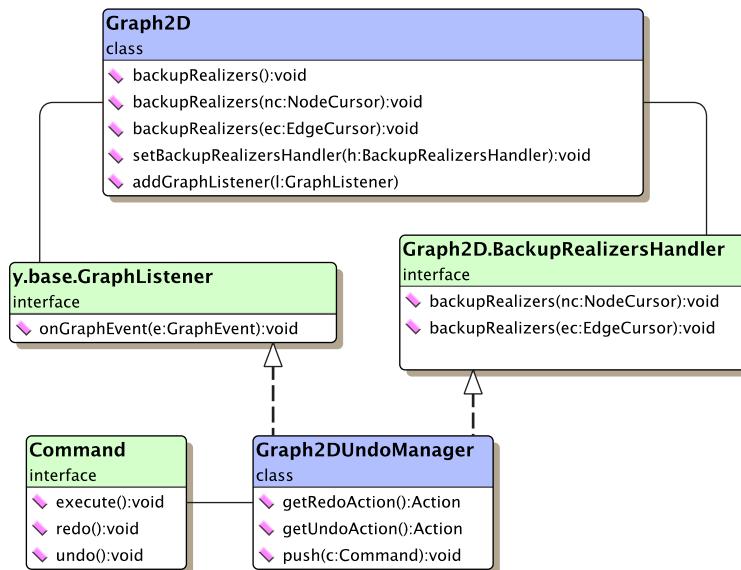
# Advanced Application Logic

The yFiles library provides advanced functionality to be used in an application context. Besides support to undo/redo user actions inside a view, there is also support to generate an animated transformation for graph layout changes.

## Undo/Redo

Class `Graph2DUndoManager` provides undo/redo support for Graph2D changes of both structural and graphical nature. It implements the interfaces `GraphListener` and `Graph2D.BackupRealizersHandler`, and is accordingly registered twice with a Graph2D object.

**Figure 6.50. Class complex around undo/redo functionality**



For graphical changes Graph2DUndoManager relies on the correct [replication behavior of realizers](#) since the actual undo mechanism keeps backup copies of all node and edge realizers that are associated with modified or even deleted graph elements.

### Note

Graph elements that have been deleted and then get reinserted as a result of an undo command are represented by their original objects. The realizer objects that are associated with these reinserted graph elements are also taken from the backup store. However, the graph structure has a different order after a reinsert operation since the graph elements are merely appended to the respective graph data structures.

To reduce the number of actual undo/redo steps, sequences of graph changes can be grouped into single undo/redo commands. So-called pre-event and post-event commands provided by class `GraphEvent` serve as special bracketing indicators. Class `Graph` offers direct support to insert these commands into the undo/redo history, see the methods listed in [API Excerpt 6.60, “Support for undo/redo history bracketing”](#) [231]

### Important

If used, pre-event and post-event commands have to be properly balanced to guarantee correct working of the undo/redo mechanism.

### API Excerpt 6.60. Support for undo/redo history bracketing

## Clipboard

```
// Open the bracket, i.e., insert a pre-event command into the undo/redo
// history.
// The subsequent commands present a single undo/redo step.
firePreEvent()
// Close the bracket, i.e., insert a post-event command into the undo/redo
// history.
firePostEvent()
```

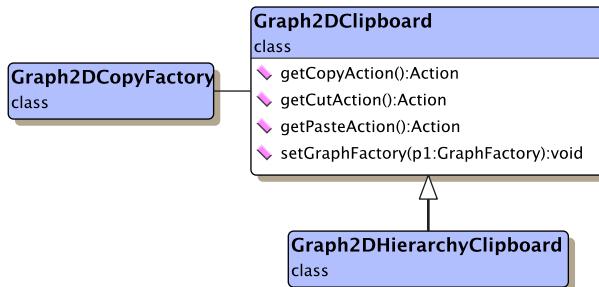
Graph2DUndoManager has getter methods that return complete Swing Actions for integration of both undo and redo operations into an application's context.

[UndoRedoDemo.java](#) is a tutorial demo that shows the yFiles support for undo/redo functionality in an application context.

## Clipboard

Class [Graph2DClipboard](#) provides clipboard functionality for [Graph2D](#) objects. The clipboard can be used to create a copy of selected parts of a Graph2D instance, and can also be used to paste a previously copied subgraph back into a graph again.

**Figure 6.51. Clipboard classes hierarchy**



Graph2DClipboard provides Swing Actions that encapsulate all necessary clipboard functionality, namely the three operations cut, copy, and paste. Copies of graph elements are created by means of a GraphFactory object. By default, this factory is an instance of class [Graph2DCopyFactory](#).

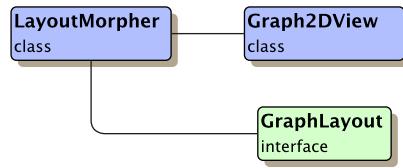
Graph hierarchies, i.e., graphs with an associated [HierarchyManager](#) object, use a specialized clipboard variant, class [Graph2DHierarchyClipboard](#). It supports copy and paste functionality for grouped nodes and nested graph structures in a consistent manner.

[ClipboardDemo.java](#) is a tutorial demo that shows the yFiles clipboard functionality in an application context.

## Layout Morphing

Class [LayoutMorpher](#) is an implementation of the general animation concept defined by interface [AnimationObject](#). It generates a smooth animation that shows a graph's transformation from one layout to another. To this end class LayoutMorpher utilizes an object of type [GraphLayout](#) that is expected to hold positional information for all graph elements from the original graph which is displayed by the associated Graph2DView.

**Figure 6.52. Usage relations for class LayoutMorpher**



LayoutMorpher provides methods to optionally animate changes in the view port's clipping and zoom level, or to end the animation with a specific node being in the center of the view. To start the generated animation method `execute()` has to be called.

Note that the calculated animation highlights changes in the locations of nodes and the locations of control points of edges. In contrast, changes in width or height of any node are not animated.



# Chapter 7. Graph Hierarchies

This chapter describes the yFiles concept of graph hierarchies, a.k.a. hierarchically organized graphs. Discussed are all aspects from managing to user interaction, visual representation, and also layout preparation.

Graph hierarchies can be used to naturally organize large data sets or inherently hierarchically structured data. The yFiles library provides sophisticated support for graph hierarchies that seamlessly integrates with the notion of a graph as described in [Chapter 3, General Concepts in yFiles \[13\]](#).

## Concepts

A graph hierarchy is an augmentation to the graph concept where proper graph structures are allowed to be nested within other graphs to arbitrary depth, forming a hierarchically organized inclusion tree of graphs. At the top of this inclusion tree is the so-called "root graph," the top level graph that is present in any graph hierarchy.

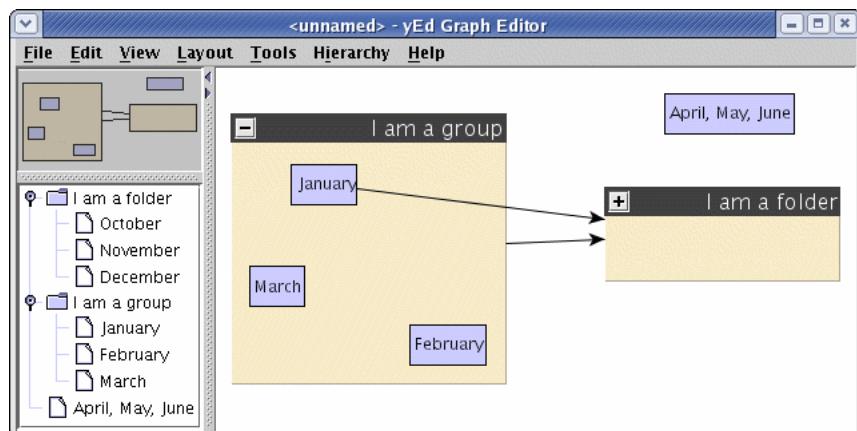
## Grouping and Nesting

The yFiles library knows of two complementary ideas, namely "grouping" and "nesting." Grouping and nesting differ in both structural as well as visual nature. While both approaches make use of proxy nodes that reside in the original graph,

- grouped nodes are stored within the same graph,
- nested graph structures are stored separately.

Thus, grouping is merely of conceptual nature, while nesting is a true hierarchical inclusion of graph structures. Note that grouped nodes can live side by side with nested graph structures. [Figure 7.1, "Grouping and nesting side by side" \[235\]](#) shows grouping and nesting within one graph.

**Figure 7.1. Grouping and nesting side by side**



## Special Proxy Elements

### Group Node and Folder Node

The proxy nodes that are used to either enclose grouped nodes (with grouping) or represent nested graph structures (with nesting) are called "group node" and "folder node," respectively.

The nested graph that is associated with a folder node is called the "inner graph" of this folder node. The graph where the folder node resides in is called the "parent graph" of the inner graph, and is hierarchically one level above the inner graph. Note that a folder node serves as a means to link inner graph to parent graph. A folder node is also called "anchor node."

From a conceptual point of view, grouped nodes that are contained in a group node can be thought of as being hierarchically one level below the graph where the group node resides in.

Technically, however, these nodes belong to the same graph as the group node.

The difference in semantics of grouping and nesting is also reflected by the visual representation of group nodes and folder nodes. While grouped nodes contained in a group node are visible within the graph containing the group node itself, the nested graph structure that is associated with a folder node is hidden. However, a folder node provides access to its inner graph at any time, e.g., via a mouse double click that occurs within its area.

[Figure 7.1, “Grouping and nesting side by side” \[235\]](#) shows both a group node and a folder node within the same graph. The content of the folder node cannot be seen in the graph view, but is listed in the tree-like view in the lower left corner of the application's window.

### Inter-Edge

Besides group node and folder node there is also a proxy edge introduced that is used to replace an edge that connects nodes on different hierarchy levels. Such an edge would actually connect nodes in different graphs, which is not allowed. It is therefore redirected in a manner so that it connects nodes that are within the same graph.

The resulting redirected edge, which serves as a proxy for the original edge, is a so-called "inter-edge."

# Working With Graph Hierarchies

The possibilities that are offered by a hierarchically organized graph can conveniently be used to improve a user's perception of a graph structure. For example, large data sets can be presented in a concise manner, and inherently hierarchically structured data can be modeled naturally.

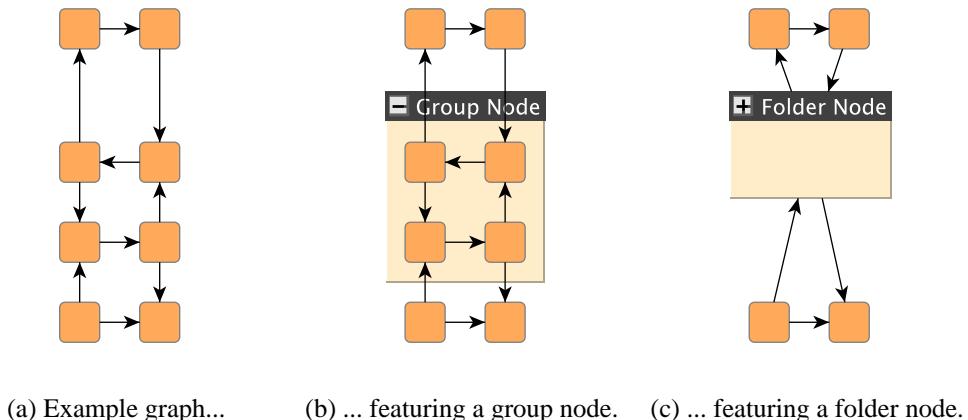
## Comparing the Concepts

Grouping is a means to visually cluster a graph structure and have nodes that somehow belong together be combined within a group node. Nesting, in contrast, performs a real clustering in the sense that a folder node's inner graph no longer bloats the original graph structure, but is stored as a separate graph.

[Figure 7.2, “Differences between grouping and nesting” \[237\]](#) presents the two concepts applied to the same sample graph side by side. The apparent differences between grouped nodes and nested nodes can easily be recognized at first sight.

Moreover, inter-edge behavior is also demonstrated. While all edges connect to their original end nodes with grouped nodes, the same edges are remodeled as inter-edges with nested nodes. The inter-edges connect to the folder node that contains the inner graph where their end nodes (the “real” nodes) reside in.

**Figure 7.2. Differences between grouping and nesting**

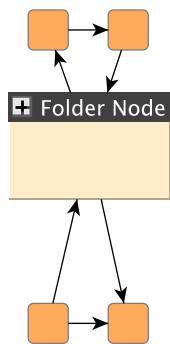


## Nesting Characteristics

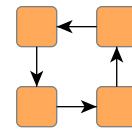
By nesting subgraphs within folder nodes a graph can easily be decomposed into a reduced version of itself and a number of inner graphs that are nicely organized. This characteristic is a perfect means to take the complexity out of a graph and present a more cleaned-up view to the user. At the same time, computation time of many algorithms is also reduced, since they actually handle less data.

[Figure 7.3, “Folder node and its inner graph” \[237\]](#) shows to the right the inner graph that, conceptually, is nested within the folder node to the left.

**Figure 7.3. Folder node and its inner graph**



(a) Example graph with folder node...



(b) ... and the folder node's inner graph.

# Managing Graph Hierarchies

The management of a graph hierarchy is a multi-faceted undertaking that is handled by a small number of classes. Adding support for grouping and nesting to a "flat" graph can easily be achieved with little effort.

## Class HierarchyManager

Class [HierarchyManager](#) maintains the model that is behind graph hierarchies, and manages all changes relevant to this model. Among other things, the HierarchyManager's responsibilities include:

- Creation and removal of group nodes and folder nodes
- Grouping and ungrouping nodes
- Nesting and un-nesting of graph structures
- Creation of a folder node's inner graph
- Handling of inter-edges
- Maintenance of the node visiting order for both hit-testing and drawing the graph elements
- Provision of graph element information

Adding support for both grouping and nesting to a graph is achieved by creating a HierarchyManager object that is associated with a given graph, which is considered the hierarchy's root graph. [Example 7.1, "Adding support for grouping and nesting to a graph" \[239\]](#) demonstrates how to add a HierarchyManager object to a yet "flat" graph, i.e., one that lacks support for grouping or nesting.

### Important

There must not be more than one HierarchyManager associated with a given graph.

### Example 7.1. Adding support for grouping and nesting to a graph

```
// 'rootGraph' is of type y.base.Graph2D.  
// Create a hierarchy manager with the given graph.  
HierarchyManager hm = new HierarchyManager(rootGraph);
```

## Creating Grouped Nodes

Conceptually, grouped nodes are one level below their containing group node in the graph hierarchy. Technically, however, they are in the same graph as their enclosing group node. [Example 7.2, "Grouping nodes" \[239\]](#) shows the creation of a group node and how to add content to it.

### Example 7.2. Grouping nodes

```
// 'hm' is of type y.view.hierarchy.HierarchyManager.  
  
NodeList nl = new NodeList();  
nl.add(rootGraph.firstNode());  
  
// Create a new node in the root graph that is a group node.  
Node groupNode = hm.createGroupNode(rootGraph);  
  
// Add a single node to the group.  
hm.groupSubgraph(nl, groupNode);
```

## Creating Nested Graphs

Conceptually, a nested graph structure is contained within its folder node. In the graph hierarchy, it is one level below its folder node, and is accordingly called the inner graph of the folder node. Technically, however, a nested graph structure is a proper graph that is stored separately

from its parent graph, i.e., the graph where the folder node resides in.

The folder node within the parent graph serves as an anchor for the nested graph structure. This is why a folder node is also called an anchor node sometimes. [Example 7.3, “Nesting nodes” \[240\]](#) shows the creation of a folder node and how to add content to it. Also, extracting a nested graph structure from a folder node is shown, too.

### Example 7.3. Nesting nodes

```
// 'hm' is of type y.view.hierarchy.HierarchyManager.

NodeList nl = new NodeList();
nl.add(rootGraph.firstNode());

// Create a new node in the root graph that is a folder node.
Node folderNode = hm.createFolderNode(rootGraph);

// Add a single node to the folder.
hm.foldSubgraph(nl, folderNode);

// Count the inner graph's nodes.
int innerGraphNodeCount = hm.getInnerGraph(folderNode).N();

// Move the folder node's content up one level back to the parent graph.
hm.unfoldSubGraph(hm.getInnerGraph(folderNode), nl);
```

Whenever the HierarchyManager is asked to populate a folder node, the involved nodes are removed from their original graph and created anew inside the folder node's inner graph.

### Handling Inter-edges

In a hierarchically organized graph that contains folder nodes, there can also occur edges that connect nodes from inner graphs to nodes from other graphs within the graph hierarchy. Generally, these edges have to be remodeled to connect nodes that are on the same hierarchy level, since an edge whose nodes are in different graphs is not legal.

Remodeling an edge to an inter-edge that represents its original is automatically done by the HierarchyManager. Both edge ends are promoted to the nearest common graph up in the hierarchy that contains all folder nodes of the involved inner graphs.

The HierarchyManager afterwards provides information on the “real” source and target of the inter-edge. See [API Excerpt 7.1, “Methods relating to inter-edges” \[240\]](#) for the methods to get the real source and target of an inter-edge.

### API Excerpt 7.1. Methods relating to inter-edges

```
// Returns the real source node associated with the given inter-edge.
Node getRealSource(Edge interEdge)

// Returns the real target node associated with the given inter-edge.
Node getRealTarget(Edge interEdge)
```

### Node Visiting Order

The HierarchyManager instance that has been added to a graph is also responsible for maintaining the node visiting order that is used, e.g., for drawing or for hit-testing the graph elements. [API Excerpt 7.2, “Methods to control the node visiting order” \[240\]](#) lists methods that provide a way to affect the node visiting order by changing the order of children of either group node or folder node.

### API Excerpt 7.2. Methods to control the node visiting order

```
// Makes the given node the first child of its parent.
void moveToFirst(Node childNode)

// Makes the given node the last child of its parent.
void moveToLast(Node childNode)
```

## Important

The visiting order that is maintained by class HierarchyManager is not preserved when the hierarchically organized graph is saved to a file.

To preserve the visiting order between file export and import, all HierarchyManager operations that affect the node order have to be accompanied by corresponding [Graph](#) operations on the particular graph (root graph or nested graph). See [Table 7.1, “Corresponding node order methods” \[241\]](#)

**Table 7.1. Corresponding node order methods**

HierarchyManager	Graph
<code>moveToFirst(Node)</code>	<code>moveToFirst(Node)</code>
<code>moveToLast(Node)</code>	<code>moveToLast(Node)</code>

## Traversal Policies

Class HierarchyManager offers two different node traversal orders which can be applied to a hierarchically organized graph:

- Pre-traverse visits the parent before the children, and the children themselves from first to last.
- Post-visit visits the children before the parent, and the children themselves from last to first.

[API Excerpt 7.3, “Node traversal methods” \[241\]](#) lists HierarchyManager's methods for both traversal policies.

### API Excerpt 7.3. Node traversal methods

```
// Node traversal used for drawing.
void postTraverse(HierarchyManager.NodeVisitor visitor)
void postTraverse(Node rootNode, HierarchyManager.NodeVisitor visitor)

// Node traversal used for hit-testing.
void preTraverse(HierarchyManager.NodeVisitor visitor)
void preTraverse(Node rootNode, HierarchyManager.NodeVisitor visitor)
```

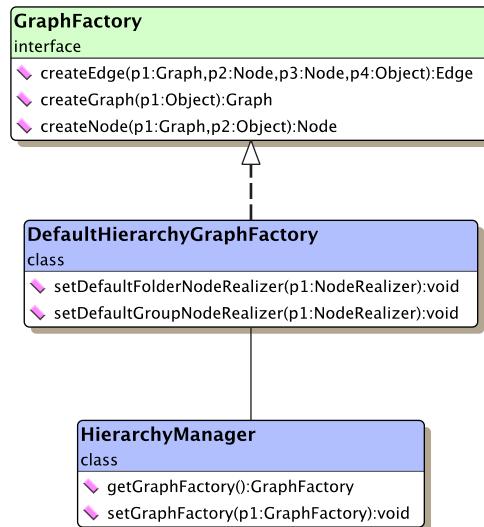
Pre-traverse is used, e.g., for drawing the nodes of a hierarchically organized graph, since it makes sure that children are drawn atop their parents. Post-traverse on the other hand, is useful for hit-testing the graph elements, since elements atop of others should receive hit events first.

## Class DefaultHierarchyGraphFactory

Class [DefaultHierarchyGraphFactory](#) is used by HierarchyManager objects to have new graph elements within the hierarchically organized graph created and properly configured. The graph factory handles creation of both normal graph elements as well nested graph structures. It also creates and configures the corresponding proxy nodes, i.e., either group nodes or folder nodes.

Nested graph structures created by DefaultHierarchyGraphFactory are of type [Graph2D](#), the realizer implementations taken for group nodes and folder nodes can be controlled using appropriate methods. By default, class [GroupNodeRealizer](#) is used for both group nodes and folder nodes, since it can render both presentations. See also the section called “[Node Realizers” \[243\]](#)

**Figure 7.4. Class DefaultHierarchyGraphFactory**



Whenever a new graph is created by this graph factory, it will inherit all listeners that are registered with its parent graph, i.e., all `GraphListener`, `Graph2DListener`, and `Graph2DSelectionListener` references are copied.

To control the creation of graph objects the instance of `DefaultGraphFactory` that is used can be replaced with a customized factory using the appropriate method of class `HierarchyManager`.

## Tutorial Demo Code

[HierarchyDemo.java](#) is an extensive tutorial demo application that discusses all aspects of graph hierarchies. The [README file](#) gives an overview of the application's features and functionality.

# Node Realizers

Graph hierarchies make use of special proxy elements that indicate either the grouping of a collection of nodes, or the nesting of another graph structure. These proxies are called group nodes and folder nodes, respectively, and are rendered by specialized node realizers.

## Class GroupNodeRealizer

Class [GroupNodeRealizer](#) provides the visual presentation for both group nodes and folder nodes. Common to their presentation is a title bar that presents the node's label, their differences are summarized in [Table 7.2, “GroupNodeRealizer rendering and behavior” \[243\]](#)

**Table 7.2. GroupNodeRealizer rendering and behavior**

Group node	Folder node
displays open state icon (stylized minus sign)	displays closed state icon (stylized plus sign)
size is automatically determined to entirely enclose its grouped nodes	size is controlled normally
is not responsible for drawing its grouped nodes	optionally displays a scaled version of the nested graph structure

GroupNodeRealizer can be in one of two states, either open or closed, where

- open means that the realizer renders a group node, and
- closed means that it renders a folder node.

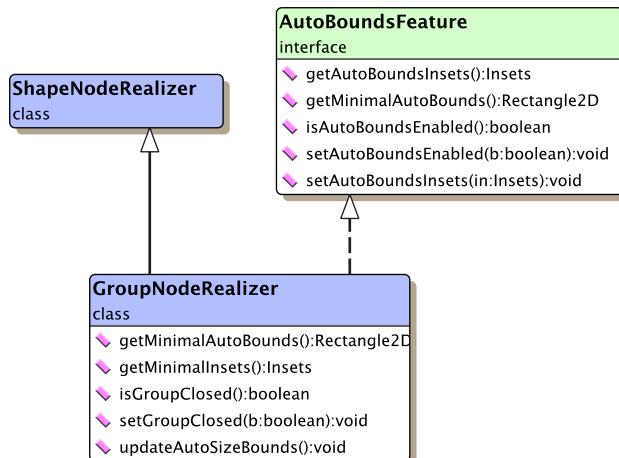
The respective state a GroupNodeRealizer is in, is directly reflected by the displayed state icon.

Using the services of its static inner class [GroupNodeRealizer.StateChangeListener](#) GroupNodeRealizer's state can automatically be updated. In addition, the current state can also be controlled using the methods listed in [API Excerpt 7.4, “Class GroupNodeRealizer's state methods” \[243\]](#)

### API Excerpt 7.4. Class GroupNodeRealizer's state methods

```
// Getter and setter methods.
boolean isGroupClosed()
void setGroupClosed(boolean closed)
```

**Figure 7.5. Class GroupNodeRealizer**



### Resizing Policy

Class GroupNodeRealizer implements [interface AutoBoundsFeature](#), which by default is enabled whenever a GroupNodeRealizer object is in open state, i.e., a group node is rendered.

Auto bounds feature is responsible for adjusting the size and location of the rendered group node. It automatically determines the group node's bounds on the basis of the enclosing rectangle of the grouped nodes that a group node contains. At the same time, changes to the bounds using the appropriate methods are completely ignored.

#### Note

Auto bounds feature applies to group nodes only, and is of no relevance to folder nodes.

To explicitly change a group node's bounds, the auto bounds feature has to be disabled so that the change can have effect, and again enabled afterwards. The group node's bounds is then adopted as long as the enclosing rectangle of the grouped nodes is entirely contained. If it is not, then the bounds is reset to this rectangle.

### Interface AutoBoundsFeature

Interface [AutoBoundsFeature](#) serves as a means to have the size and location of a node realizer's rendering be controlled by factors other than explicit assignment. In particular, it is responsible to automatically update the bounds whenever such a factor changes. Also, direct manipulation of the bounds using the respective node realizer methods is ignored.

AutoBoundsFeature allows the definition of a minimal bounds for its node realizer, and defines the notion of insets that are used to compensate for any width and height that exceeds this minimal bounds.

# View Implementations

## Class HierarchyJTree

Class [HierarchyJTree](#) is an implementation of a view that displays the nodes of a hierarchically organized graph in a tree-like fashion. The presentation resembles a directory structure with normal nodes being leaves, group nodes being opened directories, and folder nodes being closed ones.

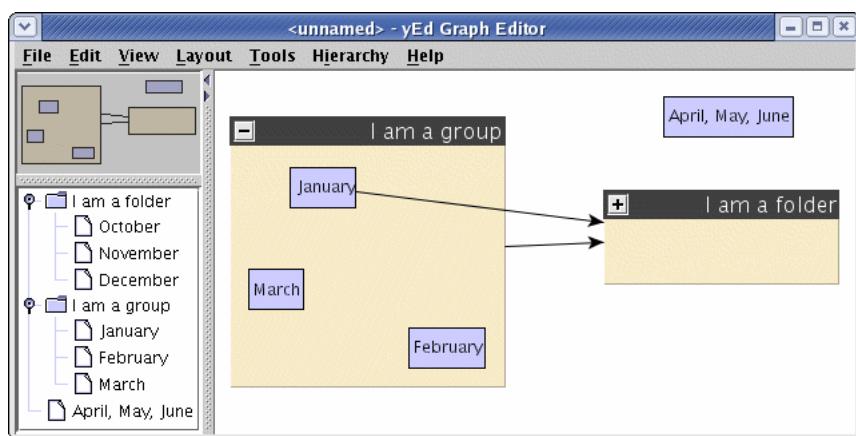
### Note

HierarchyJTree is perfectly suited to display "flat" graphs also.

HierarchyJTree uses the first node label as the descriptive text to be displayed. Node labels that render HTML code result in rendered HTML in the tree-like view, too, since HierarchyJTree utilizes a [DefaultTreeCellRenderer](#) object for this task.

[Figure 7.6, “Graph view provided by HierarchyJTree” \[245\]](#) shows a HierarchyJTree placed in the lower left corner inside an application's window.

**Figure 7.6. Graph view provided by HierarchyJTree**



Class HierarchyJTree's presentation of the selection state indication is at any time synchronized with the selection state as present in the hierarchically organized graph.

## Related Classes

### Class DefaultNodeChangePropagator

Using the services of class [DefaultNodeChangePropagator](#) HierarchyJTree also gets notified whenever a group node is changed to a folder node or vice-versa. Such modifications to the graph hierarchy result in a label change of the respective node in the tree-like view.

### Class HierarchyTreeTransferHandler

Class [HierarchyTreeTransferHandler](#) can be used to add drag-and-drop functionality to a HierarchyJTree. It is a proper `javax.swing.TransferHandler` that enables changing the tree-like representation of a hierarchically organized graph by means of simple mouse drag gestures. Each such change is directly converted to an actual change of the graph hierarchy itself.

[Example 7.4, “Enabling drag-and-drop for a HierarchyJTree” \[245\]](#) shows how to enable the drag-and-drop gesture on the content of the HierarchyJTree. The HierarchyManager that is associated with the hierarchically organized graph is given to HierarchyTreeTransferHandler at creation time.

### Example 7.4. Enabling drag-and-drop for a HierarchyJTree

```
// 'hierarchyManager' is of type y.view.hierarchy.HierarchyManager.  
HierarchyJTree tree = new HierarchyJTree(hierarchyManager);
```

## Related Classes

---

```
tree.setDragEnabled(true);
// The given HierarchyManager is responsible for the hierarchically organized
// graph that is presented in the HierarchyJTree.
tree.setTransferHandler(new HierarchyTreeTransferHandler(hierarchyManager));
```

# User Interaction

Group nodes and folder nodes introduce new user interaction aspects with hierarchically organized graphs. These aspects are reflected by appropriately enhanced view mode implementations.

The view modes that support both grouping and nesting can be activated by simply registering an instance of `class HierarchyEditMode` as shown in [Example 7.5, “Registering an edit mode that provides support for graph hierarchies” \[247\]](#)

## Example 7.5. Registering an edit mode that provides support for graph hierarchies

```
public class GraphHierarchyDemo
{
    protected Graph2DView view;
    protected HierarchyManager hm;
    protected EditMode editMode;

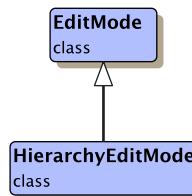
    public GraphHierarchyDemo()
    {
        view = new Graph2DView();
        // Add support for grouping and nesting to the graph.
        hm = new HierarchyManager(view.getGraph2D());

        // Register an augmented edit mode that provides support for graph
        // hierarchies.
        editMode = new HierarchyEditMode();
        view.addViewMode(editMode);
    }
}
```

## Class HierarchyEditMode

Class `HierarchyEditMode` takes the role of `class EditMode` in hierarchically organized graphs. It has associated a number of child modes that provide adequate editing capabilities for graph hierarchies.

**Figure 7.7. Class HierarchyEditMode**



[Figure 7.7, “Class HierarchyEditMode” \[247\]](#) shows class `HierarchyEditMode`'s inheritance structure which is typical for the associated child modes, too. Each of the hierarchy view modes listed in [Table 7.3, “Specialized view modes associated with class HierarchyEditMode” \[247\]](#) inherits from the corresponding view mode from package `y.view` that handles similar functionality in graphs without support for grouping or nesting.

**Table 7.3. Specialized view modes associated with class HierarchyEditMode**

Classname	Description
<code>HierarchyCreateEdgeMode</code>	Handles edge creation in hierarchically organized graphs. In particular, edge creation involving group nodes or their grouped nodes is taken special care of. The group node's border area is reserved for creation of edges connecting to the group node itself. This also implies that bend creation is not possible at a group node's border.

## Tutorial Demo Code

Classname	Description
HierarchyHotSpotMode	Handles node resizing in hierarchically organized graphs. In particular, auto bounds-constrained group node resizing policies are taken special care of.
HierarchyMoveSelectionMode	Handles moving a collection of selected graph elements in hierarchically organized graphs. In particular, moving group nodes is taken special care of. Whenever a group node is moved, all its grouped nodes are moved, too. This mode does also provide support to add further nodes to a group node.
HierarchySelectionBoxMode	Handles creating a rectangular box in the view and changes the selection state of graph elements contained in that box in hierarchically organized graphs. In particular, group node selection is taken special care of. A group node's selection state is only changed when it is entirely contained in the selection box.

## Tutorial Demo Code

The class fragment that is presented in [Example 7.6, “Toggling a GroupNodeRealizer's state”](#) [?] briefly shows how a view mode would change the state of a GroupNodeRealizer-rendered node in response to a mouse click on the node's state icon.

### Example 7.6. Toggling a GroupNodeRealizer's state

```
class ToggleOpenClosedStateViewMode extends ViewMode
{
    HierarchyManager hm;

    public void mouseClicked(MouseEvent ev)
    {
        // Convert the mouse event's coordinates from view to world coordinates.
        double x = translateX(ev.getX());
        double y = translateY(ev.getY());

        // Retrieve the node that has been hit at the location.
        Node v = getHitInfo(ev).getHitNode();

        // Test if the node is rendered by GroupNodeRealizer.
        GroupNodeRealizer gnr = getGroupNodeRealizer(v);
        if (gnr != null)
        {
            // Get the state label.
            NodeLabel stateLabel = gnr.getStateLabel();
            // Test, if the mouse event occurred on the state icon.
            if (stateLabel.getBox().contains(x, y))
            {
                // Retrieve the HierarchyManager of the hierarchically organized graph.
                hm = HierarchyManager.getInstance(view.getGraph2D());

                if (hm.isFolderNode(v))
                    openFolder(v); // Invokes hm.openFolder(v) ultimately...
                else
                    closeGroup(v); // Invokes hm.closeGroup(v) ultimately...
            }
        }
    }
}
```

[HierarchyDemo.java](#) provides further view mode implementations in the context of hierarchically organized graphs.

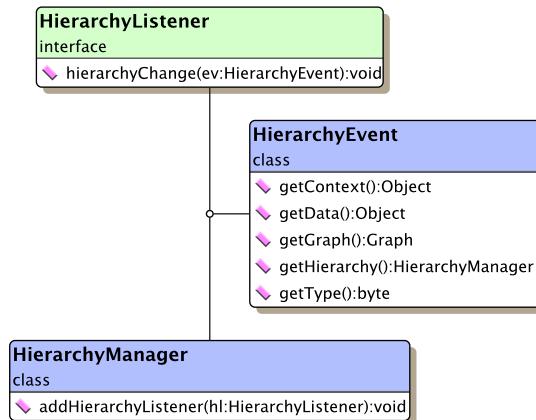
# Events and Listeners

## Hierarchy Changes

Interface `HierarchyListener` notifies interested parties of changes in a hierarchically organized graph. The information is conveyed using objects of type `HierarchyEvent`, and covers changes that relate to hierarchy level modifications of graph elements.

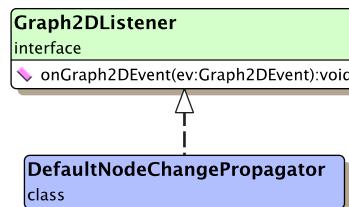
To receive notifications, `HierarchyListener` implementations have to be properly registered with a `HierarchyManager` object. [Figure 7.8, “Context for using `HierarchyListener`” \[249\]](#) gives an overview on interface `HierarchyListener`'s role.

**Figure 7.8. Context for using `HierarchyListener`**



Class `DefaultNodeChangePropagator` can additionally be registered with the root graph of a hierarchically organized graph to notify interested parties of node label changes.

**Figure 7.9. Context for using `DefaultNodeChangePropagator`**



# Automatic Layout

The `HierarchyManager` object that is associated with a hierarchically organized graph maintains all data describing the graph structure as well as a number of additional aspects. Since this information is not directly accessible for a layout algorithm, it has to be properly prepared before a layout can be calculated.

## Class GroupLayoutConfigurator

Class `GroupLayoutConfigurator` converts relevant parts of the `HierarchyManager`'s model into data accessor-based information that is understood by layout algorithms from the `yFiles` library package `y.layout`.

In particular, this information includes which nodes are group nodes, and the auto bounds and insets settings for each group node. [API Excerpt 7.5, “Convenience methods from class GroupLayoutConfigurator” \[250\]](#) shows the methods that are used to prepare and restore all relevant graph hierarchy aspects.

**API Excerpt 7.5. Convenience methods from class GroupLayoutConfigurator**

```
// Prepares all relevant information for a layout algorithm.  
void prepareAll()  
  
// Restores all group node specific sizes and insets after a layout algorithm  
// has been run, and updates the model accordingly.  
void restoreAll()
```

[Example 7.7, “Layout preparation with graph hierarchies” \[250\]](#) demonstrates how to use `GroupLayoutConfigurator`'s methods.

### Example 7.7. Layout preparation with graph hierarchies

```
// 'graph' is of type y.view.Graph2D.  
  
HierarchicGroupLayouter hgl = new HierarchicGroupLayouter();  
GroupLayoutConfigurator glc = new GroupLayoutConfigurator(graph);  
  
// Prepares all relevant information for a layout algorithm.  
glc.prepareAll();  
  
new BufferedLayouter(hgl).doLayout(graph);  
  
// Restores all group node specific sizes and insets after a layout algorithm  
// has been run, and updates the model accordingly.  
glc.restoreAll();
```

---

# Chapter 8. Input and Output

This chapter presents various file formats that are supported as graph exchange, respectively, graph export formats with yFiles. It also explains which classes to use for reading and writing graph structure data and how to prepare a graph for image export.

## Graph Exchange and Graph Export Formats

Package `y.io` provides the necessary functionality for both reading and writing graph structure data from and to a file, respectively. All of the classes that implement input/output functionality work on an instance of type `Graph2D`. As a consequence, writing out a graph is not restricted to the graph structure itself, but also supports the complete set of visual attributes, i.e., in particular, everything related to the visual representation of graph elements.

Directly supported file formats are GML, XGML, TGF, and YGF for graph exchange, and GIF and JPG for graph export (image export). Extension packages provide further file formats like GraphML for graph exchange, and SVG, PDF, EMF, SWF, and EPS for graph export (vector graphics export). Via the Java Image I/O API (package `javax.imageio`, available with Java 2 Standard Edition version 1.4 and higher) additional file formats for graph export, for example PNG, are also accessible. Table 8.1, “Graph exchange and export file formats” [251] lists the features of all yFiles graph exchange and export file formats.

**Table 8.1. Graph exchange and export file formats**

Format Name (Abbreviation)	Human Readable	Nested Graph Structure	Visual Attributes	Type	Extension Package	Note
GML	Yes	Yes	Yes	Exchange	No	
XGML	Yes	Yes	Yes	Exchange	No	XML-ish variant of GML.
TGF	Yes	No	No	Exchange	No	
YGF	No	Yes	Yes	Exchange	No	Proprietary; non-standard; binary.
GIF	No		Yes	Export (Bitmap)	No	
JPG	No		Yes	Export (Bitmap)	No	
GraphML	Yes	Yes	Yes	Exchange	Yes	XML-based; extension package is free of charge.
SVG	Yes		Yes	Export (Vector Graphics)	Yes	XML-based; extension package is free of charge.
PDF	No		Yes	Export (Vector Graphics)	Yes	
SWF	No		Yes	Export (Vector Graphics)	Yes	
EMF	No		Yes	Export (Vector Graphics)	Yes	
EPS	No		Yes	Export (Vector Graphics)	Yes	
PNG	No		Yes	Export (Bitmap)	No	Accessible via the Java Image I/O API available with J2SE 1.4 and higher.

The graph exchange formats mainly concentrate on the graph structure, which also includes nes-

ted graph structure. They do not support visual aspects that are not associated with the graph, like, e.g., zoom level, selection state, or a background image. In contrast, the graph export formats fully support these visual aspects.

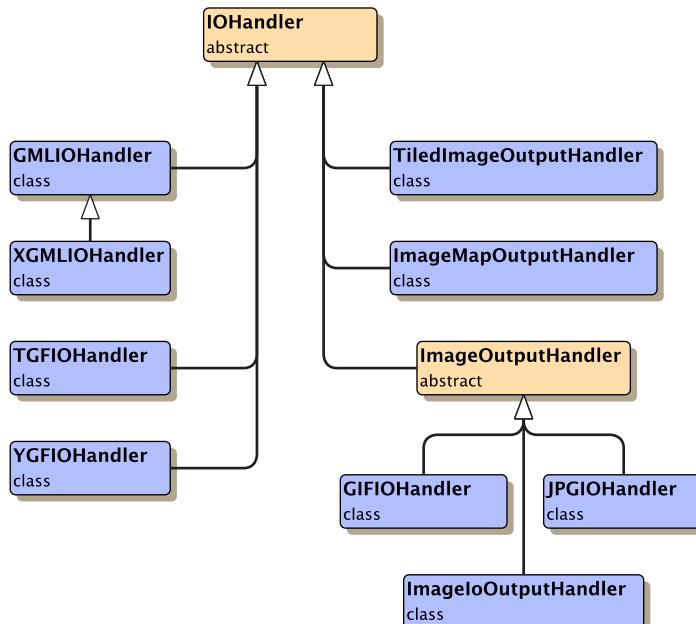
The graph exchange formats (except TGF) do, however, support all visual attributes that are directly related to the representation of graph elements. First and foremost, this means that most of the information encoded by classes `NodeRealizer` and `EdgeRealizer` is written to/read from file.

Reading and writing graph structures from/to file is described in [Reading and Writing Graph Structure Data \[255\]](#) the section [Exporting a Graph's Visual Representation \[274\]](#) explains image export of a graph's visual representation.

## Input/Output Handlers

Common to reading/writing of all file formats is the use of so-called input/output handlers, i.e., subclasses of abstract class `IOHandler`. The hierarchy of these input/output handlers is shown in [Figure 8.1, “The class hierarchy in package y.io” \[252\]](#). Observe that the subclasses on the right side provide only for image export facilities, i.e., output of a graph's visual representation, while those on the left side provide for true graph structure exchange, i.e., input and output.

**Figure 8.1. The class hierarchy in package y.io**



### Note

To stay consistent with the naming scheme the classes `GIFIOHandler` and `JPGIOHandler` should have been named `GIFOutputHandler` and `JPGOutputHandler`, respectively.

Abstract class `IOHandler` provides the basic mechanism to read/write a graph from/to a file. This mechanism comprises methods to read a graph from a stream and to write a graph to a stream, respectively. Also included are getter methods for a format-specific descriptive string and file name extension. [API Excerpt 8.1, “Abstract methods from class IOHandler” \[252\]](#) lists these four most essential methods. They are all defined abstract, so descendants of class `IOHandler` have to implement them appropriately.

### API Excerpt 8.1. Abstract methods from class IOHandler

```

abstract void read(Graph2D graph, InputStream in)
abstract void write(Graph2D graph, OutputStream out)
  
```

```
abstract String getFileFormatString()
abstract String getFileNameExtension()
```

A subclass that implements only output functionality has to override the `IOHandler.read(Graph2D, InputStream)` method with an empty body. Additionally, it should provide its own `canRead()` method which should always return `false`.

### Note

By default, the methods `canRead()`, `canWrite()`, and `canWriteSubset()` from abstract class `IOHandler` all return `true`.

The source code in [Example 8.1, “Using getter methods from class IOHandler” \[253\]](#) shows how getter methods defined in class `IOHandler` can be used to quickly match file formats to their respective input/output handler implementations and also shows how to build a list holding the descriptive strings for supported file formats.

### Example 8.1. Using getter methods from class IOHandler

```
// 'ioHandlers' is of type java.util.Collection.

IOHandler getIOHandler(String fileName)
{
    Iterator iter = ioHandlers.iterator();
    while (iter.hasNext())
    {
        IOHandler ioh = (IOHandler)iter.next();
        // Test if the given string ends with the specific file name extension of
        // any of the registered input/output handlers on the list.
        if (fileName.endsWith(ioh.getFileNameExtension()))
            return ioh;
    }
    // No matching input/output handler has been found.
    return null;
}

Collection makeIOHandlerDescription()
{
    Collection descrList = new Vector();

    Iterator iter = ioHandlers.iterator();
    while (iter.hasNext())
    {
        IOHandler ioh = (IOHandler)iter.next();
        // For every registered input/output handler on the list add its
        // descriptive format string.
        descrList.add(ioh.getFileFormatString());
    }
    // Return the description list.
    return descrList;
}
```

Both reading a graph from and writing a graph to a file is actually done with only few lines of source code. [Example 8.2, “Reading and writing a graph structure from/to file” \[253\]](#) shows two methods that can be used in conjunction with any descendant of class `IOHandler`.

### Example 8.2. Reading and writing a graph structure from/to file

```
void readGraphFromFile(Graph2D graph, IOHandler ioh, String inFile)
{
    try {
        // Reading in the graph using the given IOHandler.
        ioh.read(graph, inFile);
    }
    catch (IOException ioEx) {
        // Something went wrong. Complain.
        printErrorMessage("Cannot read graph from file '" + inFile + "'.");
    }
}

void writeGraphToFile(Graph2D graph, IOHandler ioh, String outFile)
{
    try {
        // Writing out the graph using the given IOHandler.
        ioh.write(graph, outFile);
    }
}
```

## Input/Output Handlers

---

```
    catch (IOException ioEx) {
        // Something went wrong. Complain.
        printErrorMessage("Cannot write graph to file '" + outFile + "'.");
    }
}
```

## Reading and Writing Graph Structure Data

By default, the yFiles library already offers various possibilities for writing a graph to different file formats. [Table 8.2, “Supported file formats” \[255\]](#) lists the supported graph exchange file formats.

**Table 8.2. Supported file formats**

File Name Extension	Format Name	Description
ygf	<a href="#">Y Graph Format</a>	Proprietary binary file format.
gml	<a href="#">Graph Modeling Language</a>	Text-based file format. Not to be confused with <a href="#">GraphML</a> .
xgml	<a href="#">Graph Modeling Language</a>	Text-based file format. XML-ish variant of GML.
tgf	<a href="#">Trivial Graph Format</a>	Text-based file format. Saves only a graph's structure and existing labels.

[Table 8.3, “Additional file formats” \[255\]](#) lists further graph exchange file formats that are provided by [yFiles extension packages](#).

**Table 8.3. Additional file formats**

File Name Extension	Format Name	Description
graphml	<a href="#">GraphML</a>	XML-based file format. Not to be confused with <a href="#">GML</a> .

# YGF

YGF, the "Y Graph Format," is a binary file format that supports the entire range of possible graph structure constellations. In particular, this means that beyond "flat" graphs also hierarchically organized graphs can be saved, preserving all relevant hierarchical information. This information includes, e.g., any inter-edges from the hierarchy, the structure of any inner graphs, and whether a node containing an inner graph is a group node or a folder node.

## Common Usage

A YGF file is written using class `YGFIOHandler`. This class is responsible for properly (de)serializing all graph elements to an object stream which is written to file. For (de)serialization of classes `NodeRealizer` and `EdgeRealizer`, `YGFIOHandler` resorts to their respective (de)serialization methods. The code fragment in [Example 8.3, "Instantiating a YGFIOHandler"](#) [256] shows how to instantiate a `YGFIOHandler` and using it to write a graph to file.

### Example 8.3. Instantiating a `YGFIOHandler`

```
// Instantiate a YGF I/O handler and write the graph to file.
IOHandler ioh = new YGFIOHandler();
writeGraphToFile(graph, ioh, "MyYGF.ygf");
```

## Extending YGF With Custom Data

Adding custom data from a graph to a YGF file can be accomplished by overriding appropriate methods from class `YGFIOHandler`. There is a number of protected methods that cover all sorts of graph elements, i.e., nodes and edges, but also group nodes, folder nodes, and inter-edges. For each of these elements there is a method available for both input and output, i.e., to write the respective item and to read it. [API Excerpt 8.2, "Output methods from class YGFIOHandler"](#) [256] shows all output methods for graph elements.

Note that the input/output methods for the respective realizer types are almost never suitable subjects for customization. Instead, the "Info" methods from class `YGFIOHandler` are customized. They, in turn, invoke the "Realizer" methods.

### API Excerpt 8.2. Output methods from class `YGFIOHandler`

```
// The more important methods when creating a specialization of class
// YGFIOHandler.
// "Normal" graph stuff.
// Note: All methods can throw java.io.IOException.
protected void write(Graph2D g, ObjectOutputStream out)
protected void writeEdgeInfo(Graph2D g, Edge e, ObjectOutputStream out)
protected void writeNodeInfo(Graph2D g, Node v, ObjectOutputStream out)

// Graph hierarchy stuff.
// Note: All methods can throw java.io.IOException.
protected void writeFolderNodeInfo(Graph2D g, Node v, ObjectOutputStream out)
protected void writeGroupNodeInfo(Graph2D g, Node v, ObjectOutputStream out)
protected void writeInterEdgeInfo(Graph2D g, Edge e, ObjectOutputStream out)

// The less important methods when creating a specialization of class
// YGFIOHandler. Invoked by the above methods.
protected void writeEdgeRealizer(Graph2D g, Edge e, ObjectOutputStream out)
protected void writeNodeRealizer(Graph2D g, Node v, ObjectOutputStream out)
```

## Some Rules

The important things to take care of when overriding any of the methods, can be summarized as follows:

- Providing both input and output logic, i.e., write and read implementations at once make for a complete I/O handler.
- Input and output have to be symmetric, i.e., everything written also has to be read (and vice versa). In particular, the sequence of items has to match.

Furthermore, to avoid possible backward compatibility problems that can arise from a customized file format that evolves over time, the following have proven to be good practices:

- Using version identifier marks at the beginning of a section of appended custom data. (This will help in distinguishing different versions of a customized YGF file format.)
- Providing default values for custom data that cannot be read. (For example, because the data is not present in older versions of a customized YGF file format.)

**Example 8.4, “Customized version of a YGFIOHandler”** [257] presents a customized YGFIOHandler that writes and reads custom data to/from a generated YGF file. The custom data has to be provided by an edge map that can be set using the setter method. Note that the actual data that is to be (de)serialized has to implement the `java.io.Serializable` interface.

### Example 8.4. Customized version of a YGFIOHandler

```

public class MyYGFIOHandler extends YGFIOHandler
{
    protected DataProvider em;

    public void setEdgeMap(EdgeMap em){ this.em = em; }

    // Customized version to append data to the normal edge information stuff.
    protected void writeEdgeInfo(Graph2D graph, Edge e, ObjectOutputStream out)
    throws IOException
    {
        // First, write out a version identifier mark.
        out.writeByte(YVersion.VERSION_1);

        super.writeEdgeInfo(graph, e, out);
        // After the normal stuff has been written, append the custom data.
        // The data is retrieved from the data provider (which is an edge map).
        out.writeObject((String)em.get(e));
    }

    // Customized version to read the normal edge information stuff and also the
    // appended data.
    protected void readEdgeInfo(Graph2D graph, Edge e, ObjectInputStream in)
    throws IOException
    {
        String relationType;

        switch (in.readByte())
        {
            case YVersion.VERSION_1:
                // The correct version identifier mark has been seen. Good!

                super.readEdgeInfo(graph, e, in);
                // After the normal stuff, read the appended custom data also.
                try {
                    relationType = (String)in.readObject();
                }
                catch (ClassNotFoundException cnfEx) {
                    relationType = null;
                }
                // The data is stored to the data provider (which is an edge map).
                ((EdgeMap)em).set(e, relationType);
                break;
            default:
                // Bad! The version identifier mark does not match.
                throw new IOException("Unknown file format.");
        }
    }
}

```

By means of the two methods `writeNodeRealizer(Graph2D, Node, ObjectOutputStream)` and `writeEdgeRealizer(Graph2D, Edge, ObjectOutputStream)` the respective realizer type is written to a YGF file. Properly adding data that is held by customized versions of abstract classes `NodeRealizer` or `EdgeRealizer` can then be accomplished by overriding the appropriate (de)serialization method of either realizer type. **API Excerpt 8.3, “The realizer's (de)serialization methods”** [257] lists those methods.

### API Excerpt 8.3. The realizer's (de)serialization methods

```

// These are present in both class NodeRealizer and class EdgeRealizer.
void read(ObjectInputStream in)

```

## Tutorial Demo Code

---

```
void write(ObjectOutputStream out)
```

The same [rules](#) that apply to overriding I/O methods from class YGFIOHandler also apply to overriding the serialization methods from the realizer classes.

## Tutorial Demo Code

For examples of customized realizers see the tutorial demo applications:

- [FixedPortsNodeRealizer.java](#)
- [RoundRectNodeRealizer.java](#)
- [StateNodeRealizer.java](#) together with [StateNodeRealizerDemo.java](#)
- [UMLClassNodeRealizer.java](#)

# GML

**GML**, the "Graph Modeling Language," supports the entire range of possible graph structure constellations. In particular, this means that beyond "flat" graphs also hierarchically organized graphs can be saved, preserving all relevant hierarchical information. This information includes, e.g., any inter-edges from the hierarchy, the structure of any inner graphs, and whether a node containing an inner graph is a group node or a folder node.

## What it Looks Like

Example 8.5, "GML representation" [259] shows the GML representation for the example graph from Figure 8.2, "A simple graph" [259]

The GML format in its entirety is exhaustively presented in the section called "GML File Format" [260]

**Figure 8.2. A simple graph**



**Example 8.5. GML representation**

```

Creator "yFiles"
Version 2.2
graph
[ hierachic 1
  directed 1
  node
  [ id 0
    graphics
    [ x 200.0
      y 0.0
    ]
    LabelGraphics
    [ text "January" ]
  ]
  node
  [ id 1
    graphics
    [ x 425.0
      y 75.0
    ]
    LabelGraphics
    [ text "December" ]
  ]
  edge
  [ source 1
    target 0
    graphics
    [ Line
      [ point
        [ x 425.0
          y 75.0
        ]
        point
        [ x 425.0
          y 0.0
        ]
        point
        [ x 200.0
          y 0.0
        ]
      ]
    ]
    LabelGraphics
    [ text "Happy New Year!"
      model "six_pos"
      position "head"
    ]
  ]

```

] ] ]

## Common Usage

A GML file is written using class [GMLIOHandler](#). The code fragment in [Example 8.6, “Instantiating a GMLIOHandler” \[260\]](#) shows how to instantiate a GMLIOHandler and using it to write a graph to file.

### Example 8.6. Instantiating a GMLIOHandler

```
// Instantiate a GML I/O handler and write the graph to file.
IOHandler ioh = new GMLIOHandler();
writeGraphToFile(graph, ioh, "MyGML.gml");
```

## Extending GML With Custom Data

Adding custom data from a graph to a GML file can be accomplished by providing appropriate encoder and parser logic. In package [y.io.gml](#) reside the classes for the default GML encoding and parsing behavior. These classes cover all sorts of graph elements, i.e., nodes and edges, but also hierarchy nodes and inter-edges. For any of these elements there are classes available for both input and output, i.e., to encode the respective item and to parse it.

Replacing the default GML I/O logic with customized behavior takes place at the level of these classes. Due to the nature of both the GML encoding as well as its parsing process, however, there have to be provided additional classes, so-called factories. These classes are responsible for instantiating the customized classes each time the encoding or parsing process needs the respective functionality.

The factories have to be properly registered with the GMLIOHandler instance that is used for file I/O. This is done using the methods from [API Excerpt 8.4, “Methods to register custom factories with a GMLIOHandler” \[260\]](#). Accordingly, the registered factories implement interfaces [EncoderFactory](#) and [ParserFactory](#), respectively.

### API Excerpt 8.4. Methods to register custom factories with a GMLIOHandler

```
static void setEncoderFactory(EncoderFactory _encoderFactory)
static void setParserFactory(ParserFactory _parserFactory)
```

## Tutorial Demo Code

For examples on how to provide the factory classes and the customized encoding and parsing logic see the tutorial demo applications:

- [CustomGMLDemo.java](#) together with [CustomGMLFactory.java](#), [CustomNodeObjectEncoder.java](#), and [CustomNodeParser.java](#)
- [ParserFactory.java](#) and [EncoderFactory.java](#) from the [PortsDemo](#) application

## GML File Format

The following tables describe all GML features, the attributes, and keys/tags that are handled by class [GMLIOHandler](#).

**Table 8.4. Type explanations**

Type	Description
attribute [String]	A key/value pair where the value is a quoted string.
attribute [Integer]	A key/value pair where the value is an unquoted signed integer.

Type	Description
attribute [Double]	A key/value pair where the value is an unquoted signed real number in double precision.
attribute [Integer / Boolean]	A key/value pair where the value is an unquoted signed integer. Non-negative values are interpreted as true by the parser.
attribute [String / Color]	A key/value pair where the value is a quoted string of the form #RRGGBB or #RRGGBBAA, alternatively. Each of the pairs is a hexadecimal value in the range 00 to FF that represents the red (RR), green (GG), blue (BB), and alpha (AA) component of the given color.
section (single)	A key/value pair where the value is a compound of an arbitrary number of allowed key/value pairs all enclosed in brackets, i.e., '[' and ']'.
section (multiple)	Multiple key/value pairs in direct sequence where the values are compounds of an arbitrary number of allowed key/value pairs all enclosed in brackets, i.e., '[' and ']'.

## Basic GML

The base set of GML tags. These are essential to save the structure of a graph.

**Table 8.5. GML toplevel**

Item	Type	Description	Par sed	Wr itten
.Creator	attribute [String]	The name of the creator of the file.	No	Yes
.Version	attribute [Double]	The version of the creator.	No	Yes
.graph	section (single)	The actual graph data.	Yes	Yes

**Table 8.6. GML .graph level**

Item	Type	Description	Par sed	Wr itten
.label	attribute [String]	The textual name of the graph.	No	Yes
.directed	attribute [Integer / Boolean]	Predicate whether edges are directed.	No	Yes
.node	section (multiple)	A simple node.	Yes	Yes
.edge	section (multiple)	A simple edge.	Yes	Yes
.defaultnodesize	attribute [String]	May be set to 'labelsize'. The parser will automatically assign a size for nodes where width or height is not set.	Yes	No
.defaultnodeinsetshorizontal	attribute [Integer]	Used in conjunction with defaultnodesize = 'labelsize'. Determines horizontal insets between label and node border.	Yes	No
.defaultnodeinsetsvertical	attribute [Integer]	Used in conjunction with defaultnodesize = 'labelsize'. Determines vertical insets between label and node border.	Yes	No
.defaultnodedeminwidth	attribute [Integer]	Used in conjunction with defaultnodesize = 'labelsize'. Determines minimal width of automatically assigned widths.	Yes	No

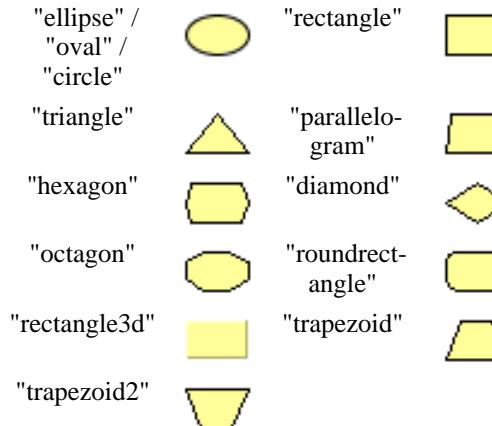
## GML File Format

Item	Type	Description	Par sed	Wr it-ten
.defaultnodeminheight	attribute [Integer]	Used in conjunction with defaultnodesize = 'labelsize'. Determines minimal height of automatically assigned heights.	Yes	No

**Table 8.7. GML .graph.node level**

Item	Type	Description	Par sed	Wr it-ten
.id	attribute [Integer]	The unique node ID.	Yes	Yes
.name	attribute [String]	The textual name of the node (label).	Yes	No
.label	attribute [String]	The textual name of the node (label).	Yes	Yes
.graphics	section (single)	The graphical representation of the node.	Yes	Yes
<a href="#">.LabelGraphics</a>	section (multiple)	Label graphics definitions.	Yes	Yes

**Table 8.8. GML .graph.node.graphics level**

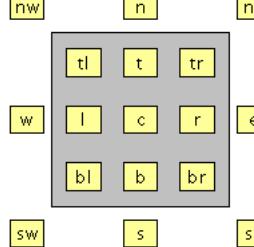
Item	Type	Description	Par sed	Wr it-ten
.x	attribute [Double]	The x-coordinate of the center of the node.	Yes	Yes
.y	attribute [Double]	The y-coordinate of the center of the node.	Yes	Yes
.w	attribute [Double]	The width of the node.	Yes	Yes
.h	attribute [Double]	The height of the node.	Yes	Yes
.type	attribute [String]	The name of the shape type. This is one of: "ellipse"/"oval"/"circle", "rectangle", "triangle", "parallelogram", "hexagon", "diamond", "octagon", "roundrectangle", "rectangle3d", "trapezoid", "trapezoid2". 	Yes	Yes
.dropShadowColor	attribute	The color of a node's drop shadow.	Yes	Yes

<b>Item</b>	<b>Type</b>	<b>Description</b>	<b>Par sed</b>	<b>Wr itten</b>
	[Color]			
.dropShadowOffsetX	attribute [Double]	The drop shadow's x-offset.	Yes	Yes
.dropShadowOffsetY	attribute [Double]	The drop shadow's y-offset.	Yes	Yes
.image	attribute [String]	The URL of the image used for image nodes.	Yes	Yes
.alphaImage	attribute [Integer / Boolean]	Whether or not alpha transparency is supported to modify an image node's hit-testing behavior.	Yes	Yes
.customconfiguration	attribute [String]	The configuration of a generic node.	Yes	Yes
.userdataclass	attribute [String]	The class name for the user data part of a generic node.	Yes	Yes
.userdata	attribute [String]	The user data part of a generic node.	Yes	Yes
.fill	attribute [Color]	The fill color of the node.	Yes	Yes
.hasFill	attribute [Integer / Boolean]	Whether or not the node has a fill color.	Yes	Yes
.fill2	attribute [Color]	A second fill color for a node.	Yes	Yes
.outline	attribute [Color]	The stroke color used for the node.	Yes	Yes
.hasOutline	attribute [Integer / Boolean]	Whether or not the node has an outline.	Yes	Yes
.width	attribute [Integer]	The line width of the outline.	Yes	No
.outlineWidth	attribute [Integer]	The line width of the outline.	Yes	Yes
.outlineStyle	attribute [String]	The name of the outline style. This is one of: "line", "dashed", "dotted".	Yes	Yes

**Table 8.9. GML .graph.node.LabelGraphics level**

<b>Item</b>	<b>Type</b>	<b>Description</b>	<b>Par sed</b>	<b>Wr itten</b>
.type	attribute [String]	The content type of the label, either "text" for textual labels or "index_label" for automatic numeric labels.	Yes	No
.text	attribute [String]	The label's text.	Yes	Yes
.outline	attribute [Color]	The outline color of the label.	Yes	Yes
.fill	attribute [Color]	The background color of the label.	Yes	Yes
.color	attribute [Color]	The text color of the label.	Yes	Yes
.model	attribute	The label model. Only written when the label	Yes	Yes

## GML File Format

Item	Type	Description	Par sed	Wr it-ten
	[String]	model is not internal, and does not contain the entire set of positions outside the node.		
.anchor	attribute [String]	The anchor type of the label that determines the position of the label relative to its node. Positions within the node are "c", "t", "tl", "tr", "l", "r", "bl", "b", or "br". Positions around the node are "n", "ne", "e", "se", "s", "sw", "w", or "nw".  	Yes	Yes
.x	attribute [Double]	The absolute x-coordinate of the label. This value will only be written if the label position does not correspond to a predefined label anchor. Also, this value will only be read if no label anchor is specified.	Yes	Yes
.y	attribute [Double]	The absolute y-coordinate of the label. This value will only be written if the label position does not correspond to a predefined label anchor. Also, this value will only be read if no label anchor is specified.	Yes	Yes
.borderDistance	attribute [Double]	The distance between label and the node's border.	Yes	Yes
.autoSizePolicy	attribute [String]	The auto sizing policy of the label. This is one of: "node_height", "node_width", or "content".	Yes	Yes
.fontSize	attribute [Integer]	The size of the label font.	Yes	Yes
.fontStyle	attribute [String]	The style of the label font, either "plain", "italic" or "bold".	Yes	Yes
.fontName	attribute [String]	The logical name of the label font, either "Dialog", "DialogInput", "Monospaced", "Serif", "SansSerif", or "Symbol".	Yes	Yes
.alignment	attribute [String]	The alignment of multi-line label, either "center", "left" or "right".	Yes	Yes
.rotationAngle	attribute [Double]	The label's rotation angle.	Yes	Yes
.icon	attribute [String]	URL for the icon's image.	Yes	No
.horizontalTextPosition	attribute [String]	A label's horizontal text positioning. This is one of: "left", "left_aligned", "center", "right", or "right_aligned".	Yes	No
.verticalTextPosition	attribute [String]	A label's vertical text positioning. This is one of: "top", "center", or "bottom".	Yes	No
.iconTextGap	attribute [Integer]	Distance between icon and label text.	Yes	No
.underlineText	attribute [Integer / Boolean]	Whether the label's text is underlined.	Yes	Yes
.visible	attribute [Integer / Boolean]	Whether the label is visible.	Yes	Yes

<b>Item</b>	<b>Type</b>	<b>Description</b>	<b>Par sed</b>	<b>Wr itten</b>
	Boolean]			

**Table 8.10. GML .graph.edge level**

<b>Item</b>	<b>Type</b>	<b>Description</b>	<b>Par sed</b>	<b>Wr itten</b>
.source	attribute [Integer]	The ID of the source node (.graph.node.id).	Yes	Yes
.target	attribute [Integer]	The ID of the target node (.graph.node.id).	Yes	Yes
.label	attribute [String]	The textual name of the edge (label).	Yes	Yes
.graphics	section (single)	The graphical representation of the edge.	Yes	Yes
.edgeAnchor	section (single)	The port definitions.	Yes	Yes
.LabelGraphics	section (multiple)	Label graphics definitions.	Yes	Yes

**Table 8.11. GML .graph.edge.graphics level**

<b>Item</b>	<b>Type</b>	<b>Description</b>	<b>Par sed</b>	<b>Wr itten</b>
.type	attribute [String]	The name of the linetype. This is one of: "arc", "spline", "bezier", or "quadCurve".	Yes	Yes
.quadCurveStraightness	attribute [Double]	The straightness of a quad curve edge.	Yes	Yes
.customconfiguration	attribute [String]	The configuration of a generic edge.	Yes	Yes
.userdataclass	attribute [String]	The class name for the user data part of a generic edge.	Yes	Yes
.userdata	attribute [String]	The user data part of a generic edge.	Yes	Yes
.smoothBends	attribute [Integer / Boolean]	Whether the edge has smoothed bends.	Yes	Yes
.width	attribute [Integer]	The stroke width used for the line.	Yes	Yes
.style	attribute [String]	The style used for the line. This is one of: "line", "dashed", "dotted".	Yes	Yes
.fill	attribute [String]	The color of the line.	Yes	Yes
.arrow	attribute [String]	"last", "first", or "both" depending on whether there are arrowheads at the edge's source end and target end.	Yes	No
.sourceArrow	attribute [String]	The type of the arrowhead at the edge's source end. This is one of: "delta", "standard", "diamond", "short", "white_delta", "white_diamond", or "none". If the arrowhead is of custom type, its name is encoded.	Yes	Yes

## GML File Format

---

Item	Type	Description	Par sed	Wr it-ten
.targetArrow	attribute [String]	The type of the arrowhead at the edge's target end. This is one of: "delta", "standard", "diamond", "short", "white_delta", "white_diamond", or "none". If the arrowhead is of custom type, its name is encoded.	Yes	Yes
.reversedPathRendering	attribute [Integer / Boolean]	Whether the edge's path is rendered starting at the target end instead of the source end.	Yes	Yes
.arcType	attribute [String]	The type of an arc edge. This is one of: "fixed-Height" or "fixedRatio".	Yes	Yes
.arcHeight	attribute [Double]	The height of an arc edge.	Yes	Yes
.arcRatio	attribute [Double]	The ration of an arc edge.	Yes	Yes
.line	section (single)	The positions of the ports and bends of the edge.	Yes	Yes

**Table 8.12. GML .graph.edge.graphics.line level**

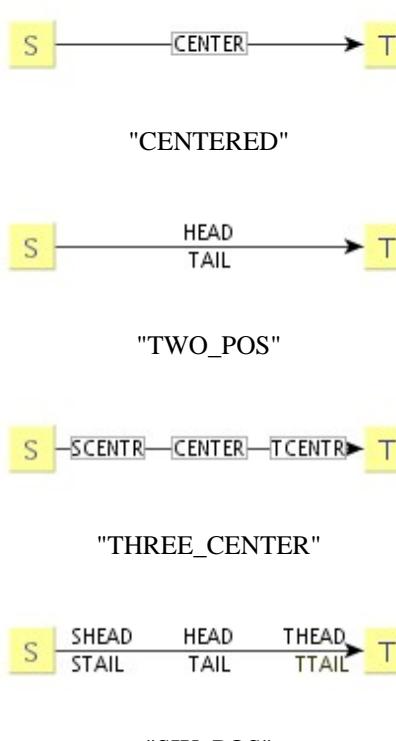
Item	Type	Description	Par sed	Wr it-ten
.point	section (multiple)	A single bend or port.	Yes	Yes

**Table 8.13. GML .graph.edge.graphics.line.point level**

Item	Type	Description	Par sed	Wr it-ten
.x	attribute [Double]	The x-coordinate of the point.	Yes	Yes
.y	attribute [Double]	The y-coordinate of the point.	Yes	Yes

**Table 8.14. GML .graph.edge.LabelGraphics level**

Item	Type	Description	Par sed	Wr it-ten
.type	attribute [String]	The content type of the label, either "text" for textual labels or "index_label" for automatic numeric labels.	Yes	No
.text	attribute [String]	The label's text.	Yes	Yes
.outline	attribute [Color]	The border color of the label.	Yes	Yes
.fill	attribute [Color]	The background color of the label.	Yes	Yes
.color	attribute [Color]	The text color of the label.	Yes	Yes
.model	attribute [String]	The model of the label that determines how to interpret the position of a label. Known models	Yes	Yes

<b>Item</b>	<b>Type</b>	<b>Description</b>	<b>Par sed</b>	<b>Wr itten</b>
		are "center_slider", "side_slider", "six_pos", "three_center", "two_pos", "free" and "centered". For each model certain label positions can be assigned (see below).		
.position	attribute [String]	<p>The symbolic position of the label, that determines the actual position of the label. Known positions are "scentr", "center" "tcentr", "shead", "head" "thead", "stail", "tail" and "ttail". The available edge label models are "CENTERED", "TWO_POS", "THREE_CENTER", and "SIX_POS".</p>  <p>"CENTERED"</p> <p>"TWO_POS"</p> <p>"THREE_CENTER"</p> <p>"SIX_POS"</p>	Yes	Yes
.x	attribute [Double]	The absolute x-coordinate of the label. This value must be used to specify the position of the label for the label model types "center_slider", "side_slider" and "free".	Yes	Yes
.y	attribute [Double]	The absolute y-coordinate of the label. This value must be used to specify the position of the label for the label model types "center_slider", "side_slider" and "free".	Yes	Yes

## GML File Format

Item	Type	Description	Par sed	Wr it-ten
		<p>"SIDE_SLIDER"</p> <p>"FREE"</p>		
.fontSize	attribute [Integer]	The size of the label font.	Yes	Yes
.fontStyle	attribute [String]	The style of the label font, either "plain", "italic" or "bold".	Yes	Yes
.fontName	attribute [String]	The logical name of the label font, either "Dialog", "DialogInput", "Monospaced", "Serif", "SansSerif", or "Symbol".	Yes	Yes
.alignment	attribute [String]	The alignment of multi-line label, either "center", "left" or "right".	Yes	Yes
.rotationAngle	attribute [Integer]	The label's rotation angle.	Yes	Yes
.icon	attribute [String]	URL for the icon's image.	Yes	No
.horizontalTextPosition	attribute [String]	A label's horizontal text positioning. This is one of: "left", "left_aligned", "center", "right", or "right_aligned".	Yes	No
.verticalTextPosition	attribute [String]	A label's vertical text positioning. This is one of: "top", "center", or "bottom".	Yes	No
.iconTextGap	attribute [Integer]	Distance between icon and label text.	Yes	No
.underlineText	attribute [Integer / Boolean]	Whether the label's text is underlined.	Yes	Yes
.visible	attribute [Integer / Boolean]	Whether the label is visible.	Yes	Yes

**Table 8.15. GML .graph.edge.edgeAnchor level**

Item	Type	Description	Par sed	Wr it-ten
.xSource	attribute [Double]	The position of the source port x-coordinate relative to half its width measured from the center of the node.	Yes	Yes
.ySource	attribute [Double]	The position of the source port y-coordinate relative to half its height measured from the center of the node.	Yes	Yes
.xTarget	attribute [Double]	The position of the target port x-coordinate relative to half its width measured from the center of the node.	Yes	Yes
.yTarget	attribute [Double]	The position of the target port y-coordinate relative to half its height measured from the center of the node.	Yes	Yes

## **GML Hierarchy Extension**

Extension to the base set of GML tags that model nested graphs, inner nodes, group nodes, and edges between different hierarchy levels (so-called inter-edges).

**Table 8.16. GML .graph level**

Item	Type	Description	Par sed	Wr itten
.hierarchic	attribute [Integer / Boolean]	Predicate whether the graph may contain sub-folders and/or group nodes.	No	Yes
.interedge	section (multiple)	An inter-edge definition.	Yes	Yes

**Table 8.17. GML .graph.node level**

Item	Type	Description	Par sed	Wr itten
.graphUrl	attribute [String]	Definition of an inner graph, referenced by an URL.	Yes	No
.graph	section (single)	Inline definition of an inner graph.	Yes	Yes
.isGroup	attribute [Integer / Boolean]	Whether or not the node is a group node that can be referenced by other nodes using the gid attribute.	Yes	Yes
.gid	attribute [Integer]	The ID of the group node that this node belongs to.	Yes	Yes

### **Note**

The inline definition of an inner graph using the .graph.node.graph combination is used for folder nodes.

**Table 8.18. GML .graph.node.graphics level**

Item	Type	Description	Par sed	Wr itten
.topBorderInset	attribute [Integer]	Top inset for a group node.	Yes	Yes
.bottomBorderInset	attribute [Integer]	Bottom inset for a group node.	Yes	Yes
.leftBorderInset	attribute [Integer]	Left inset for a group node.	Yes	Yes
.rightBorderInset	attribute [Integer]	Right inset for a group node.	Yes	Yes

**Table 8.19. GML .graph.edge level**

Item	Type	Description	Par sed	Wr itten
.id	attribute [Integer]	A (scope) unique edge ID.	Yes	Yes

**Table 8.20. GML .graph.interedge level**

Item	Type	Description	Parsed	Written
.sourcePath	attribute [String]	The relative path of node IDs (.graph.node.id), separated by ':' to the real source node.	Yes	Yes
.targetPath	attribute [String]	The relative path of node IDs (.graph.node.id), separated by ':' to the real target node.	Yes	Yes
.representative	attribute [Integer]	The ID of the representing edge in this graph (.graph.edge.id).	Yes	Yes

The encoding of an inter-edge uses two key/value pairs to define a path-like expression that uniquely identifies both its source and target node. The expression is the sequence of node IDs starting from the current graph scope and descending into the nested graph structure. The separating colon (':') models the transition to a deeper hierarchy level.

## XGML

XGML is an XML-ish variant of the GML file format where sections and attributes as listed in the various tables of section [GML File Format](#) are wrapped in <section> and <attribute> tags, respectively.

### What it Looks Like

[Example 8.7, “XGML representation” \[271\]](#) shows the beginning of a file having “GML-over-XML” encoding.

#### Example 8.7. XGML representation

```
<section name="xgml">
  <attribute key="Creator" type="String">yFiles</attribute>
  <attribute key="Version" type="String">2.3.1</attribute>
  <section name="graph">
    <attribute key="hierarchic" type="int">1</attribute>
    <attribute key="directed" type="int">1</attribute>
    <section name="node">
      <attribute key="id" type="int">0</attribute>
      <section name="graphics">
        <attribute key="x" type="double">200.0</attribute>
        <attribute key="y" type="double">0.0</attribute>
      </section>
      <section name="LabelGraphics">
        <attribute key="text" type="String">January</attribute>
      </section>
    </section>
    ...
  </section>
```

## Common Usage

An XGML file is written using class **XGMLIOHandler** which is a descendant of **GMLIOHandler** and is used similarly to its superclass. The code fragment in [Example 8.8, “Instantiating an XGMLIOHandler” \[271\]](#) shows how to instantiate an **XGMLIOHandler** and use it to write a graph to file.

### Note

**XGMLIOHandler** can be used with the Java 2 Runtime Environment beginning with version 1.4.

#### Example 8.8. Instantiating an XGMLIOHandler

```
// Instantiate an XGML I/O handler and write the graph to file.
IOHandler ioh = new XGMLIOHandler();
writeGraphToFile(graph, ioh, "MyXGML.xgml");
```

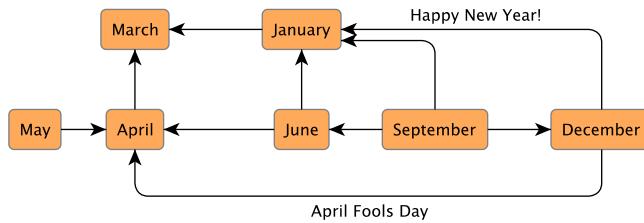
## TGF

TGF, the "Trivial Graph Format," supports the pure graph structure together with (at most) one label per graph element. Anything related to position, dimension, or visual representation is ignored. Also, TGF does not provide support for nested graph structures.

### What it Looks Like

[Example 8.9, “TGF representation” \[272\]](#) shows the TGF representation for the example graph from [Figure 8.3, “A simple graph” \[272\]](#)

**Figure 8.3. A simple graph**



**Example 8.9. TGF representation**

```

1 January
2 March
3 April
4 May
5 December
6 June
7 September
#
1 2
3 2
4 3
5 1 Happy New Year!
5 3 April Fools Day
6 3
6 1
7 5
7 6
7 1
  
```

### Common Usage

A TGF file is written using class `TGFIOHandler`. The code fragment in [Example 8.10, “Instantiating a TGFIOHandler” \[272\]](#) shows how to instantiate a `TGFIOHandler` and using it to write a graph to file.

**Example 8.10. Instantiating a TGFIOHandler**

```

// Instantiate a TGF I/O handler and write the graph to file.
IOHandler ioh = new TGFIOHandler();
writeGraphToFile(graph, ioh, "MyTGF.tgf");
  
```

By means of the setter methods listed in [API Excerpt 8.5, “Setter methods from class TGFIOHandler” \[272\]](#) both writing and reading of node labels and edge labels, respectively can be controlled.

**API Excerpt 8.5. Setter methods from class TGFIOHandler**

```
void setIgnoreEdgeLabels(boolean ignore)
void setIgnoreNodeLabels(boolean ignore)
```

### Note

Due to the lack of support for both coordinates and dimensions reading a graph from a TGF file can result in a representation that differs from the exported graph.

# Exporting a Graph's Visual Representation

By default, the yFiles library already offers various possibilities for writing a graph (or more precisely, a graph's visual representation) to different graphic file formats. [Table 8.21, “Supported image file formats” \[274\]](#) lists the supported graph export file formats.

Both exporting to stand-alone image files, as well as generating HTML-suited versions, i.e., tiled images or image maps is possible. Furthermore, tiled images can optionally be wrapped in HTML tables.

**Table 8.21. Supported image file formats**

File Name Extension	Format Name	Description
gif	Graphics Inter-change Format	Bitmap graphics file format.
jpg	Joint Photographic Experts Group	Bitmap graphics file format.
png	Portable Network Graphics	Bitmap graphics file format. Accessible via <a href="#">ImageIoOutputHandler</a> which uses the Java Image I/O API (available with J2SE 1.4 and higher).
html	HyperText Markup Language	Serves as a container for both tiled images and image maps.

[Table 8.22, “Supported vector graphics file formats” \[274\]](#) lists further graph export file formats that are provided by [yFiles extension packages](#).

**Table 8.22. Supported vector graphics file formats**

File Name Extension	Format Name	Description
svg, svgz	Scalable Vector Graphics	XML-based file format.
pdf	Portable Document Format	Desktop publishing file format for representing documents in a device independent and display resolution independent way.
swf	Shockwave Flash	Vector graphics file format for publishing multimedia contents on the Web.
emf	Enhanced Windows Metafile	Vector graphics file format on Microsoft®Windows® systems.
eps	Encapsulated PostScript	PostScript® document with additional restrictions that describes an image or drawing.

The base class for all image export is abstract class [ImageOutputHandler](#). It has methods from its direct superclass [IOHandler](#) that are related to file input disabled (they return either `false` or throw "Unsupported Operation" exception).

File output, i.e., the actual process of exporting a [Graph2D](#) to an image is implemented as two steps:

1. The graph is rendered to an image by its current view, i.e., an instance of type [Graph2DView](#).
2. The resulting image is encoded and written to an output stream either by directly using one of the classes for image export, such as [GIFIOHandler](#) or [JPGIOHandler](#), or by using the services of class [ImageIoOutputHandler](#).

[API Excerpt 8.6, “Abstract methods from class ImageOutputHandler” \[275\]](#) lists the methods representing the two steps. They are defined abstract, so descendants of class [ImageOutputHandler](#)

have to implement them appropriately.

### API Excerpt 8.6. Abstract methods from class ImageOutputHandler

```
protected abstract BufferedImage createImage(int width, int height)
protected abstract void encode(BufferedImage image, OutputStream out)
```

For situations where no proper Graph2DView is associated with a Graph2D object, class ImageOutputHandler offers the possibility to create a default Graph2DView object. Subsequently, this view can be registered with the graph as the current view, and be used to render the image. Note that this process is also triggered whenever an ImageOutputHandler encounters a graph that lacks a current view.

As a further option, ImageOutputHandler also provides for anti-aliasing with image export.

## Preparing the Graph

Before a graph's visual representation can be exported to an image file, it has to be properly prepared. Preparation of a [Graph2D](#) means setting up the current view, more precisely, the instance of type [Graph2DView](#) that contains the graph.

This Graph2DView object determines the width and height of the image, the graph's clipping that is to be exported, and also its zoom level. Further settings influence the level of detail that is to be shown on the resulting image, i.e., if, among other things, there should be labels or not. Moreover, anything that is displayed in this view, for example a background image, can be exported to the file, too.

Setting up a view can be done in several ways, either

- directly as shown in [Example 8.11, “Directly setting up a view for image export” \[275\]](#) (similar code can be seen in tutorial demo application [GraphFormatConverter.java](#)), or
- using the services of class [ViewPortConfigurator](#).

### Example 8.11. Directly setting up a view for image export

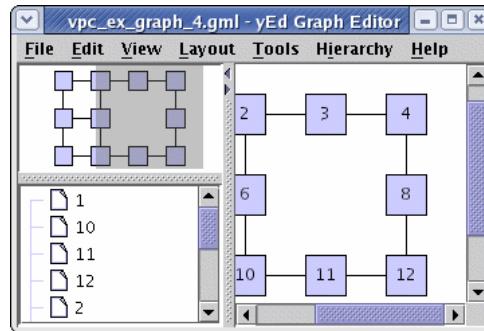
```
void configureView(Graph2DView view)
{
    Graph2D graph = view.getGraph2D();
    Rectangle box = graph.getBoundingBox();
    Dimension dim = box.getSize();

    // Set the view's width and height, in turn this also sets the image's size.
    view.setSize(dim);
    // The clipping should show the entire graph. (The graph's bounding is a
    // little enlarged.)
    view.zoomToArea(box.getX() - 10, box.getY() - 10,
                    box.getWidth() + 20, box.getHeight() + 20);

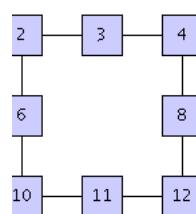
    // Set the detail threshold so that it is never switched to less detail mode.
    view.setPaintDetailThreshold(0.0);
}
```

Class ViewPortConfigurator is a convenience that can be used to set up a view. It supports common scenarios for image export, for example, exporting the entire graph or only that part of a graph that can be seen in its current view. In addition, it offers functionality for adjusting the size of an image, for example, it allows to automatically adjust the height of an image given the desired width and the graph's scaling factor. The results when using ViewPortConfigurator for the two main setup scenarios for image export are depicted in [Figure 8.4, “Clipping scenarios with class ViewPortConfigurator” \[275\]](#)

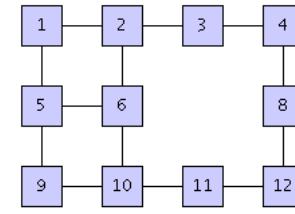
**Figure 8.4. Clipping scenarios with class ViewPortConfigurator**



Application window showing the original view together with its graph.



Resulting image when only the visible part from the Graph2DView containing the graph is being exported using [CLIP\\_VIEW](#).



Resulting image when the complete graph is being exported using [CLIP\\_GRAPH](#).

**Example 8.12.** “Using class ViewPortConfigurator to set up a view for image export” [276] shows how to configure a view using class ViewPortConfigurator. Except for image size and graph clipping all configurator settings are left to their respective default values, i.e., a zoom level of 100%, and an additional margin of 15 pixels around the bounding box of the graph is used to set up the view.

### Example 8.12. Using class ViewPortConfigurator to set up a view for image export

```
void configureExportView(Graph2DView exportView)
{
    ViewPortConfigurator vpc = new ViewPortConfigurator();

    // Register the graph to be exported with the configurator instance.
    // Depending on the other settings (see below) the graph will be used to
    // determine the image size, for example.
    vpc.setGraph2D(exportView.getGraph2D());

    // The complete graph should be exported, hence set the clipping type
    // accordingly.
    vpc.setClipType(ViewPortConfigurator.CLIP_GRAPH);
    // The graph's bounding box should determine the size of the image.
    vpc.setSizeType(ViewPortConfigurator.SIZE_USE_ORIGINAL);

    // Configure the export view using mainly default values, i.e., zoom level
    // 100%, and 15 pixel margin around the graph's bounding box.
    vpc.configure(exportView);
}
```

A convenient technique to prepare a Graph2D instance for image export whenever its current view cannot be modified, because

- the user experience would change unacceptably (e.g., the desired clipping and zoom level for the image differ from the settings of the view), or
- there simply is no current view (e.g., when processing graph files),

is to temporarily replace the currently active Graph2DView instance with a newly created one. This dedicated “export” view is then configured and used to write out the image, and afterwards the original Graph2DView instance is restored. The methods in [Example 8.13](#), “Using a dedic-

ated "export" view" [277] demonstrate how to do the replacing and restoring.

### Example 8.13. Using a dedicated "export" view

```
Graph2DView replaceCurrentWithExportView(Graph2D graph, ImageOutputHandler ioh)
{
    // Save the currently active view.
    Graph2DView originalView = (Graph2DView)graph.getCurrentView();

    // Create a new Graph2DView instance with the graph. This will be the
    // dedicated view for image export.
    Graph2DView exportView = ioh.createDefaultGraph2DView(graph);
    // Use the Graph2DRenderer instance of the currently active view. (Optional.)
    exportView.setGraph2DRenderer(originalView.getGraph2DRenderer());

    // Replace the currently active view containing the graph with the "export"
    // view.
    graph.setCurrentView(exportView);

    return originalView;
}

void restoreOriginalView(Graph2D graph, Graph2DView originalView)
{
    // Remove the "export" view from the graph.
    graph.removeView(graph.getCurrentView());
    // Reset the current view to the originally active view.
    graph.setCurrentView(originalView);
}
```

#### Note

Using method `replaceCurrentWithExportView(Graph2D, ImageOutputHandler)` from [Example 8.13, “Using a dedicated “export” view” \[277\]](#) with an `ImageOutputHandler` of type `GIFIOHandler` will result in a transparent background for the generated images. This is the default behavior of method `createDefaultGraph2DView(Graph2D)` from class `GIFIOHandler`.

#### Note

Setting a `Graph2DRenderer` on the export view is only necessary when the `Graph2DRenderer` used with the original view is a specialized class that provides custom behavior.

## Using GIF and JPG

In the second step of exporting a `Graph2D` to an image file, the actual encoding is done. Classes `GIFIOHandler` and `JPGIOHandler` are responsible for encoding the already rendered image to GIF and JPG file format, respectively.

As a single possibility of control, class `JPGIOHandler` offers a setter method to define the encoding quality. The other thing to observe with class `GIFIOHandler` is its `createDefaultGraph2DView(Graph2D)` method. By default, this method sets a transparent background to be used for image generation; [Example 8.14, “Resetting the default background renderer for a Graph2DView instance” \[277\]](#) shows how to circumvent this behavior easily. (One occasion for resetting the background renderer would be right after the call to the aforementioned method.)

### Example 8.14. Resetting the default background renderer for a Graph2DView instance

```
// 'exportView' is of type y.view.Graph2DView.

exportView.setBackgroundRenderer(new DefaultBackgroundRenderer(exportView));
```

The code fragment in [Example 8.15, “Creating an ImageOutputHandler” \[278\]](#) shows how to instantiate the actual descendants of `ImageOutputHandler` for exporting a `Graph2D` to an image. And [Example 8.16, “Exporting a Graph2D object as an image” \[278\]](#) summarizes all necessary steps for image export in their natural order.

### Example 8.15. Creating an ImageOutputHandler

```
// 'ioh' is of type y.io.ImageOutputHandler.

// Create the proper ImageOutputHandler for image generation.
if (exportAsGIF)
    // Either GIF...
    ioh = new GIFIOHandler();
else
{
    // ... or JPG file format.
    ioh = new JPGIOHandler();
    // Set the image quality to 90%. This yields a good compromise between small
    // file size and high quality.
    ((JPGIOHandler)ioh).setQuality(0.9f);
}

// Write the generated image to a file.
String ext = ioh.getFileNameExtension();
// Mind the dot! It is not part of the actual file name extension...
exportGraphToImageFileFormat(graph, ioh, "My" + ext.toUpperCase() + "." + ext);
```

### Example 8.16. Exporting a Graph2D object as an image

```
void exportGraphToImageFileFormat(Graph2D graph, ImageOutputHandler ioh,
                                  String outFile)
{
    Graph2DView originalView = replaceCurrentWithExportView(graph, ioh);
    configureExportView((Graph2DView)graph.getCurrentView());

    writeGraphToFile(graph, ioh, outFile);
    restoreOriginalView(graph, originalView);
}
```

## Using the Java Image I/O API

Class [ImageIoOutputHandler](#) can be used in the second step of exporting a Graph2D to an image file. It serves as an adapter to so-called image writers that are registered with the Java Image I/O API. Out of the box, J2SE 1.4 supports PNG (Portable Network Graphics) as an additional image file format. J2SE 5.0 adds BMP file format, a bitmap graphics file format popular on the Microsoft Windows platform.

The central class of Java Image I/O API is `javax.imageio.ImageIO`. It offers several static methods that allow to query and to retrieve available image writers. [Example 8.17, “Creating a PNG image output handler” \[278\]](#) demonstrates the use of `ImageIO` to create an `ImageOutputHandler` object that provides for PNG image exporting.

### Example 8.17. Creating a PNG image output handler

```
ImageOutputHandler createPNGOutputHandler()
{
    // Use the services of Java Image I/O API to see whether there is an image
    // writer registered that is capable of writing the PNG graphics file format.
    Iterator it = ImageIO.getImageWritersBySuffix("png");
    ImageWriter iw = (ImageWriter)(it.hasNext() ? it.next() : null);

    // Return an image output handler that serves as an adapter to this image
    // writer.
    return ((iw == null) ? null : new ImageIoOutputHandler(iw));
}
```

## Image Tiling

Producing images from graphs is a memory intensive task, especially when the given graph extends a lot in both horizontal and vertical direction. Image tiling, i.e., producing a number of small images instead of a single big one, helps in decreasing the memory footprint substantially. Also, compared to a single big image many small ones are much better suited for deployment on web pages, because web browsers can cache them more efficiently.

Class [TiledImageOutputHandler](#) can be used as a thin wrapper around any of the classes for image export, e.g., [GIFIOHandler](#), or [JPGIOHandler](#). The respective class is then used as a delegate that is invoked from within the tiling process. To control the tiling, class [TiledImageOutputHandler](#) provides methods to set the number of rows and columns, or, alternatively, to set the maximal size of a tile. In the latter case, the number of rows and columns is determined automatically. As a further option an additional HTML table that holds the tiles can be generated.

[Example 8.18, “Using class TiledImageOutputHandler” \[279\]](#) demonstrates how image tiling can be done with only a few lines of additional code.

### Example 8.18. Using class TiledImageOutputHandler

```
IOHandler getTilingWrapper(ImageOutputHandler ioh, int row, int column,
                           boolean htmlTable)
{
    // Check for valid parameter values.
    if (row < 1 || column < 1)
        return ioh;

    TiledImageOutputHandler tioh = new TiledImageOutputHandler(ioh);
    // Set the tiling accordingly.
    tioh.setRowCount(row);
    tioh.setColumnCount(column);

    // Set HTML table generation accordingly.
    tioh.setHTMLTableGenerationActive(htmlTable);

    return tioh;
}
```

## Image Maps

Class [ImageMapOutputHandler](#) can be used to generate corresponding HTML image maps for images exported from graphs. For every rectangular shaped graph element, i.e., nodes, node labels, and edge labels, it can generate an HTML `<area>` element with the dimensions of the respective element used as active area. It can also generate an HTML `<area>` element for arbitrarily shaped edge paths. There, the active area is determined by the path together with a distance around it. All such generated `<area>` elements will be encapsulated in one HTML `<map>` element.

To have an HTML `<area>` element created, the respective graph element has to be associated with a link information record of type [LinkInfo](#). Most importantly, this record contains the URL the link should point to. Association of graph element and link information record is done with the help of a [LinkMap](#) object which is given to class [ImageMapOutputHandler](#).

[Example 8.19, “Creating a simple image map” \[279\]](#) shows a code fragment that associates all nodes of a graph to link to one URL.

### Example 8.19. Creating a simple image map

```
// Create the link map that will hold the link information records for graph
// elements.
LinkMap lm = new LinkMap();
lm.setName("yWorks");

// Create one link information record for every node of the graph.
LinkInfo li = new LinkInfo();

// The actual URL that should be used for the link.
li.setAttribute(LinkInfo.HTML_REFERENCE, "http://www.yworks.com/");
// An alternate text for the link.
li.setAttribute(LinkInfo.HTML_ALT, "Visit yWorks");
// A tool tip text (for some UAs).
li.setAttribute(LinkInfo.HTML_TITLE, "Visit yWorks");

// Associate the link information record with every node from the graph.
for (NodeCursor nc = graph.nodes(); nc.ok(); nc.next())
    lm.put(nc.node(), li);

ImageMapOutputHandler imoh = new ImageMapOutputHandler();
// Hand the link map with all its link information records over to the image
// map generator.
imoh.setReferences(lm);
```

## Image Maps

---

```
// Get the generated HTML <map> element.  
String mapElement = imoh.createHTMLString();
```

For further examples see also [ImageMapDemo.java](#) from the tutorial demo applications.

## yFiles Extension Packages

yFiles extension packages offer further possibilities to export or exchange (i.e. read and write) graph structure data.

### GraphML

[GraphML](#) is the XML-based result of the graph drawing community's joint effort to define a common file format for exchanging graph structure data.

The GraphML extension package enables yFiles to read and write graphs in GraphML and Zip-compressed GraphML file format. It covers all essential structural and graphical attributes defined in yFiles. Please see [Chapter 10, GraphML Extension Package \[295\]](#) for an in-depth discussion.

See the [GraphML primer](#) for an exhaustive description of the GraphML graph exchange file format.

### ySVG

The ySVG extension package adds support for the [Scalable Vector Graphics \(SVG\)](#) file format to the yFiles graph visualization library. Features include:

- An extendable SVG and SVGZ writer for yFiles [Graph2D](#) objects that makes it possible to add hyperlinks, roll-over effects, and other useful stuff to the generated SVG diagrams.
- The possibility to define the appearance of nodes in SVG format.

Please see [Chapter 11, ySVG Extension Package \[311\]](#) for more information on this extension package.

### yExport

The yExport extension package adds support for popular vector graphics file formats to the yFiles graph visualization library. The following output formats are available to save the visual representations for nodes and edges as well as any other graphical decorations visible:

- PDF (Portable Document Format) - desktop publishing file format for representing documents in a device independent and display resolution independent way.
- SWF (Shockwave Flash) - vector graphics file format for publishing multimedia contents on the Web.
- EMF (Windows Enhanced Metafile) - vector graphics file format on Microsoft Windows systems.
- EPS (Encapsulated PostScript) - PostScript document with additional restrictions that describes an image or drawing.

The PDF output capabilities of yExport enable to easily integrate diagrams in PDF documents, the de-facto standard for printable documents. EMF is the format of choice to import, respectively paste via the clipboard, high-quality diagrams into Windows text processing and graphics applications, like, e.g., Microsoft Word or Microsoft PowerPoint®.

Please see [Chapter 12, yExport Extension Package \[315\]](#) for more information on this extension package.



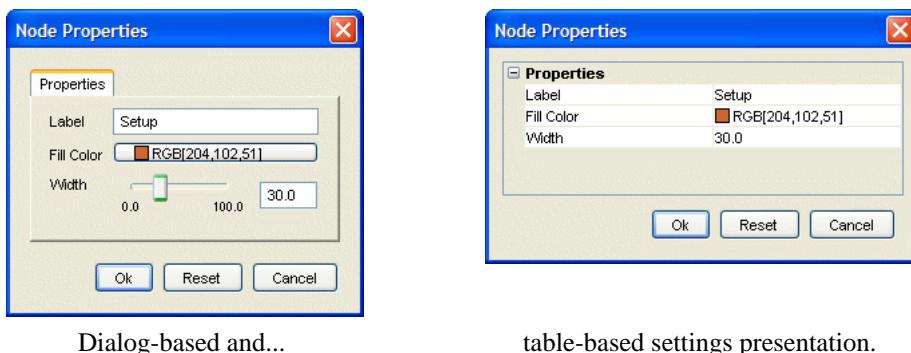
# Chapter 9. Settings Dialogs and Components

This chapter discusses GUI utility functionality that provides means to quickly create property editors for items of diverse types.

## Overview

The classes from package `y.option` constitute a framework to conveniently handle property management and manipulation. It offers support for internationalization and localization of dialogs (i18n/110n for short), as well as (de)serialization of settings. The framework separates responsibilities applying a Model-Delegate pattern, which facilitates, for example, different styles of settings presentations and manipulation. [Figure 9.1, “Dialog-based and table-based settings presentation” \[283\]](#) shows the alternative presentation styles.

**Figure 9.1. Dialog-based and table-based settings presentation**



The separation also permits the use of multiple dialogs (in other words, multiple views) to present and modify settings that are managed by a common model. Synchronization of model and view(s) is covered by different policies, following "push" and "pull" semantics.

## Model-Delegate Pattern

The classes from package `y.option` reflect a modification of the Model-View-Controller (MVC) paradigm, dubbed Model-Delegate pattern. Two of the three responsibilities from MVC, namely the view and controller part, are brought together in a so-called User Interface (UI) Delegate. Hence classes with the following roles can be distinguished:

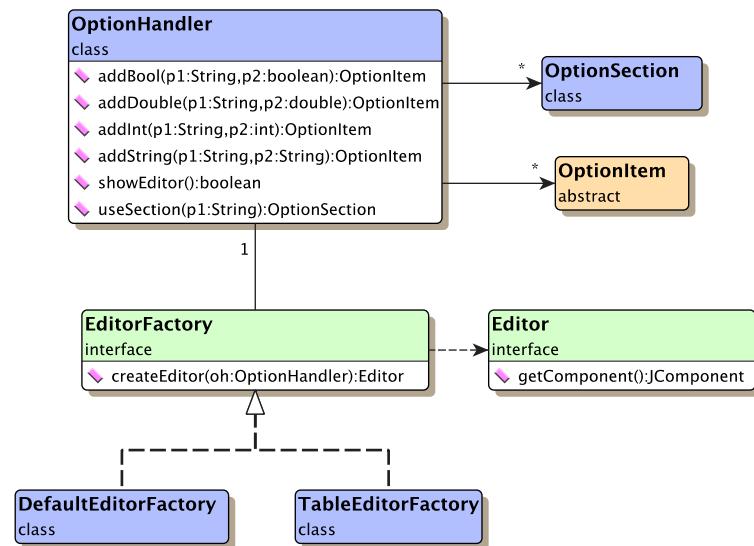
- Model classes: hold data specific to the application
- View/Controller classes: present information to and interact with the user

In terms of MVC then, class `OptionHandler` together with classes `OptionItem` and `OptionSection` make up the model which holds the data. The view/controller part is in the implementations of interface `Editor`.

[Figure 9.2, “Settings framework core classes” \[283\]](#) shows the essential classes of the settings framework.

**Figure 9.2. Settings framework core classes**

## Model-Delegate Pattern



# Basic Functionality

## Creating Model and View

Class `OptionHandler` is the settings framework's central class. It presents a kind of container to add so-called "option items" to, and notifies registered listeners of any change to the value or state of such an option item.

An option item is a descendant of abstract class `OptionItem` that represents a specific type of value, e.g., a boolean value, an integral value, but also a color object or an enumeration type. Example 9.1, "Option handler setup" [285] shows the creation of a simple option handler where several option items are added to an `OptionHandler` instance.

### Example 9.1. Option handler setup

```
public OptionHandler createOptionHandlerForNode(NodeRealizer nr)
{
    // Create a new OptionHandler object.
    OptionHandler oh = new OptionHandler("Node Properties");
    // The first section holds the properties.
    oh.useSection("Properties");
    // Add a number of different option items to this option handler.
    oh.addString("Label", nr.getLabelText());
    oh.addColor("Fill Color", nr.getFillColor());
    oh.addDouble("Width", nr.getWidth(), 0.0, 100.0);

    // Return the created option handler.
    return oh;
}
```

Option items are the fundamental blocks that actually hold data, and in conjunction with classes `OptionHandler` and `OptionSection` they make up the model. Presentation of the model's data and also the handling of any user interaction with it is the responsibility of so-called "editors," i.e., implementations of interface `Editor`. Following the MVC paradigm, multiple editors (in other words, multiple views) can be associated with an option item.

### Note

An option handler itself is also represented by an editor.

The settings framework knows several ways to create an actual user interface for the editor of an option handler. With the single line presented in Example 9.2, "Showing a default modal dialog for an option handler" [285a] default modal dialog for an option handler can be readily created and displayed. The resulting dialog features OK, Reset, and Cancel buttons.

### Example 9.2. Showing a default modal dialog for an option handler

```
// Create a default modal dialog for the option handler.
// If "OK" was pressed, 'result' is true.
boolean result = oh.showEditor();
```

By means of an `EditorFactory` an editor of type `Editor` can be retrieved that handles an option handler's entire presentation and also the user interaction. Interface `Editor` defines the general contract for an option handler's editor and provides the method listed in API Excerpt 9.1, "Interface Editor's Swing component getter method" [285] to retrieve a proper `javax.swing.JComponent`.

### API Excerpt 9.1. Interface Editor's Swing component getter method

```
// Getting a proper JComponent for displaying the editor.
JComponent getComponent()
```

Note that the `JComponent` features all embedded item editors, however has no buttons yet.

Table 9.1, “Predefined editor factory implementations” [286] lists the predefined editor factory implementations from package `y.option`. These support different presentation styles.

**Table 9.1. Predefined editor factory implementations**

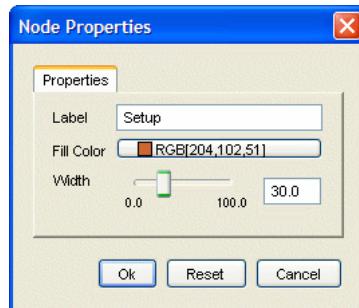
Classname	Description
DefaultEditorFactory	Creates a dialog-style presentation for an option handler.
TableEditorFactory	Creates a table-style presentation for an option handler.

### Note

Both `DefaultEditorFactory` and `TableEditorFactory` actually return an editor that has type `DefaultCompoundEditor` instead of only `Editor`.

Figure 9.3, “Dialog presentation” [286] shows the results of using the services of class `DefaultEditorFactory` for option handler presentation.

**Figure 9.3. Dialog presentation**



The code snippet in Example 9.3, “Creating a dialog-based presentation of an option handler” [?] demonstrates the use of `DefaultEditorFactory` to create a dialog-based presentation for an option handler.

**Example 9.3. Creating a dialog-based presentation of an option handler**

```
// 'nr' is of type NodeRealizer.

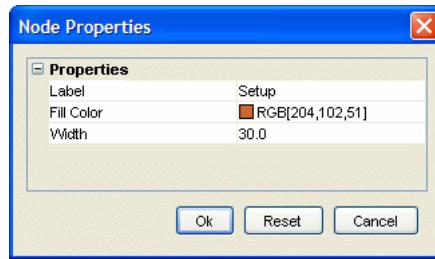
// Create an option handler for a node's properties.
OptionHandler oh = createOptionHandlerForNode(nr);

// Instantiate a new DefaultEditorFactory object and let it create a
// dialog-based (as opposed to table-based) presentation, the so-called
// "editor," for the given OptionHandler instance.
Editor ed = new DefaultEditorFactory().createEditor(oh);

// Get the editor's JComponent that holds the actual presentation of the
// properties.
JComponent ohEditorComponent = ed.getComponent();
```

Figure 9.4, “Table presentation” [286] shows the results of using the services of class `TableEditorFactory` for option handler presentation.

**Figure 9.4. Table presentation**



The code snippet in [Example 9.4, “Creating a table-based presentation of an option handler” \[287\]](#) demonstrates the use of TableEditorFactory to create a table-based presentation for an option handler.

### Example 9.4. Creating a table-based presentation of an option handler

```
// 'nr' is of type NodeRealizer.

// Create an option handler for a node's properties.
OptionHandler oh = createOptionHandlerForNode(nr);

// Instantiate a new TableEditorFactory object and let it create a table-based
// (as opposed to dialog-based) presentation, the so-called "editor," for the
// given OptionHandler instance.
Editor ed = new TableEditorFactory().createEditor(oh);

// Get the editor's JComponent that holds the actual presentation of the
// properties.
JComponent ohEditorComponent = ed.getComponent();
```

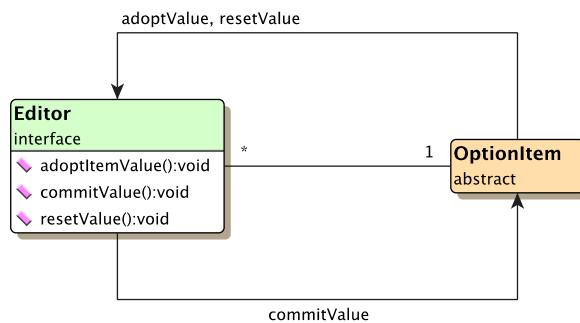
## Synchronizing Editors and Option Items

To keep the model (option item) and its views (editors) in sync, it is necessary that any modification of values or state on either of the two sides can be propagated to the other side. Interface [Editor](#) supports different policies for explicit synchronization:

- propagation of any modifications from view(s) to model ("push" semantics)
- propagation of any modifications from model to view(s) ("pull" semantics)

Related to synchronization issues is resetting values to their original value to undo any modifications on the view's side. [Figure 9.5, “Synchronization policies between editors and option items” \[287\]](#) depicts the synchronization policies and directions supported by the model-view architecture of editors and option items.

**Figure 9.5. Synchronization policies between editors and option items**



In addition to explicit synchronization, a table-style editor created by TableEditorFactory by default also supports bi-directional implicit synchronization. Whenever a value or a state in either

model or view changes, the change is immediately propagated to the other side.

Implicit synchronization can be controlled using the methods listed in [API Excerpt 9.2, “Methods to control implicit synchronization” \[288\]](#). They are supported by class TableEditorFactory as a convenience, however, ItemEditor also provides these methods.

### API Excerpt 9.2. Methods to control implicit synchronization

```
// Getter/setter methods from class y.option.TableEditorFactory.  
boolean isAutoAdopt()  
boolean isAutoCommit()  
  
void setAutoAdopt(boolean autoAdopt)  
void setAutoCommit(boolean autoCommit)
```

## Defining Dependencies Between Option Items

Class [ConstraintManager](#) offers convenient means to define so-called constraints for option items. Constraints denote dependencies between option items, more specifically between the value and the state of option items. They are used to impose modifications to a "target" option item's state in response to a change in value of a "source" option item.

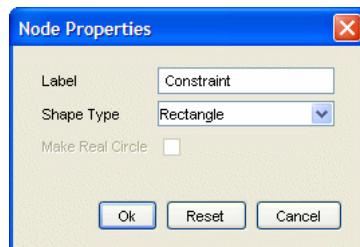
ConstraintManager has a number of methods that can be directly used to define a dependency to automatically enable/disable a single option item or a group of option items. [API Excerpt 9.3, “Methods to directly define constraints” \[288\]](#) lists some of these methods.

### API Excerpt 9.3. Methods to directly define constraints

```
// Source/target definition by reference.  
void setEnabledOnValueEquals(OptionItem source, Object value, OptionGroup target)  
void setEnabledOnValueEquals(OptionItem source, Object value, OptionItem target)  
  
// Name-based source/target definition.  
void setEnabledOnValueEquals(String source, Object value, String target)
```

Figure 9.6, “Option handler with constrained option item” [288] depicts the effects of constrained option items where a boolean option item is automatically disabled due to the enumeration option item's value.

**Figure 9.6. Option handler with constrained option item**



The constraint to automatically enable/disable the boolean option item from Figure 9.6, “Option handler with constrained option item” [288] is established as seen in [Example 9.5, “Defining a simple constraint for option items” \[288\]](#).

### Example 9.5. Defining a simple constraint for option items

```
public OptionHandler createConstrainedOptionHandlerForNode(NodeRealizer nr)  
{  
    // Create a new OptionHandler object.  
    OptionHandler oh = new OptionHandler("Node Properties");  
    // Add a number of different option items to this handler.  
    oh.addString("Label", nr.getLabelText());  
    // Define a set of possible values for the enum option item.  
    String shapeType[] = {"Rectangle", "Rounded Rectangle", "Ellipse"};
```

```

int tmp = -1;
if (nr instanceof ShapeNodeRealizer)
    tmp = ((ShapeNodeRealizer)nr).getShapeType();
oh.addEnum("Shape Type", shapeType, tmp);

// To be used as the target for a constraint.
oh.addBool("Make Real Circle", false);

// Create a new ConstraintManager object that defines constraints for some
// option items.
ConstraintManager cm = new ConstraintManager(oh);
cm.setEnabledOnValueEquals("Shape Type", "Ellipse", "Make Real Circle");

// Return the created option handler.
return oh;
}

```

Dependencies between option items can also be defined using multiple "source" option items. The methods listed in [API Excerpt 9.4, “Methods to define condition-based constraints” \[289\]](#) take as parameter an object of type `ConstraintManager.Condition` that can be used to formulate a condition that combines the values of more than one option item.

### API Excerpt 9.4. Methods to define condition-based constraints

```

void setEnabledOnCondition(Condition c, OptionItem target)
void setEnabledOnCondition(Condition c, OptionGroup target)

```

[API Excerpt 9.5, “Methods to define constraint conditions” \[289\]](#) presents some of the methods provided by class `ConstraintManager` to create `Condition` objects. Using methods from static inner class `Condition` which offer logical operations like AND, INVERSE, OR, and XOR, any number of conditions can furthermore be combined to create new `Condition` objects of increased complexity.

### API Excerpt 9.5. Methods to define constraint conditions

```

// Source item definition by reference.
// (Excerpt.)
Condition createConditionValueEquals(final OptionItem src, final Object value)
Condition createConditionValueIs(final OptionItem src, final Object[] values)
Condition createConditionValueIs(final OptionItem src, final Collection values)

// Name-based source item definition.
// (Excerpt.)
Condition createConditionValueEquals(String source, final Object value)
Condition createConditionValueIs(final String source, final Object[] values)
Condition createConditionValueIs(final String source, final Collection values)

```

Note that class `ConstraintManager` provides its services using the general events and listener mechanism, i.e., as a listener for property change events. Using the same general mechanism, it is easily possible to have arbitrarily complex conditional modifications applied to any option item's value or state. [API Excerpt 9.6, “Option handler methods for listener \(de\)registration” \[289\]](#) lists class `OptionHandler`'s methods to register custom property change listeners for single properties as well as all properties managed by an option handler.

### Important

There is no check or validation scheme provided to ensure that the entirety of dependencies defined for an option handler's items is free from any circular dependencies.

### API Excerpt 9.6. Option handler methods for listener (de)registration

```

// Listener registration methods.
void addPropertyChangeListener(PropertyChangeListener pcl)
void addPropertyChangeListener(String property, PropertyChangeListener pcl)

// Listener deregistration methods.
void removePropertyChangeListener(PropertyChangeListener pcl)
void removePropertyChangeListener(String property, PropertyChangeListener pcl)

```

## Undefined Values in Option Items

Normally, an option item holds a value of specific type. There may arise situations, however, where an option item cannot display anything useful, and instead should show a blank entry to indicate a so-called "undefined value" state. Such situations may be:

- The value to be presented by the option item is not compatible with its type. This may be due to the type being bounded, or consisting of only a set of distinct values (as in an enumeration option item).
- An application has multiple values for one option item to be displayed.

The latter occurs whenever the content of an option handler is derived from multiple entities, which have different values for at least one property. A simple example would be an option handler that is used to show the size property of two nodes, and the size of the first differs from that of the second node. The option handler's option item should consequently indicate the "undefined value" state for the size property.

[API Excerpt 9.7, “Methods to handle undefined values” \[290\]](#) show the methods that can be used to explicitly set this special state for an option item.

### API Excerpt 9.7. Methods to handle undefined values

```
// Getter and setter method for undefined values in option items.
boolean isValueUndefined()
void setValueUndefined(boolean valueUndefined)
```

#### Note

When using method `setValueUndefined` on an option item, it should be invoked last.

A special case regarding undefined values is presented in [Example 9.6, “Simple OptionHandler dialog” \[290\]](#). Class `EnumOptionItem` automatically assumes "undefined value" state whenever a value has been set that is not contained in the set of allowed ones. To enforce this behavior, method `setValueUndefined` is overridden and simply ignores arguments that contradict the current state.

### Example 9.6. Simple OptionHandler dialog

```
public OptionHandler createEnumerationOptionHandlerForNode(NodeRealizer nr)
{
    // Create a new OptionHandler object.
    OptionHandler oh = new OptionHandler("Node Properties");
    // Add a number of different option items to this handler.
    oh.addString("Label", nr.getLabelText());
    // Define a set of possible values for the enum option item.
    String shapeType[] = {"Rectangle", "Rounded Rectangle", "Ellipse"};
    int tmp = -1;
    if (nr instanceof ShapeNodeRealizer)
        tmp = (int)((ShapeNodeRealizer)nr).getShapeType();
    oh.addEnum("Shape Type", shapeType, tmp);

    // Return the created option handler.
    return oh;
}
```

## Internationalization and Localization

The settings framework supports the task of localizing option handlers by means of so-called GUI factories, i.e., implementations of interface `GuiFactory`. A GUI factory can be registered with an `OptionHandler` object, which will then use it to replace the names of option items, enumeration values, and also the option handler's name itself by localized versions thereof. [Figure 9.7, “German localized option handler” \[291\]](#) depicts the German localized version of an option handler in dialog presentation.

**Figure 9.7. German localized option handler**


Example 9.7, “Option handler internationalization” [291] shows the option handler’s setup where option item names, etc. are given placeholder names.

### Note

To emphasize their purpose, the placeholder names consist of all-caps letters and under-scores.

### Example 9.7. Option handler internationalization

```
public OptionHandler createI18nOptionHandlerForNode(NodeRealizer nr)
{
    // Create a new OptionHandler object.
    OptionHandler oh = new OptionHandler("NODE_PROPERTIES");
    // Add a number of different option items to this handler.
    oh.addString("LABEL", nr.getLabelText());
    // Define a set of possible values for the enum option item.
    String shapeType[] = {"RECTANGLE", "ROUNDED_RECTANGLE", "ELLIPSE"};
    int tmp = -1;
    if (nr instanceof ShapeNodeRealizer)
        tmp = (int)((ShapeNodeRealizer)nr).getShapeType();
    oh.addEnum("SHAPE_TYPE", shapeType, tmp);
    // Return the created option handler.
    return oh;
}
```

Commonly, localized text is stored in resource bundle properties files that contain key/value pairs where the value holds the actual localization. Class  `ResourceBundleGuiFactory` loads such a resource bundle properties file and makes the contents accessible. Example 9.8, “Localization using class  `ResourceBundleGuiFactory`” [291] demonstrates how a  `ResourceBundleGuiFactory` instance is used in conjunction with an option handler.

### Example 9.8. Localization using class `ResourceBundleGuiFactory`

```
public void localizeOptionHandler(OptionHandler oh,
                                  String language, String country)
{
    // Set the locale as given by 'language' and 'country'.
    Locale.setDefault(new Locale(language, country));

    try {
        // Create a new ResourceBundleGuiFactory object which retrieves localized
        // Strings from a resource bundle.
        ResourceBundleGuiFactory rbgf = new ResourceBundleGuiFactory();

        // Use the GUI factory to load the information specific to this class (more
        // specifically, its option handlers).
        rbgf.addBundle("MyOptionHandlerLocalization");

        // Register the GUI factory with the option handler, so that it can
        // retrieve the localized information.
        oh.setGuiFactory(rbgf);
    }
    catch (MissingResourceException mrEx) {
        System.err.println(mrEx);
    }
}
```

[Example 9.9, “Localization bundle contents” \[292\]](#) shows the contents of both a German and an English resource bundle properties file. The placeholders that have been set in [Example 9.7, “Option handler internationalization” \[291\]](#) are used to define “paths” that uniquely identify any name used within the option handler. A name’s “path” is the result of a dot-separated concatenation of all placeholders of either option sections and/or option items that are “above” it in the option handler’s element hierarchy, followed by the placeholder of the name itself. The root element of this element hierarchy is the option handler’s name.

Note that names of enumeration values are identified using an additional qualifier, .VALUE, which is inserted after the placeholder of the enumeration option item’s name.

### Example 9.9. Localization bundle contents

```
German resource bundle file 'MyOptionHandlerLocalization_de.properties'
=====
# Sample ResourceBundle properties file

NODE_PROPERTIES = Knotenattribute
NODE_PROPERTIES.LABEL = Beschriftung
NODE_PROPERTIES.SHAPE_TYPE = Form
NODE_PROPERTIES.SHAPE_TYPE.VALUE.RECTANGLE      = Rechteck
NODE_PROPERTIES.SHAPE_TYPE.VALUE.ROUNDED_RECTANGLE = Abgerundetes Rechteck
NODE_PROPERTIES.SHAPE_TYPE.VALUE.ELLIPSE          = Ellipse
NODE_PROPERTIES.SHAPE_TYPE.TOOLTIP = Auswahl m\u00fcliglicher Knotenformen

English resource bundle file 'MyOptionHandlerLocalization.properties'
=====
# Sample ResourceBundle properties file

NODE_PROPERTIES = Node Properties
NODE_PROPERTIES.LABEL = Label
NODE_PROPERTIES.SHAPE_TYPE = Shape Type
NODE_PROPERTIES.SHAPE_TYPE.VALUE.RECTANGLE      = Rectangle
NODE_PROPERTIES.SHAPE_TYPE.VALUE.ROUNDED_RECTANGLE = Rounded Rectangle
NODE_PROPERTIES.SHAPE_TYPE.VALUE.ELLIPSE          = Ellipse
NODE_PROPERTIES.SHAPE_TYPE.TOOLTIP = Selection of possible shapes
```

Using another qualifier, .TOOLTIP, which can be appended to any option item’s path, presents a convenient possibility to add a localized tooltip text for an option item without actually defining such a text in code.

## Serialization of Settings

The settings framework supports the convenient (de)serialization of an option handler’s entire values. Class [PropertiesIOHandler](#) is an implementation of interface [OptionsIOHandler](#) that can be used to read settings from/write settings to a `java.util.Properties` object.

[Example 9.10, “Reading an option handler’s settings from a file” \[292\]](#) demonstrates deserialization of such a `Properties` object from a file.

### Example 9.10. Reading an option handler’s settings from a file

```
public Properties deserializeOH(OptionHandler oh)
{
    // Create a new Properties object.
    Properties p = new Properties();

    // Use the Properties object to load the option handler's settings from a file.
    try {
        // Initialize the given Properties object from a file with the given name.
        FileInputStream fis = new FileInputStream("MySettings.properties");
        p.load(fis);
        fis.close();
    }
    catch (IOException ioEx) {
        ioEx.printStackTrace();
    }

    // Register a PropertiesIOHandler with the option handler.
    // Upon creation, this handler is initialized with the Properties object that
    // holds the values for the option items.
    oh.setOptionsIOHandler(new PropertiesIOHandler(p));
    // Read the values for the option items from the Properties object (using the
    // services of the PropertiesIOHandler).
    oh.read();
}
```

```
    return p;
}
```

Example 9.11, “Writing an option handler's settings to file” [293] demonstrates serialization of a given Properties object to a file.

### Example 9.11. Writing an option handler's settings to file

```
public void serializeOH(OptionHandler oh, Properties p)
{
    // Install an IOHandler that
    oh.setOptionsIOHandler(new PropertiesIOHandler(p));

    // Store the option handler's settings to a file.
    try {
        // Write the given Properties object to file using the given name and
        // header information.
        FileOutputStream fos = new FileOutputStream("MySettings.properties");
        p.store(fos, "My saved settings.");
        fos.flush();
        fos.close();
    }
    catch (IOException ioEx) {
        ioEx.printStackTrace();
    }
}
```

## Tutorial Demo Code

The tutorial demo application [NodePropertyEditorDemo.java](#) shows both creation and usage of an option handler, synchronization between model and view, and also use of the "undefined value" state for option items. [OptionHandlerDemo.java](#) is an extensive demo that focusses on multiple editors and their synchronization with the model.

[DiagonalLayoutModule.java](#) shows internationalization and localization (or i18n/l10n for short) aspects, as well as the (de)serialization of option handler settings. For i18n/l10n see also the resource bundle properties files [DiagonalLayoutModule.properties](#) or [OptionHandlerDemo.properties](#).

# Advanced Topics

## Synchronizing Multiple Editors

The Model-Delegate pattern that is applied by the settings framework allows multiple editors (i.e., multiple views) concurrent access to the model. In order to have each compound editor with its set of embedded item editors display the current values and state from the model, it is necessary these are propagated in time.

Interface `ItemEditor` provides the methods listed in [API Excerpt 9.8, “Methods to control implicit synchronization” \[294\]](#) to support implicit synchronization of values and state whenever they are changed in the model, i.e., whenever an option item's value or its state is modified.

### API Excerpt 9.8. Methods to control implicit synchronization

```
boolean isAutoAdopt()
boolean isAutoCommit()

void setAutoAdopt(boolean autoAdopt)
void setAutoCommit(boolean autoCommit)
```

As mentioned in [the section called “Synchronizing Editors and Option Items” \[287\]](#) a compound editor that is created by class `TableEditorFactory` by default supports both "push" and "pull" semantics with value and state change propagation. To achieve implicit synchronization for other compound editors also, code similar to that presented in [Example 9.12, “Setting embedded editors to auto-adopt state” \[294\]](#) can be used. It sets each embedded item editor of a given compound editor to auto-adopt state, so that it automatically "pulls" the current value and state of its respective option item whenever it has been modified.

### Example 9.12. Setting embedded editors to auto-adopt state

```
public void setAutoAdopt(Editor editor, boolean autoAdopt)
{
    if (editor instanceof CompoundEditor)
    {
        // Iterate over all embedded editors of this compound editor.
        for (Iterator it = ((CompoundEditor)editor).editors(); it.hasNext(); )
        {
            // Recurse.
            setAutoAdopt((Editor)it.next(), autoAdopt);
        }
    }
    else if (editor instanceof ItemEditor)
    {
        // Set the autoAdopt property on this item editor.
        ((ItemEditor)editor).setAutoAdopt(autoAdopt);
    }
}
```

## Tutorial Demo Code

`OptionHandlerDemo.java` is an extensive tutorial demo application that shows multiple editors and their synchronization.

# Chapter 10. GraphML Extension Package

The yFiles GraphML extension package enables the graph visualization library yFiles to read and write graphs in GraphML and Zip-compressed GraphML file format.

## GraphML

The [GraphML](#) file format results from the joint effort of the graph drawing community to define a common format for exchanging graph structure data. It uses an XML-based syntax and supports the entire range of possible graph structure constellations including, most notably, hierarchically organized graphs.

### What GraphML Looks Like

An abbreviated excerpt of a GraphML file that shows the encoding of a graph is presented in [Example 10.1, “Abbreviated GraphML representation” \[295\]](#). The basic graph structure is encoded using the GraphML elements `<graph>`, `<node>`, and `<edge>`. Each of these elements has an XML attribute `id` whose value is used to uniquely identify graphs, nodes, and edges within a GraphML file. XML attributes `source` and `target` which are part of the `<edge>` element, reference unique node IDs to indicate both source and target node of an edge.

[Example 10.1, “Abbreviated GraphML representation” \[295\]](#) also shows yFiles-specific enhancements to the GraphML file format that describe the visual representation of a graph. These enhancements are nested within the `<data>` element of a node or an edge.

### Example 10.1. Abbreviated GraphML representation

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- This file was written by the JAVA GraphML Library. -->
<graphml xmlns="http://graphml.graphdrawing.org/xmlns/graphml"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns/graphml
  http://www.yworks.com/xml/schema/graphml/1.0/ygraphml.xsd"
  xmlns:y="http://www.yworks.com/xml/graphml">
  <key id="d0" for="node" yfiles.type="nodegraphics"/>
  <key id="d1" for="edge" yfiles.type="edgegraphics"/>
  <graph id="G" edgedefault="directed">
    <node id="n0">
      <data key="d0">
        <y:ShapeNode>
          <y:Geometry x="170.5" y="-15.0" width="59.0" height="30.0"/>
          <y:Fill color="#CCCCFF" transparent="false"/>
          <y:BorderStyle type="line" width="1.0" color="#000000"/>
          <y:NodeLabel>January</y:NodeLabel>
          <y:Shape type="rectangle"/>
        </y:ShapeNode>
      </data>
    </node>
    <node id="n1"/>
    <edge id="e1" source="n1" target="n0">
      <data key="d1">
        <y:PolyLineEdge>
          <y:Path sx="0.0" sy="-15.0" tx="29.5" ty="0.0">
            <y:Point x="425.0" y="0.0"/>
          </y:Path>
          <y:LineStyle type="line" width="1.0" color="#000000"/>
          <y:Arrows source="none" target="standard"/>
          <y:EdgeLabel>Happy New Year!</y:EdgeLabel>
          <y:BendStyle smoothed="false"/>
        </y:PolyLineEdge>
      </data>
    </edge>
  </graph>
</graphml>
```

The GraphML file format is also described in the [GraphML Primer](#). The definition of the XML structure can be seen from the XML schema definition files [graphml-structure-1.0rc.xsd](#) and [graphml-attributes-1.0rc.xsd](#), respectively.

## GraphML Default Extension Mechanism

The GraphML default extension mechanism allows to declare so-called GraphML attributes that can be used to conveniently store additional information in a GraphML file. In its original form,

this mechanism directly supports additional data of type boolean, int, long, float, double, and String. By means of customization to the default mechanism, however, GraphML attributes can be used to store data of arbitrary complexity and type.

A GraphML attribute is declared using the `<key>` element which comprises a unique identifier as well as scope, name, and the domain of values (i.e., the actual type) for the attribute. [Table 10.1, “XML attributes for GraphML `<key>` element” \[296\]](#) lists the set of XML attributes and describes their function within the declaration of a GraphML attribute. Note that in a GraphML file, `<key>` elements must be placed before any `<graph>` element.

**Table 10.1. XML attributes for GraphML `<key>` element**

XML Attribute	Value Domain	Description
<code>id</code>	NMTOKEN	Uniquely identifies the GraphML attribute declaration within a GraphML file. Required to enable the GraphML attribute look-up mechanism.
<code>for</code>	one of "graph", "node", or "edge"	Determines the scope of the GraphML attribute.
<code>attr.name</code>	NMTOKEN	Identifying name for the GraphML attribute that can be used by an application.
<code>attr.type</code>	one of "boolean", "int", "long", "float", "double", or "string"	Determines the domain for the values (i.e., the actual type) of the GraphML attribute.

Actual values for a GraphML attribute are defined using the `<data>` element which is nested within the GraphML elements `<graph>`, `<node>`, or `<edge>`. Required with each `<data>` element is XML attribute `key` which is used by a look-up mechanism to find the proper GraphML attribute declaration for the contents of a given `<data>` element. See also [Table 10.2, “XML attribute for GraphML `<data>` element” \[296\]](#) for a brief explanation of XML attribute `key`.

**Table 10.2. XML attribute for GraphML `<data>` element**

XML Attribute	Value Domain	Description
<code>key</code>	NMTOKEN	Refers to the declaration of a GraphML attribute by matching its unique ID, i.e., the value given for the <code>id</code> attribute of a GraphML <code>&lt;key&gt;</code> element.

This GraphML attribute look-up mechanism is responsible for matching the value given for the `key` attribute of a `<data>` element to the unique ID defined by the `id` attribute of a `<key>` element. The mapping that is established by this match is essential for delegating the parsing of a given `<data>` element to the proper parser code that handles the specific contents. [Example 10.2, “GraphML attribute look-up mechanism” \[296\]](#) shows the correct setup to refer to a specific GraphML attribute when defining its values using the `<data>` element.

### Example 10.2. GraphML attribute look-up mechanism

```

<!-- Declaration of a GraphML attribute. -->
<key id="someValidIdentifier" ... />

<!-- Referring to a specific GraphML attribute by matching the (unique) -->
<!-- identifier of its declaration. -->
<data key="someValidIdentifier">
...
</data>

```

## Customizing the GraphML Extension Mechanism

The GraphML default extension mechanism can be customized to enable GraphML attributes that hold arbitrarily complex data. The basic technique to enhance the extension mechanism is to define a new XML attribute with the `<key>` element and nest user-defined XML elements inside the `<data>` element.

The newly introduced XML attribute plays a crucial role when reading GraphML file format: All <key> elements available in a file are parsed early on, and their respective set of XML attributes is handed over for inspection to any registered parser. Parser logic that corresponds to a GraphML attribute declaration then signals acceptance based on the outcome of this inspection. Subsequently, when the <data> elements of the file are parsed, the GraphML attribute look-up mechanism is used to find the matching unique ID of a <key> element, which in turn selects the corresponding parser logic that handles the contents of the <data> element.

The yFiles GraphML extension package defines the additional XML attribute `yfiles.type` to have the GraphML parser logic branch to code that handles yFiles-specific complex data such as that relating to the visual representation of a `Graph2D` from a GraphML file. As listed in [Table 10.3, “Valid combinations for XML attributes for and `yfiles.type`” \[297\]](#) there is a set of values predefined for `yfiles.type`, where each is valid only in conjunction with an appropriate value for XML attribute `for`.

**Table 10.3. Valid combinations for XML attributes for and `yfiles.type`**

Combination of Values	Description
<code>for= "node"</code>	<code>yfiles.type= "nodegraphics"</code>
<code>for= "edge"</code>	<code>yfiles.type= "edgegraphics"</code>
<code>for= "graph"</code>	<code>yfiles.type= "postprocessors"</code>
<code>for= "edge"</code>	<code>yfiles.type= "portconstraints"</code>

The declaration of the GraphML attributes that hold all data relating to the visual representation of nodes and edges is shown in [Example 10.3, “yFiles-specific GraphML attributes for realizer-related data” \[297\]](#)

### Example 10.3. yFiles-specific GraphML attributes for realizer-related data

```
<!-- Node realizer-related data. -->
<key id="d0" for="node" yfiles.type="nodegraphics" />

<!-- Edge realizer-related data. -->
<key id="d1" for="edge" yfiles.type="edgegraphics" />
```

Any data specific to the realizer of a graph element is nested within the <data> element for the respective node or edge. [Example 10.4, “Edge realizer-related data” \[297\]](#) shows the entire encoding for an edge realizer.

### Example 10.4. Edge realizer-related data

```
<!-- Edge realizer-related data nested within the <data> element of its -->
<!-- respective edge. -->
<edge id="e0" source="n0" target="n1">
  <data key="d1">
    <y:PolyLineEdge>
      <y:Path sx="0.0" sy="0.0" tx="0.0" ty="0.0" />
      <y:LineStyle type="line" width="1.0" color="#000000" />
```

```
<y:Arrows source="none" target="standard"/>
<y:EdgeLabel x="47.0" y="-21.555956740606405"
    width="4.0" height="4.0"
    visible="true"
    alignment="center"
    fontFamily="Dialog" fontSize="12" fontStyle="plain"
    textColor="#000000"
    modelName="six_pos" modelPosition="tail"
    preferredPlacement="anywhere"
    distance="2.0" ratio="0.5">A simple edge label</y:EdgeLabel>
<y:BendStyle smoothed="false"/>
</y:PolyLineEdge>
</data>
</edge>
```

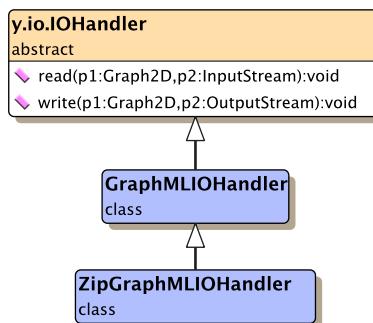
The XML structure of the enhanced GraphML file format as defined by the yFiles GraphML extension package can be seen from the following XML schema definition files:

- [ygraphml.xsd](#) declares additional XML elements to be nested within GraphML <data> elements
- [ygraphics.xsd](#) defines new XML elements that encode predefined yFiles realizer implementations and also all kinds of data types to be used with these newly defined XML elements
- [yprocessors.xsd](#) defines new XML elements that encode so-called post-processors

# Using the yFiles GraphML Extension Package

Class `GraphMLIOHandler` from the extension package provides the yFiles library with I/O support for the GraphML file format. It is a direct subclass of abstract class `IOHandler` from package `y.io` of the yFiles library, see also [Figure 10.1, “GraphML I/O handler class hierarchy”](#) [299]

**Figure 10.1. GraphML I/O handler class hierarchy**



The code fragment in [Example 10.5, “Instantiating a GraphMLIOHandler”](#) [299] shows how to instantiate and use a `GraphMLIOHandler` to write an instance of type `Graph2D` to a GraphML-encoded file. Class `ZipGraphMLIOHandler` can be used in a similar manner to write Zip-compressed GraphML files which are up to 50 times smaller than their uncompressed counterparts.

## Example 10.5. Instantiating a GraphMLIOHandler

```

public void encodeAsGraphML(Graph2D graph)
{
    // Instantiate a GraphML I/O handler and write the graph to file.
    try {
        IOHandler ioh = new GraphMLIOHandler();
        ioh.write(graph, "MyGraphML.graphml");
    }
    catch (IOException ioEx) {
        // Something went wrong. Complain.
    }
}
  
```

## Direct Support for Simple Data Types

Using the GraphML default extension mechanism to add data of simple type such as boolean, int, long, float, double, or String, can be done by means of the methods from class `GraphMLIOHandler` as listed in [API Excerpt 10.1, “GraphMLIOHandler support for the GraphML default extension mechanism”](#) [299]

### API Excerpt 10.1. GraphMLIOHandler support for the GraphML default extension mechanism

```

// Storing additional data with a graph.
void addGraphAttribute(DataProvider out, DataAcceptor in, String attrName,
                      int type)

// Storing additional data with the nodes of a graph.
void addAttribute(NodeMap map, String attrName, int type)
void addNodeAttribute(DataProvider out, DataAcceptor in, String attrName,
                      int type)

// Storing additional data with the edges of a graph.
void addAttribute(EdgeMap map, String attrName, int type)
void addEdgeAttribute(DataProvider out, DataAcceptor in, String attrName,
                      int type)
  
```

These methods are used to register the data accessors that will be accessed whenever any data is actually read from or written to a GraphML file. They also determine the scope of a GraphML attribute (i.e., whether it holds values for a graph, the nodes, or the edges), its identifying name, and the domain for its values. Note that this domain, i.e., the actual type of the GraphML attribute, can be set using the constants `TYPE_BOOLEAN`, `TYPE_INT`, `TYPE_LONG`, `TYPE_FLOAT`, `TYPE_DOUBLE`, and `TYPE_STRING` defined in interface [AttributeConstants](#).

### Example 10.6. Adding a GraphML attribute to GraphMLIOHandler

```
void addNodeColorGraphMLAttribute(GraphMLIOHandler handler, NodeMap nm)
{
    // The given node map will be accessed by the reader and writer classes
    // whenever a GraphML file is read or written.
    handler.addAttribute(nm, "color", AttributeConstants.TYPE_STRING);
}
```

[Example 10.6, “Adding a GraphML attribute to GraphMLIOHandler” \[300\]](#) shows how to define a GraphML attribute that stores additional data of simple type for all nodes of a graph. [Example 10.7, “Outline of a GraphML file with GraphML attribute of simple type” \[300\]](#) is an excerpt of a GraphML file that shows both the resulting GraphML attribute declaration and also the definition of its values with each of the nodes.

### Example 10.7. Outline of a GraphML file with GraphML attribute of simple type

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="...">
    ...
    <!-- Definition of a GraphML attribute to store additional data for a -->
    <!-- graph's nodes. -->
    <key id="d0" for="node" attr.name="color" attr.type="string"/>

    <graph id="G" edgedefault="directed">
        ...
        <!-- A node that has a <data> element referring to the GraphML attribute -->
        <!-- "d0." The node's value (of type string) is "green." -->
        <node id="n0">
            <data key="d0">green</data>
        </node>
        <node id="n1">
            <data key="d0">black</data>
        </node>
        ...
    </graph>
</graphml>
```

## Important

The yFiles GraphML extension package makes use of the data accessor implementations of the yFiles library to store the values of a GraphML attribute. However, these implementations provide no support for either `TYPE_LONG` or `TYPE_FLOAT`. Therefore, when storing the values of a GraphML attribute in a given data acceptor, `TYPE_LONG` is type-casted to `int` and `TYPE_FLOAT` is type-casted to `double`.

## General Support for Structured Data

Storing structured data, i.e., more than just simple data types with a GraphML attribute is delegated by `GraphMLIOHandler` to subordinate input and output handlers that are specialized for handling specific types of complex data, like, e.g., realizer-related information. These subordinate handlers are implementations of interfaces [OutputHandler](#) and [DOMInputHandler](#) that are registered with class `GraphMLIOHandler`. They are responsible both for serializing and deserializing the structured data and also for defining an additional XML attribute that is used to enhance the `<key>` element as described in [Customizing the GraphML Extension Mechanism](#).

The methods listed in [API Excerpt 10.2, “Getting subordinate input and output handlers registered with GraphMLIOHandler” \[301\]](#) can be used to get all subordinate input and output hand-

lers for either graphs, nodes, or edges that are registered with a GraphMLIOHandler. New custom handlers can be registered by simply adding them to one of the returned collections.

### API Excerpt 10.2. Getting subordinate input and output handlers registered with GraphMLIOHandler

```
// Getter methods for the registered subordinate input and output handlers of a
// GraphMLIOHandler instance.
Collection<?> getInputHandlers()
Collection<?> getOutputHandlers(int scope)
```

Note that the actual scope for input handlers is determined automatically at parse time while the scope for output handlers has to be set using the constants SCOPE\_GRAPH, SCOPE\_NODE, and SCOPE\_EDGE defined in interface `GraphMLConstants`.

[Example 10.8, “Adding custom subordinate handlers with node scope” \[301\]](#) shows how custom handlers with node scope are registered with a GraphMLIOHandler.

### Example 10.8. Adding custom subordinate handlers with node scope

```
// 'handler' is of type GraphMLIOHandler.

// Register a custom subordinate output handler for nodes.
MyNodeOutputHandler nodeOut = new MyNodeOutputHandler();
handler.getOutputHandlers(GraphMLConstants.SCOPE_NODE).add(nodeOut);

// Register a custom subordinate input handler for nodes.
MyNodeInputHandler nodeIn = new MyNodeInputHandler();
handler.getInputHandlers().add(nodeIn);
```

[Table 10.4, “Helper classes for GraphML \(de\)serialization” \[301\]](#) lists useful helper classes for (de)serializing data to/from a GraphML file. They provide convenient support for generating XML-conformant content and parsing valid XML data for many data types commonly used by the yFiles library.

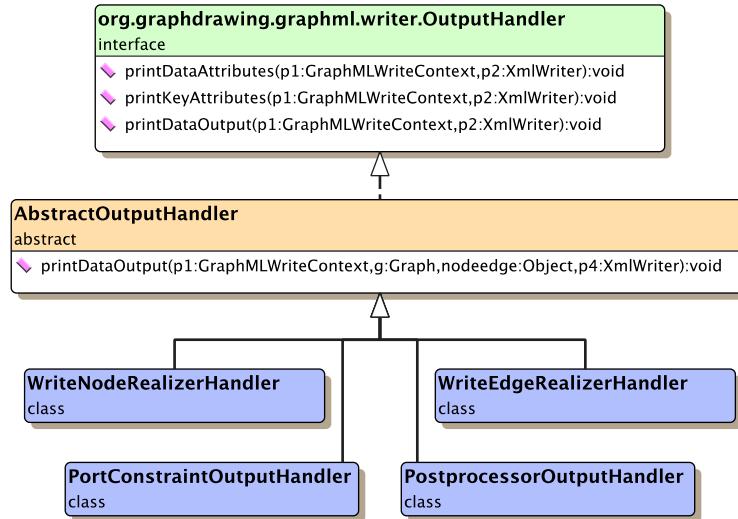
**Table 10.4. Helper classes for GraphML (de)serialization**

Classname	Description
<code>DomXmlWriter</code>	Implementation of interface <code>XmlWriter</code> that is used for writing GraphML. This class is used by GraphMLIOHandler by default.
<code>GraphicsSerializationToolkit</code>	Can be used for reading and writing GraphML. Offers methods to (de)serialize yFiles-specific types and also common data types such as <code>java.awt.Color</code> or <code>java.awt.Insets</code> .

### Subordinate Output Handlers

Subordinate output handlers are responsible for introducing an appropriate XML attribute with the GraphML `<key>` element and for generating any user-defined XML elements which are nested within GraphML `<data>` elements. The XML attribute serves as the identifying marker to choose the corresponding parser logic when the GraphML attribute declaration has been parsed. The user-defined XML elements encode the respective structured data. Abstract class `AbstractOutputHandler` provides a convenient base class for actual implementations. [Figure 10.2, “GraphML output handler class hierarchy” \[301\]](#) shows the class hierarchy of predefined output handler implementations.

**Figure 10.2. GraphML output handler class hierarchy**



**Table 10.5. Predefined output handlers** [302] lists the predefined output handlers of the yFiles GraphML extension package.

**Table 10.5. Predefined output handlers**

Classname	Description
WriteNodeRealizerHandler	Dispatches the serialization of node realizer data to so-called node realizer serializers; is registered with GraphMLIOHandler by default.
WriteEdgeRealizerHandler	Dispatches the serialization of edge realizer data to so-called edge realizer serializers; is registered with GraphMLIOHandler by default.
PostprocessorOutputHandler	Serializes structured data related to post-processor configuration; is registered with GraphMLIOHandler by default.
PortConstraintOutputHandler	Serializes structured data related to port constraints and edge groups; has to be registered with GraphMLIOHandler.

**Example 10.9. Custom subordinate output handler with node scope** [302] presents a custom output handler with node scope that writes user-defined data to a GraphML file. Note that the content of the <data> element is written using the output method `printDataOutput` of abstract class `AbstractOutputHandler` which has extended signature.

`AbstractOutputHandler`'s method with extended signature is a convenience that enables easy access to the actual graph and the graph element that is to be written. The graph element is either a node or an edge according to the scope of the output handler, hence it can safely be type-casted to `y.base.Node` or `y.base.Edge`, respectively.

### Example 10.9. Custom subordinate output handler with node scope

```

public class MyNodeOutputHandler extends AbstractOutputHandler {
    public void printKeyAttributes(GraphMLWriteContext context,
                                   XMLWriter writer) {
        // Additional XML attribute to enhance the GraphML <key> element.
        // The leading blank before the attribute's name is added by the writer of
        // the <key> element.
        writer.writeAttribute("myCustomType", "true");
    }

    // Convenience method that provides access to the graph and also its nodes or
    // edges according to the output handler's scope.
    public void printDataOutput(GraphMLWriteContext context, Graph graph,
                               Object nodeedge, XMLWriter writer) {
        NodeRealizer nr = ((Graph2D)graph).getRealizer((y.base.Node)nodeedge);
    }
}
  
```

```

        writer.writeStartElement("myData", null);
        writer.writeAttribute("value", nr.getHeight() * nr.getWidth());
        writer.writeEndElement();
    }

    // Not supported yet.
    public void printKeyOutput(GraphMLWriteContext context, XMLWriter writer) {}
}
    
```

## Subordinate Input Handlers

Subordinate input handlers are responsible for inspecting the set of XML attributes of GraphML <key> elements and for parsing the contents of GraphML <data> elements. The inspection yields whether an input handler provides parser logic that corresponds to a GraphML attribute declaration, i.e., whether the parser logic can handle the contents of GraphML <data> elements that refer to the given <key> element by matching its unique ID. If true, the GraphML parser logic branches to the input handler for parsing (the contents of) all such GraphML <data> elements. Abstract class `AbstractDOMInputHandler` provides a convenient base class for actual implementations. Figure 10.3, “GraphML DOM input handler class hierarchy” [303] shows the class hierarchy of predefined DOM input handler implementations.

**Figure 10.3. GraphML DOM input handler class hierarchy**

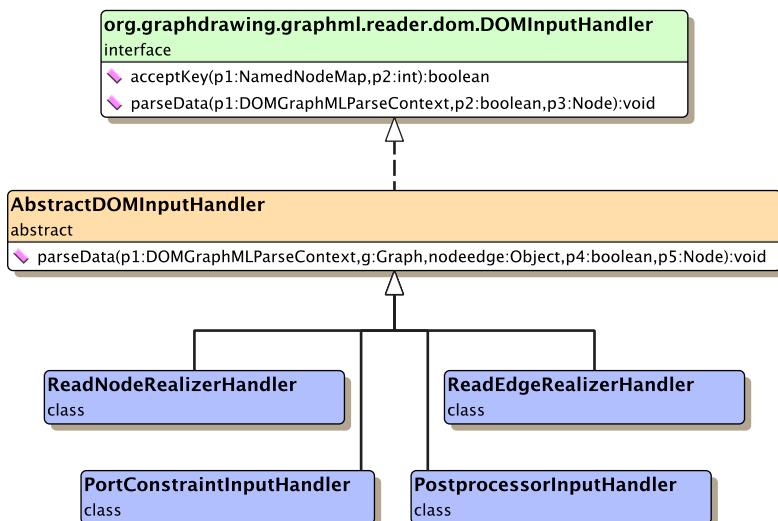


Table 10.6, “Predefined input handlers” [303] lists the predefined input handlers of the yFiles GraphML extension package.

**Table 10.6. Predefined input handlers**

Classname	Description
ReadNodeRealizerHandler	Dispatches the deserialization of node realizer data to so-called node realizer serializers; is registered with GraphMLIOHandler by default.
ReadEdgeRealizerHandler	Dispatches the deserialization of edge realizer data to so-called edge realizer serializers; is registered with GraphMLIOHandler by default.
PostprocessorInputHandler	Deserializes structured data related to post-processor configuration; is registered with GraphMLIOHandler by default.
PortConstraintInputHandler	Deserializes structured data related to port constraints and edge groups; has to be registered with GraphMLIOHandler.

A subordinate input handler signals whether it can handle the contents of GraphML <data> elements through its `acceptKey` method. When parsing, it extracts the actual data by processing the DOM (Document Object Model) structure of a GraphML file. To this end it makes use of package `org.w3c.dom`, the component API of the Java API for XML Processing. [Example 10.10](#), “Custom input handler with node scope” [304] presents a custom input handler with node scope that reads user-defined data from a GraphML file. Note that the content of the <data> element is read using the input method `parseData` of abstract class `AbstractDOMInputHandler` which has extended signature.

`AbstractDOMInputHandler`'s method with extended signature is a convenience that enables easy access to the actual graph and the graph element that is to be read. The graph element is either a node or an edge according to the scope of the input handler, hence it can safely be type-casted to `y.base.Node` or `y.base.Edge`, respectively.

### Example 10.10. Custom input handler with node scope

```
public class MyNodeInputHandler extends AbstractDOMInputHandler {
    // Assume (parsing) responsibility only for those <data> elements that refer
    // to GraphML attribute declarations with specific additional XML attribute.
    public boolean acceptKey(NamedNodeMap map, int scopeType) {
        if (scopeType != GraphMLConstants.SCOPE_NODE) {
            return false;
        }
        // 'map' holds the XML attributes of a <key> element.
        Node node = map.getNamedItem("myCustomType");
        return ((node == null) ? false : "true".equals(node.getNodeValue()));
    }

    // Parse the <data> element.
    protected void parseData(DOMGraphMLParseContext context, Graph graph,
                             Object nodeedge, boolean defaultMode,
                             org.w3c.dom.Node domNode) {
        // Default mode is not supported.
        if (defaultMode) {
            return;
        }

        String label = "";
        // 'domNode' holds the <data> element, its XML attributes, and all XML
        // elements nested within.
        org.w3c.dom.NodeList children = domNode.getChildNodes();
        if (children != null) {
            for (int i = 0; i < children.getLength(); i++) {
                org.w3c.dom.Node n = children.item(i);
                if (n.getNodeType() == org.w3c.dom.Node.ELEMENT_NODE &&
                    "myData".equals(n.getLocalName())) {
                    label = parseMyDataElement(n);
                }
            }
        }
        ((Graph2D)graph).getRealizer((y.base.Node)nodeedge).setLabelText(label);
    }

    // Parse the attribute of <myData> element.
    String parseMyDataElement(org.w3c.dom.Node domNode) {
        NamedNodeMap nm = domNode.getAttributes();
        org.w3c.dom.Node a = nm.getNamedItem("value");
        String txt = "Node's area is ";
        txt += ((a == null) ? "n/a." : "" + a.getNodeValue() + " square pixels.");
        return txt;
    }

    // Not supported yet.
    protected void applyDefault(DOMGraphMLParseContext c, Graph g, Object noed) {}
}
```

## Support for Custom Realizer Implementations

The yFiles GraphML extension package enables convenient (de)serialization of node and edge realizers by means of so-called realizer serializers. Realizer serializers encapsulate the logic for encoding and parsing the different kinds of realizer implementations available with the yFiles library.

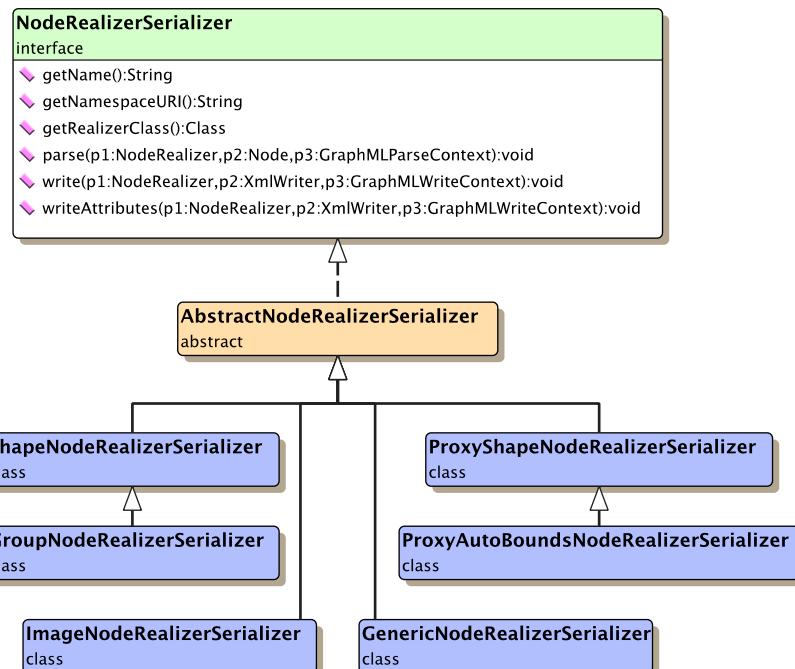
The serializers for both node realizer and edge realizer implementations are registered with the corresponding subordinate input and output handlers, i.e., with classes

- [ReadNodeRealizerHandler](#) and [WriteNodeRealizerHandler](#) (node realizer serializers) or
- [ReadEdgeRealizerHandler](#) and [WriteEdgeRealizerHandler](#) (edge realizer serializers).

These subordinate handlers are used by class `GraphMLIOHandler` and serve to dispatch both serialization and deserialization of specific realizer types to the matching serializer from the set of their registered serializers.

[Figure 10.4, “GraphML node realizer serializer class hierarchy” \[305\]](#) depicts the hierarchy of node realizer serializer classes that are predefined by the yFiles GraphML extension package. Note that it reflects the hierarchy of the node realizer classes from package `y.view`.

**Figure 10.4. GraphML node realizer serializer class hierarchy**



### Note

The hierarchy of predefined edge realizer serializer classes is analogous to that of the node realizer serializer classes as shown in [Figure 10.4, “GraphML node realizer serializer class hierarchy” \[305\]](#).

Adding support for custom realizer implementations to the GraphML file format can be done by registering appropriate custom serializers with the `RealizerSerializerManager` that is associated with a `GraphMLIOHandler` instance. [API Excerpt 10.3, “Getter method for the RealizerSerializerManager of a GraphMLIOHandler instance” \[305\]](#) shows the getter method from class `GraphMLIOHandler` that returns the `RealizerSerializerManager` instance.

### API Excerpt 10.3. Getter method for the RealizerSerializerManager of a GraphMLIOHandler instance

```
RealizerSerializerManager getRealizerSerializerManager()
```

[API Excerpt 10.4, “Methods to register custom realizer serializers” \[306\]](#) lists the methods from class `RealizerSerializerManager` to register serializers for node and edge realizer implementations.

## API Excerpt 10.4. Methods to register custom realizer serializers

```
void addNodeRealizerSerializer(NodeRealizerSerializer nrs)
void addEdgeRealizerSerializer(EdgeRealizerSerializer ers)
```

Any serializer that is registered using one of these methods is automatically added to the sets of serializers available with the corresponding input and output handlers of a given GraphMLIOHandler instance.

The tasks of a serializer are similar to those described for subordinate input and output handlers in [Subordinate Input Handlers](#) and [Subordinate Output Handlers](#), respectively. The encoding part of a serializer is responsible for generating any user-defined XML elements which are nested within GraphML <data> elements. The parsing part extracts its data by processing the DOM (Document Object Model) structure of a GraphML file starting at the DOM node that represents a GraphML <data> elements.

Key to correctly support a custom realizer implementation with the GraphML file format is the proper definition of the XML element that is top-level inside the GraphML <data> element. This XML element is written by the subordinate output handler that initiates encoding and read by the subordinate input handler that initiates parsing of the data associated with a realizer. It serves as a means to enclose and also identify the data of a specific realizer implementation. Method [getName](#), which is defined by both node serializer and edge serializer interfaces, is used to define the enclosing top-level element. It is invoked by both output handler and input handler, respectively:

- When a GraphML file is written, the String object returned is used by the output handler as the name of the top-level XML element. (Note that method [writeAttributes](#) can be used to define additional XML attributes with the element.)
- When a GraphML file is read, the String object returned is used by the input handler to find the matching realizer serializer that knows how to parse the data of a specific realizer.

Once the matching realizer serializer is determined, the input handler uses method [createRealizerInstance](#) to instantiate a new realizer. This realizer object is then configured using the data from the GraphML file.

Tutorial demo application [CustomNodeRealizerSerializerDemo.java](#), which uses [CustomNodeRealizerSerializer.java](#), shows the code of a custom realizer serializer, how it is registered with the RealizerSerializerManager of a GraphMLIOHandler instance, and also how it is used.

## yFiles GraphML Post-processors

The yFiles GraphML extension package provides support for advanced graph post-processing functionality. So-called post-processors are embedded statements in a GraphML file that reference code which is executed after the graph defined in the file is parsed. They allow to conveniently manipulate essential aspects of a GraphML-encoded graph directly following the parsing process and prior to display.

For example, post-processors can be used to automatically compute a layout for a freshly parsed graph. Consequently, post-processors come in handy when graphs are automatically generated from data that inherently provide only structural information, i.e., that lack coordinates or even dimensional information.

A post-processor can be any class that inherits from [YModule](#) which most notably includes the layout modules from package [y.module](#), but also some classes from package [yext.graphml.processor](#). Note however, that the services of a post-processor are not limited to providing automatic layout to a graph, resizing its nodes, or setting the background of a view. In fact, any kind of code can be part of such a class.

Execution of a post-processor is started by calling its main startup method [start\(Graph2D\)](#) which is defined in class [YModule](#). The Graph2D object that is given as the parameter is the freshly parsed graph from the GraphML file.

The post-processor definitions are embedded in a GraphML file by means of a GraphML attribute that holds structured data. [Example 10.11, “yFiles-specific GraphML attribute for post-processors” \[307\]](#) shows the combination of values for XML attributes `yfiles.type` and `for`

that is used to declare the post-processor GraphML attribute with graph scope.

### Example 10.11. yFiles-specific GraphML attribute for post-processors

```
<!-- Support for post-processors for a graph. -->
<key id="d2" for="graph" yfiles.type="postprocessors"/>
```

[Example 10.12, “Post-processor setup” \[307\]](#) shows the general scheme of embedding post-processor definitions into a GraphML file. The `<y:Postprocessors>` element nested within the `<data>` element of a graph encloses any number of proper post-processor definitions, which comprise both the classname of the post-processor and the configuration for its parameters. The classname is given with the `<y:Processor>` element and the configuration is encoded using nested `<y:Option>` elements. XML attribute name of such a `<y:Option>` element denotes a correspondingly named option item, which is part of the [OptionHandler](#) object that is used to configure the post-processor's parameters.

As described above, a post-processor is invoked after the graph defined in the GraphML file has been parsed. If multiple `<y:Processor>` elements appear within a `<y:Postprocessors>` element, they constitute a post-processing chain where the actual invocation order of post-processors is determined by their order of appearance.

### Example 10.12. Post-processor setup

```
<!-- A post-processor's setup nested within the <data> element of a graph. -->
<graph id="G" edgedefault="directed">
  <data key="d2">
    <y:Postprocessors>
      <y:Processor class="yext.graphml.processor.NodeSizeAdapter">
        <y:Option name="IGNORE_WIDTHS" value="false"/>
        <y:Option name="IGNORE_HEIGHTS" value="false"/>
        <y:Option name="ADAPT_TO_MAXIMUM_NODE" value="false"/>
      </y:Processor>
    </y:Postprocessors>
  </data>
  ...
</graph>
```

The tutorial demo application [PostprocessorDemo.java](#) demonstrates the post-processor functionality provided by the yFiles GraphML extension package. Note that the demo can also be used to conveniently generate proper XML markup for predefined post-processors from either the yFiles library or the yFiles GraphML extension package.

#### Tip

If the yFiles GraphML extension package is in the class path for command line tool [GraphFormatConverter.java](#), it can be used, for example, to conveniently create GIF or JPG images from a given graph encoded in GraphML file format. Furthermore, if suitable post-processors are embedded in the GraphML file, the actual conversion that is performed by the tool can also automatically adjust the widths of all nodes or calculate a layout for the graph.

## Tutorial Demo Code

Loading and saving graphs in GraphML file format is demonstrated in the tutorial demo application [GraphMLDemo.java](#). [CustomGraphMLDemo.java](#) shows a detailed example of how the GraphML default extension mechanism can be used to read and write simple data types.

For examples how customized encoding and parsing logic can be provided, please see the following tutorial demo applications:

- [ComplexExtensionGraphMLDemo.java](#) shows how to read and write structured data. It makes use of [ComplexExtensionGraphMLIOHandler.java](#) which adds custom input and output handlers by directly subclassing GraphMLIOHandler and accessing its protected fields.
- [CustomNodeRealizerSerializerDemo.java](#) together with [CustomNodeRealizerSerializer.java](#) shows how custom realizer serializers can be used.

## Tutorial Demo Code

---

[PostprocessorDemo.java](#) demonstrates the post-processor functionality provided by the yFiles GraphML extension package.

# Advanced Topics

## yFiles XSLT Support for GraphML

The yFiles GraphML extension package provides advanced conversion services for arbitrary XML-based data. By means of a suitable XSLT stylesheet any valid XML file can be transformed into a graph structure encoded in GraphML file format.

The tutorial demo application [XMLXSLDemo.java](#) shows how to import XML files as GraphML. Predefined XSLT stylesheets are available for the following XML-based file types and information:

- Ant build scripts: [ant2graphml.xsl](#)
- OWL Web ontology data: [owl2graphml.xsl](#)
- XML data tree structure: [xmmtree2graphml.xsl](#)

Note that the predefined XSLT stylesheets make use of the yFiles post-processors functionality: obviously, none of the aforementioned types of XML data files provide coordinates or even dimensional information for its content. The XSLT stylesheets embed post-processors in each generated GraphML file which automatically calculate a layout when the encoded graph is loaded.

## Reading and Writing Graphs of Arbitrary Type

The most convenient way to use the yFiles GraphML extension package for reading and writing GraphML file format is by using the services of class GraphMLIOHandler. However, being a true yFiles IOHandler, this class requires a Graph2D object when reading or writing a file. Using the low-level functionality of the yFiles GraphML extension package directly, it is also possible to read and write graphs of arbitrary type.

Classes [DOMGraphMLParser](#) and [DirectGraphMLWriter](#) provide the low-level services to read and write GraphML format. These classes make use of so-called graph element factories and graph element providers, respectively, which are responsible for providing objects of proper type for both the graph that is to be read or written and also its nodes and edges.

Implementations of graph element factory and graph element provider that are predefined with the yFiles GraphML extension package are classes [YGraphElementFactory](#) and [YGraphElementProvider](#). These can be instantiated using an arbitrary graph type, and are then registered with DOMGraphMLParser and DirectGraphMLWriter, respectively.

Instantiating both graph element factory and graph element provider with class [Graph](#), for example, enables reading and writing graphs of this type instead of Graph2D. As a result, the GraphML file format can be used without classes that are only present in the yFiles Viewer distribution.

[Example 10.13, “Writing graphs of type y.base.Graph to a GraphML file” \[309\]](#) shows the configuration of class DirectGraphMLWriter so that it handles graphs of type Graph when writing a GraphML file.

### Example 10.13. Writing graphs of type y.base.Graph to a GraphML file

```
public void encodeGraphAsGraphML(Graph graph, OutputStream out)
{
    // Using class y.base.Graph instead of y.view.Graph2D to write GraphML file
    // format.
    YGraphElementProvider gep = new YGraphElementProvider(graph);

    // Low-level support for writing GraphML file format.
    DirectGraphMLWriter writer = new DirectGraphMLWriter();
    writer.setGraphElementProvider(gep);
    try {
        writer.write(new DomXmlWriter(out));
    }
    catch (IOException ioEx) {
        // Something went wrong. Complain.
    }
}
```

Example 10.14, “Populating graphs of type `y.base.Graph` with data read from a GraphML file” [?] shows the configuration of class `DOMGraphMLParser` so that it uses graphs of type `Graph` when reading a GraphML file.

### Example 10.14. Populating graphs of type `y.base.Graph` with data read from a GraphML file

```
public void populateGraphFromGraphML(Graph graph, InputStream in)
{
    // Using class y.base.Graph instead of y.view.Graph2D when reading GraphML
    // file format.
    YGraphElementFactory gef = new YGraphElementFactory(graph);

    // Low-level support for processing the GraphML file format's DOM structure.
    DOMGraphMLParser parser = new DOMGraphMLParser();
    parser.setGraphElementFactory(gef);
    try {
        parser.parse(in);
    }
    catch (IOException ioEx) {
        // Something went wrong. Complain.
    }
}
```

The tutorial demo application `BaseGraphSerializationDemo.java` is a simple command line program that demonstrates how the yFiles GraphML extension package can be used to read and write an instance of type `Graph`.

---

# Chapter 11. ySVG Extension Package

The yFiles extension package ySVG enables the graph visualization library yFiles to export graphs to Scalable Vector Graphics (SVG) and Zip-compressed SVG file format, respectively.

## Scalable Vector Graphics

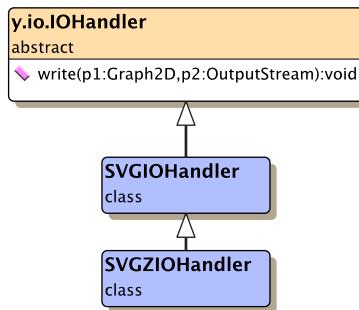
[SVG](#) is a language for describing two-dimensional graphics using XML-based syntax. It enables vector-based graphics that can be displayed in common web browsers.

The ySVG extension package uses the [Batik SVG Toolkit](#) for generating the actual SVG content.

# Using the ySVG Extension Package

Class **SVGIOHandler** from the extension package provides the yFiles library with export support for the SVG file format. It is a direct subclass of abstract class **IOHandler** from package **y.io** of the yFiles library, see also [Figure 11.1, “SVG output handler class hierarchy” \[312\]](#)

**Figure 11.1. SVG output handler class hierarchy**



The code fragment in [Example 11.1, “Instantiating a SVGIOHandler” \[312\]](#) shows how to instantiate and use a **SVGIOHandler** to export the content of a given **Graph2DView** to a SVG-encoded file. Class **SVGZIOHandler** can be used in a similar manner to write Zip-compressed SVG files which are up to 50 times smaller than their uncompressed counterparts.

## Example 11.1. Instantiating a **SVGIOHandler**

```

public void exportSVG(Graph2DView view) {
    // Instantiate a SVG output handler and export the view's content.
    try {
        IOHandler ioh = new SVGIOHandler();
        double tmpPDT = view.getPaintDetailThreshold();
        view.setPaintDetailThreshold(0.0);
        ioh.write(view.getGraph2D(), "MySVG.svg");
        view.setPaintDetailThreshold(tmpPDT);
    }
    catch (IOException ioEx) {
        // Something went wrong. Complain.
    }
}
  
```

# Using SVG Content

Besides exporting the content of a **Graph2DView** to SVG, the ySVG extension package also enables to directly use SVG content for the representation of a graph's nodes. Either of the classes **SVGNodeRealizer** or **SVGPainter** can be used to this end. Both use an **SVGModel** that holds the actual SVG content:

- **SVGPainter** is a specialized implementation of static inner interface **GenericNodeRealizer.Painter** which is used in conjunction with class **GenericNodeRealizer**. It is registered with a realizer's configuration as discussed in [the section called “Class GenericNodeRealizer” \[192\]](#).
- **SVGNodeRealizer** is a true yFiles **NodeRealizer** implementation. Tutorial demo applications **SimpleSVGNodeRealizerDemo.java** and **SVGNodeRealizerDemo.java** show how to use **SVGNodeRealizer**.

## Customizing SVG Content

The SVG content that is generated can be easily customized in order to provide interactive behavior. By means of JavaScript™ code that is embedded into the SVG Document Object Model (DOM) tree, otherwise static content can be easily augmented to support interactivity with the graph's elements, for example.

### Tutorial Demo Code

The following tutorial demo applications show how to embed JavaScript code to achieve interactive functionality for graphs exported to SVG file format:

- [HighlightConnectionsDemo.java](#) presents how to highlight incoming and outgoing edges
- [HtmlTooltipDemo.java](#) shows how to add HTML tooltips for the nodes of a graph
- [HyperlinkDemo.java](#) presents how to add hyperlinks to graph elements and their labels
- [RolloverDemo.java](#) shows how to change the representation of edges

Many of these demos can be seen in the [demo section of the ySVG extension package](#).

## **GraphML Support**

The SVG content of a SVG node realizer can be exported to GraphML file format using class **SVGNodeRealizerSerializer**. Tutorial demo application [GraphMLDemo.java](#) shows how this realizer serializer is used when writing GraphML.

---

# Chapter 12. yExport Extension Package

The yFiles extension package yExport adds support for popular vector graphics file formats to the yFiles graph visualization library. Additionally, it also allows to use the system clipboard to transfer a graph's representation between applications.

## Vector Graphics File Formats

yExport enables convenient export of the contents of a Graph2DView to a number of vector graphics file formats. [Table 12.1, “Vector graphics export file formats” \[315\]](#) lists the available output formats to save the visual representations for nodes and edges as well as any other graphical decorations visible.

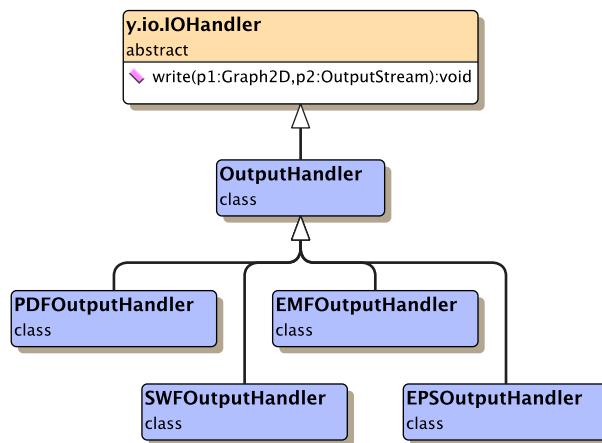
**Table 12.1. Vector graphics export file formats**

File Name Extension	Format Name	Description
pdf	Portable Document Format	Desktop publishing file format for representing documents in a device independent and display resolution independent way.
swf	Shockwave Flash	Vector graphics file format for publishing multimedia contents on the Web.
emf	Windows Enhanced Metafile	Vector graphics file format on Microsoft Windows systems.
eps	Encapsulated PostScript	PostScript document with additional restrictions that describes an image or drawing.

# Using the yExport Extension Package

Class `OutputHandler` from the extension package is the base class for all vector graphics file formats. It provides a common basis for the yExport output handlers that enhance the yFiles library. `OutputHandler` is a direct subclass of abstract class `IOHandler` from package `y.io` of the yFiles library. Figure 12.1, “Output handler hierarchy in yExport” [316] illustrates the yExport output handler hierarchy.

**Figure 12.1. Output handler hierarchy in yExport**



The code fragment in Example 12.1, “Instantiating a `PDFOutputHandler`” [316] shows how to instantiate and use a `PDFOutputHandler` to export the content of a given `Graph2DView` to a PDF-encoded file.

## Example 12.1. Instantiating a `PDFOutputHandler`

```

public void exportPDF(Graph2DView view) {
    // Instantiate a PDF output handler and export the view's content.
    try {
        IOHandler ioh = new PDFOutputHandler();
        ioh.write(view.getGraph2D(), "MyPDF.pdf");
    }
    catch (IOException ioEx) {
        // Something went wrong. Complain.
    }
}
  
```

`PDFExportDemo.java` presents an extensive tutorial demo application that shows how to export a view's contents to PDF.

Class `OutputHandler` supports passing rendering hints to the underlying rendering system. Rendering hints can be used to control the quality of text rendering, for example. API Excerpt 12.1, “Text rendering-related methods” [316] presents the methods that allow to specify the hints.

## API Excerpt 12.1. Text rendering-related methods

```

void addRenderingHint(RenderingHints.Key key, Object value)
void removeRenderingHint(RenderingHints.Key key)
  
```

# Using the System Clipboard

Class `VisualTransferable` makes available support for the system clipboard. It allows to transfer the contents of a `Graph2DView` to the clipboard using the EMF file format. In addition to this

## Using the System Clipboard

---

so-called "data flavor," optional image file formats can also be part of the transfer data in order to support EMF-unaware applications. The additional image file formats that are used as fall-back data flavors depend on the actual platform as well as the version of the Java Runtime Environment. The [VisualTransferable\(Graph2DView, boolean\)](#) constructor can be used to enable fallback data flavors as part of the transferred data.

Tutorial demo application [SystemClipboardDemo.java](#) shows how to copy the contents of a Graph2DView encoded in EMF format to the system clipboard.



---

# Appendix A. Obfuscation

This appendix covers obfuscation of yFiles classes in particular, but also obfuscation of Java code in general. Obfuscation as discussed here means *name obfuscation*.

## Why Obfuscation Matters

Generally, Java byte code shipped in .class files is inherently susceptible to reverse-engineering by simple decompilation. There are several Java byte code decompilers available that can reproduce Java source code from given byte code quite accurately.

Performing name obfuscation makes decompiled Java byte code harder to read, if not unreadable at all.

Apart from the need of protection for Java code in general, a yFiles licensee in particular is bound to the code protection requirements as stated in the license terms: the [yFiles license terms](#) explicitly require that all essential class, method, and field names of classes belonging to the yFiles library are obfuscated. The obfuscation's intended purpose, namely prevention of any unauthorized use of the library's functionality via the publicly available yFiles API, is also expressed.

Name obfuscation completely defeats any attempts to access yFiles functionality that is part of an application via publicly available class or method names.

### Important

All *private* and *package private* yFiles classes, methods, and fields are already name-obfuscated as a factory default, since they cannot be used for software development anyway. The remaining *public* and *protected* parts of the yFiles API must be obfuscated before a yFiles-based product can be released.

## Name Obfuscation and its Benefits

Name obfuscation works by replacing names in Java byte code, e.g., package names, class, method, and field names by nonsensical new names. The replacement is done in a consistent way, so that the byte code still works as before.

[Example A.1, “Method name obfuscation” \[319\]](#) conceptually shows the effects of name obfuscation for method names and method signatures. Several methods with distinct signature are mapped to a single new name.

### Example A.1. Method name obfuscation

```
// Original method names/signatures.  
public void myMethod(MyCustomType type, String name, boolean enable){/* Code */}  
public void anotherMethod(MyCustomType type){/* Code */}  
  
// Method names/signatures after name obfuscation.  
public void a(f b, String c, boolean d){/* Code */} // Formerly 'myMethod'.  
public void a(f b){/* Code */} // Formerly 'anotherMethod'.
```

By replacing different method names with a single new name, decompiled Java byte code is made rather incomprehensible to a human reader, making ad-hoc reverse-engineering attempts difficult. Note that, as a side effect of the obfuscation process, the size of any Jars that bundle Java byte code files is significantly reduced, too.

# Name Obfuscation Using yGuard

yGuard is a Java byte code obfuscator that can be used for effective name obfuscation of Java byte code. It is bundled with every yFiles distribution (see the [yGuard README file](#)), and is also [freely available](#).

Since yGuard fulfills the obfuscation requirements of the yFiles license terms, it is a perfectly suited tool to protect the yFiles part of any yFiles-based application. (Of course, it is also the perfect tool to protect the entire application.)

yGuard is available as an Apache Ant task for easy integration into any Ant-based build process. [Example A.2, “Using the yGuard Ant task” \[320\]](#) shows an Ant build file’s target that obfuscates the yFiles part of an application.

## Note

The example uses yGuard 2.0 syntax.

### Example A.2. Using the yGuard Ant task

```
<!-- Obfuscates the yFiles part of an application (i.e., y.jar) and adjusts -->
<!-- the non-yFiles part given by the application's Jar file accordingly. -->
<target name="obfuscate" depends="jar">
    <!-- yGuard Ant task. -->
    <taskdef name="yguard"
        classname="com.yworks.yguard.YGuardTask"
        classpath="${yGuardJar}"/>
    <!-- Integrated obfuscation and name adjustment... -->
    <yguard>
        <!-- Obfuscate the yFiles Jar. -->
        <inoutpair in="${yJar}" out="${yJarObf}" />
        <!-- While obfuscating, adjust the names of yFiles features in the -->
        <!-- application's Jar file. -->
        <inoutpair in="${myAppJar}" out="${myAppJarObf}" />

        <!-- ...using the yGuard 'rename' task. -->
        <rename logfile="${obfuscationLog}" replaceClassNameStrings="true">

            <property name="obfuscation-prefix" value="myprefix"/>
            <keep>
                <class classes="private" methods="private" fields="private">
                    <patternset>
                        <include name="com.mycompany.myApp.**"/>
                    </patternset>
                </class>
            </keep>

            <!-- Adjust all occurrences of now obfuscated names in plain-text -->
            <!-- files, too. -->
            <!-- For example, the names of property resource bundle files for -->
            <!-- yFiles classes must be adjusted to reflect the yFiles class's -->
            <!-- obfuscated name. -->
            <adjust replaceName="true">
                <include name="y/**/*.properties"/>
            </adjust>

        </rename>
    </yguard>
</target>
```

For detailed explanations of yGuard’s Ant syntax, please see the [yGuard manual](#).

## Adjusting Names

Any code that makes use of yFiles features, or more specifically uses names of yFiles features, needs to be adjusted to their new names. This adjustment is automatically performed for all Jar files given to the yGuard obfuscation machinery using the `<inoutpair>` element.

Using yGuard’s `<adjust>` element (inside the `<obfuscate>` element), any additional plain-text files that are not processed by the obfuscation itself, e.g., resource files, can be adjusted in several ways, too. For example, a file’s path, its name, and also its content are subject to possible modifications.

## Checking Obfuscation Success

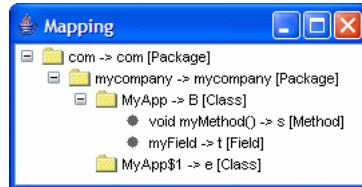
During the obfuscation process, yGuard optionally generates a log file that contains all original names, their new obfuscated names, and also additional statistical information in XML-based syntax. The log file is an important document, since it shows the mapping of original to obfuscated names. To display the information nicely, yGuard offers a convenient reverse name look-up feature.

[Example A.3, “yGuard's command line syntax” \[321\]](#) shows the command line syntax to invoke yGuard's reverse name look-up. Given a generated log file (or its GZip compressed variant), both the original class, method, and field names and their obfuscated counterparts are conveniently presented in a tree view, as depicted in [Figure A.1, “yGuard's reverse name look-up view” \[321\]](#)

### Example A.3. yGuard's command line syntax

```
java -jar yguard.jar [yGuard_obfuscation_log_file.xml[.gz]]
```

**Figure A.1. yGuard's reverse name look-up view**



To ensure that the yFiles API has been properly obfuscated, all *public* and *protected* class, method, and field names must be present in the tree view presented by yGuard's reverse name look-up feature. In other words, all names of such features must have been replaced by new names that are distinctly different from their originals.

## Tutorial Demo Code

For an example on how to obfuscate the yFiles part of a yFiles-based application using yGuard, see the obfuscation demo's [Ant build file](#) (included with the yFiles Complete distribution). The build file also shows how to adjust the occurrences of obfuscated names of any yFiles features in properties files.



---

# Glossary

This glossary naturally has a strong emphasis on graph structure related terms and concepts. The listed entries are mainly terms used by the yFiles graph visualization library.

## A

Acyclic	Attribute of a graph. A graph is called acyclic when there is no path of edges that starts and ends at the same node.
Adjacent	Edges connecting to a node are called adjacent edges. Also, nodes that are direct neighbors of each other are called adjacent.
Arc	Another name for an edge. See Also Edge.

## B

BCC	See Biconnected Component.
Bend	A bend is part of the visual representation of an edge. It determines the coordinates where two segments of the edge connect. See Also Edge.
Biconnected Component	A component where an arbitrary node can be removed and the remaining nodes are still connected. Informally, biconnected means that there are at least two different paths to reach a node from another one. See Also Connectivity, Component.
Bipartition	A graph where the node set can be partitioned into two disjoint sets such that there is no edge between nodes from the same set.

## C

Complexity	The minimum amount of resources (like time or memory) it needs to solve a problem/execute an algorithm/accomplish a task. See Also O.
Component	Part of a graph that fulfills certain connectivity criteria. Components are, e.g., connected, strongly connected or biconnected. In a connected component (or component, for short) every node is reachable from every other node. See Also Biconnected Component, Connectivity.
Connected	Attribute of a graph or a component. A graph is called connected if it consists of exactly one component. See Also Component.
Connectivity	Defines the reachability of nodes in a graph. A node is said to be reachable from another one, when there is a path of edges between them. The edges in the path are interpreted undirected.
Cycle	

---

A path of edges that starts and ends at the same node.

## D

Directed	Attribute of an edge and a graph. An edge is called directed if it has a distinct source node and a distinct target node, i.e., if it has direction. A graph is called directed if all edges from the edge set are directed.  Attribute of a tree. A tree is called directed if all its edges are uniformly pointing towards the leaf nodes. See Also Graph, Edge, Tree.
Directed Tree	See Directed.

## E

Edge	The representative for a relation between two nodes. An edge is usually depicted as a line connecting its end nodes. To indicate the destination of a directed edge an arrowhead is drawn at the target end of the line. See Also Graph.
Edge Segment	An edge segment is part of the visual representation of an edge. It connects either a port with a bend, or two bends of the edge. See Also Bend, Edge, Port.

## G

Graph	Mathematical object defined as consisting of two sets, a node set and an edge set. See Also Node, Edge.
-------	--

## I

Incoming	Attribute of a directed edge. A directed edge is an incoming edge at its target node. See Also Directed, Edge, Target.
Inner Node	A node in a tree that has more than one connecting edge, but is not the root node. In particular, in a directed tree an inner node has an arbitrary number of outgoing edges and exactly one incoming edge. See Also Root Node, Directed, Tree.

## L

Leaf Node	A node in a tree that has exactly one connecting edge. In particular, in a directed tree a leaf node has no outgoing edges and exactly one incoming edge. See Also Directed, Tree.
Link	Another name for an edge. See Also Edge.

## N

---

<b>Node</b>	Name for the representative of an entity. See Also Graph.
-------------	--

## O

<b>O</b>	<b>Big-Oh notation.</b> A theoretical measure for the execution of an algorithm. Usually denotes the time (or memory) needed relative to the problem size, i.e., some number of items.  The most frequently encountered complexity classes are (where n is the number of items):
	<ul style="list-style-type: none"> <li>• <b>O(1), constant time.</b> Does not depend on the size of the problem, i.e., an operation with O(1) always takes constant time.</li> <li>• <b>O(n), linear time.</b> The execution time grows proportional to the size of the problem, i.e., if the problem size n doubles, then an operation with O(n) will roughly take twice as long.</li> <li>• <b>O(n*n), square time.</b> The execution time grows proportional to the square of the size of the problem, i.e., if the problem size n doubles, then an operation with O(n*n) will roughly take four times as long.</li> </ul>

<b>Orthogonal</b>	Popular layout and drawing style. Nodes are usually represented by rectangles, edges are made up of alternating vertical and horizontal segments.
<b>Outgoing</b>	Attribute of a directed edge. A directed edge is an outgoing edge at its source node. See Also Directed, Edge, Source.

## P

<b>Parallel Edges</b>	Multiple edges between the same two nodes are called parallel edges. See Also Edge.
<b>Path</b>	A sequence of edges connecting two nodes.
<b>Planar</b>	Attribute of a graph. A graph is called planar, if it can be drawn in the plane such that there are no edge crossings.
<b>Port</b>	Explicitly expressed coordinates where the visual representation of an edge connects to a node.

## R

<b>Reachability</b>	See Connectivity.
<b>Reflexive Edge</b>	Another name for a self-loop. See Also Self-Loop.
<b>Relation</b>	Another name for an edge. See Also Edge.
<b>Root Node</b>	A distinguished node in a tree that has more than one connecting edge. In particular, in a directed tree the root node

---

has an arbitrary number of outgoing edges and no incoming edge.  
See Also Tree.

Rooted Tree

A directed tree with designated root node.  
See Also Tree.

## S

Self-Loop

Directed edge where source and target node are identical.  
See Also Directed, Edge.

Subgraph

Distinguishable part of a graph.  
See Also Graph.

Source

Name for the origination node of a directed edge.  
See Also Directed, Edge.

## T

Target

Name for the destination node of a directed edge.  
See Also Directed, Edge.

Tree

An acyclic connected graph where the node set can be divided into one root node, and an arbitrary number of inner nodes and leaf nodes. The edges of a tree are usually interpreted undirected.

Traditionally, a tree is depicted with its root at topmost position and its leaf nodes at the bottom (i.e., it is "growing" downwards).

See Also Root Node, Inner Node, Leaf Node, Directed Tree.

## U

Undirected

Attribute of an edge and a graph. An edge is called undirected if it does not have direction, i.e., there is no distinct source node or target node. A graph is called undirected if all its edges are undirected.  
See Also Directed, Edge, Graph.

## V

Vertex

Another name for a node.  
See Also Node.

## XYZ

yWorks

Reputable maker of quality software for state-of-the-art automatic layout and visualization of graphs.

---

# Index

## A

anchor node (see folder node)  
animation  
    general framework, [226]  
    layout morphing (see application logic)  
    real-time customization callback mechanism, [229]  
application logic  
    clipboard, [232]  
    layout morphing, [232]  
    settings dialogs (see GUI utility functionality)  
    undo/redo support, [231]

## B

bend  
    visual representation, [186]  
BFS (see breadth-first search)  
breadth-first search (BFS), [37]  
bridges (see rendering)  
bus-style edge routing (see constraints)  
byte code protection requirements (see obfuscation)

## C

centrality measures, [42]  
child mode (see view modes)  
clustering (see nesting)  
complexity  
    of common graph structure tasks, [22]  
constraints  
    edge ports, [59]  
    edge/port groups, [63]  
    port candidate sets for nodes, [61]  
    port candidates with edges, [61]  
context-sensitive menus (see user interaction)  
copying  
    a Graph instance, [23]  
    a Graph2D instance, [163]  
    a LayoutGraph implementation, [50]  
    generic mechanism for copying a graph structure, [23]  
    graph hierarchies, [163]  
cursor, [26]  
    concept, [14]

## D

data accessors  
    concept, [15]  
    customized, [34]  
    default map implementations, [33]  
    definition, [32]  
    index-based map implementations, [35]  
data provider, [32], [32]  
    registration using well-known look-up keys, [55]  
default realizer mechanism, [162], [164]  
depth-first search (DFS), [38]  
DFS (see depth-first search)

## E

edge  
    class Edge  
        creating an instance, [17], [18]  
        provided functionality  
        (see also graph elements)  
    definition, [13]  
    visual representation (see edge realizer)  
edge groups (see constraints)  
edge realizer, [164]  
    customization, [189]  
    general features, [165]  
    labels (see labels)  
    predefined classes, [170]  
    related classes, [184]  
EMF  
    file format, [281], [315]  
Encapsulated PostScript (see EPS)  
Enhanced Windows Metafile (see EMF)  
EPS  
    file format, [281], [315]

**F**

file formats  
    extension packages, [281], [295], [311], [315]  
    graph exchange  
        GML, [255], [259]  
        GraphML, [281], [295]  
        TGF, [255], [272]  
        XGML, [255], [271]  
        YGF, [255], [256]  
    graph export  
        EmF, [281]  
        EMF, [315]  
        EPS, [281], [315]  
        GIF, [274], [277]  
        JPG, [274], [277]  
        PDF, [281], [315]  
        PNG, [274], [278]  
        SVG, [281], [311]  
        SWF, [281], [315]  
    file I/O, [251]  
fine-grained rendering order  
    of graph elements (see view implementations)  
folder node, [235]  
    visual representation, [243]

**G**

GIF  
    file format, [274], [277]  
GML  
    file format, [255], [259]  
    syntax, [260]  
graph  
    class Graph  
        creating an instance, [17]  
        provided functionality, [18]  
    class Graph2D, [162]  
        default realizer mechanism, [162]  
        complete class hierarchy, [162]  
        copying a Graph instance, [23]  
        copying a Graph2D instance, [163]  
        copying a LayoutGraph implementation, [50]  
        definition, [13]

- 
- displaying and editing, [161]  
exchange file formats  
  reading from and writing to file, [255]  
export file formats  
  writing to file, [274]  
file I/O, [251]  
generic mechanism for copying a graph structure, [23]  
hierarchically organized, [56]  
layout information, [48]  
positional and dimensional information, [48]  
printing, [223]  
undirected interpretation, [13]  
visual representation, [162]
- graph elements  
  binding data to, [15], [32]  
  creating, [17]  
  fine-grained rendering order (see view implementations)  
  hiding, [19], [23]  
  removing, [19]  
  rendering order (see view implementations)  
  subclassing, [15]  
  visual representation (see realizer)
- graph hierarchies  
  class HierarchyManager, [239]  
  concepts, [235]  
  copying, [163]  
  preparing for automatic layout, [250]  
  proxy elements, [235]
- Graph Modeling Language (see GML, XGML)
- GraphML  
  attributes, [295]  
    look-up mechanism, [295]  
    support for custom data of simple type, [299]  
    support for custom structured data, [300]  
  file format, [281], [295]  
  XSLT support, [309]
- yFiles-specific features  
  edge graphics, [296]  
  node graphics, [296]  
  post-processors, [306]
- grid, [200]
- group node, [235]  
  layout policies, [56]  
  visual representation, [243]
- grouping, [56], [235]  
  compared to nesting, [237]  
  creating grouped nodes, [239]
- GUI utility functionality  
  settings dialogs, [283]  
    enabling and disabling of dialog items, [288]  
    internationalization and localization, [290]  
    synchronization, [287], [294]
- H**  
hierarchically organized graphs (see graph hierarchies)
- I**  
I/O (see file I/O)  
image export, [274]  
  creating an HTML image map, [279]
- tiling, [278]  
incoming, [13], [21]  
inter-edge, [236]  
internationalization  
  settings dialogs support for (see GUI utility functionality)  
iteration  
  forward and backward, [27]  
  low-level, [31]  
  non-cursor, [28]  
  using cursors, [26]  
    removing elements, [28]
- J**
- JPG  
  file format, [274], [277]
- K**  
keyboard actions (see user interaction)
- L**  
labels, [178]  
  automatic placement, [156]  
  different scenarios, [156]  
  general labeling, [158]  
  integrated labeling, [160]  
  label model, [157], [182], [183]  
    candidate positions, [157]  
    model parameter, [157]  
  multi-line editor, [209]
- layout  
  animation (see layout morphing)  
  arranging components (see layout stage)  
  default compound process (see layout stage)  
  excluding components (see layout stage)  
  invocation, [51]  
  invoking different layout algorithms for components (see layout stage)  
  morphing (see application logic)  
  providing supplemental data to an algorithm, [55]  
  unbuffered vs. buffered, [53]  
  wrapping graph layouts (see layout stage)
- layout stage  
  buffered layout, [54]  
  component arrangement, [67]  
    excluding components from layout calculation, [68]  
    invoking different layout algorithms for components, [68]  
    styles, [67]
- concept, [52]  
  default compound layout process, [53]  
  general-purpose, [65]  
  graph layout wrapper, [70]
- listener paradigm  
  notification of graphical property changes, [221]  
  notification of structural changes, [25]
- localization  
  of settings dialogs (see GUI utility functionality)
- M**  
map, [32], [32]  
Model-Delegate pattern

- 
- presence in package y.option, [283]  
 Model-View-Controller paradigm  
     controller classes, [208]  
     model classes, [162]  
     presence in package y.view, [161]  
     view classes, [197]  
 mouse wheel support (see view implementations)
- N**
- nesting, [235]  
     compared to grouping, [237]  
     creating nested graphs, [239]
- node  
     class Node  
         creating an instance, [17]  
         provided functionality  
         (see also graph elements)  
     definition, [13]  
     visual representation (see node realizer)
- node realizer, [164]  
     customization, [189]  
     general features, [165]  
     in graph hierarchy context, [243]  
     labels (see labels)  
     predefined classes, [166]  
     with Swing look and feel, [195], [210]
- O**
- obfuscation, [319]  
 outgoing, [13], [21]
- P**
- parallel edges, [13]  
 PDF  
     file format, [281], [315]  
 PNG  
     file format, [274], [278]  
 popup menus (see user interaction)  
 port candidates  
     (see also constraints)  
     for nodes, [61]  
     with edges, [61]  
 port constraints (see constraints)  
 Portable Document Format (see PDF)
- R**
- realizer  
     concept, [164]  
     configurations, [191]  
     customization, [189]  
     label support, [178]  
     replication behavior, [189]  
 reflexive edge (see self-loop)  
 rendering  
     support for bridges in edge paths, [205]  
 rendering order  
     inside view (see view implementations)  
     of graph elements (see view implementations)
- S**
- Scalable Vector Graphics (see SVG)  
 selection box
- customizing look (see view implementations)  
 self-loop, [13], [21], [22]  
 Shockwave Flash (see SWF)  
 shortest paths, [40]  
 spanning trees, [43]  
 SVG  
     file format, [281], [311]  
 SWF  
     file format, [281], [315]  
 Swing look and feel  
     editor support, [215]  
     with node realizers (see node realizer)
- T**
- TGF  
     file format, [255], [272]  
 tool tips  
     HTML-based, [218]  
 transitive closure and reduction, [44]  
 trees, [45]  
 Trivial Graph Format (see TGF)
- U**
- undo/redo support (see application logic)  
 user interaction, [208]  
     class CellEditorMode, [215]  
     class EditMode, [212]  
         customization, [218]  
     class HitInfo, [217]  
     class MagnifierViewMode, [216]  
     class NavigationMode, [215]  
     class ViewMode, [211]  
     double precision mouse events, [218]  
     events and listeners, [221]  
     in graph hierarchy context, [247]  
         class HierarchyEditMode, [247]  
         events and listeners, [249]  
     keyboard actions, [208]  
     mouse actions, [208], [211]  
     mouse input editor, [218]  
     popup menus, [212]  
     settings dialogs (see GUI utility functionality)
- V**
- vector graphics export (see EMF) (see EPS) (see PDF) (see SVG, PDF, SWF, EMF, EPS) (see SWF)  
 view implementations, [197], [245]  
     class Graph2DView, [197]  
         coordinate systems, [198]  
         fine-grained rendering order, [203]  
         grid support, [200]  
         less-detail rendering, [202]  
         mouse wheel support, [207]  
         rendering order, [201], [203]  
         selection box, [200]  
         zooming, [198]  
     class HierarchyJTree, [245]  
         drag-and-drop within tree, [245]  
     class Overview, [204]  
 view modes, [211]  
     chaining, [211], [213]  
     child mode, [211], [213]  
     property change notification, [211]

---

**X**

XGML

file format, [255], [271]

XSLT support (see GraphML)

**Y**

Y Graph Format (see YGF)

YGF

file format, [255], [256]

**Z**

zooming (see view implementations)