IBM ILOG JViews Gantt V8.6

Developing with the JViews Gantt SDK

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

Copyright

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

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

Trademarks

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

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

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

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

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

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

Notices

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

Table of contents

Introducing the main classes....9

Class overview....10

Data model classes....13

The interaction between abstract and concrete elements....14

Using the abstract implementation to create a custom data model....16

Concrete data model implementations....17

Binding the Gantt chart components to the data model....19

Time and duration....20

Connecting to data....23

Connecting to data in-memory....25

When to use data in-memory....26

Activities and resources....27

Populating the data model....28

Manipulating activities and resources....29

Activity and resource factories....30

Constraints....32

Reservations....34

Connecting to XML data....37

The SDXL format....38

The schedule data exchange language....39

How to write an IlvGanttModel to an SDXL file using serialization....43

© Copyright IBM Corp. 1987, 2009

How to read an IlvGanttModel from an SDXL file using serialization....46
Handling exceptions while reading SDXL files....48
Connecting to Swing TableModel instances....49
Overview....51
Data required by IlvTableGanttModel....52
Converting TableModel data to IlvTableGantt Model data....54
Configuring the IlvTableGanttModel object correctly....56
Read/write support....59
Dynamic behavior....60
Reading data from CSV files....61
Complex mappings....62
Connecting to data through JDBC....65
Overview....66
Writing queries to populate the data model....67
The harbor example....69
Passing the data to the Gantt data model....70
Implementing custom data models....72
Styling....73
Styling examples....75
Using CSS syntax in the style sheet....77
Overview....78
The origins of CSS....79
The CSS syntax....80
Applying CSS to Java objects....83
Overview....85
The CSS engine....86
The CSS data model....87
CSS recursion....91
Constructs....92
Expressions....94
Custom functions....95
Registering custom functions....97
Divergences from CSS2....98
Using style sheets....101
Applying styles....102
Disabling styling....104
The Gantt and Schedule CSS examples....105
Running the examples....106
Scheduling data....107
Customizing a Gantt chart using a simple style sheet....108

The resource data CSS example....111
Running the Example....112
Scheduling data....113
Customizing a Resource Data style sheet....114
Two kinds of rules....117
Styling Gantt and Schedule chart components....118
Styling Gantt chart and Schedule chart data....123
Overview....124
Styling activities....125
Activity model objects....126
Activity renderer target objects....128
Activity ID selectors....131
IlvGeneralActivity properties....132
IlvGeneralActivity CSS classes....133
Activity CSS pseudoclasses....134
The formatDate and formatDuration functions....135
Styling constraints....137
Constraint model objects....139
Constraint graphic target objects....141
Constraint ID selector....142
IlvGeneralConstraint properties....143
IlvGeneralConstraint CSS classes....144
Constraint CSS pseudoclasses....145
The activityProperty function....146
Styling Resource Data chart components....147
Overview....148
Styling the Chart Area component....151
Styling the Chart Legend....152
Styling the chart renderer....154
Styling the chart scales....155
Styling the chart grids....157
Styling the Resource Data chart data....159
Overview....160
Selector patterns....161
Attributes of model objects....162
CSS classes....164
Properties....165
Properties for data series....168
Gantt charts....169
The architecture of the Gantt charts....171
The Gantt beans....173

Overview....174
Structure....175
Properties....176
Basic steps for using the Gantt chart and Schedule chart beans....177
Running the samples....179
Gantt chart....180
Running the sample as an application....182
Schedule chart....185
Deploying a Gantt application....186
Using Gantt and Schedule charts....187
Chart visual properties....188
Expanding or collapsing and hiding or showing rows....190
Controlling row structure and visibility....191
Scrolling in the Gantt sheet....192
Using the Gantt sheet....195
Gantt sheet architecture....196
Rendering the data in the Gantt sheet....198
Time indicators....200
Activity layouts....205
Using the time scale....207
Changing the rows of a time scale....208
Visibility policy....210
Controlling row visibility....211
Nonlinear time scale....213
Customizing Gantt charts....215
Customization examples....216
Running the Custom Gantt example....217
Customization overview....218
Customizing the Gantt data model....219
Customizing activity rendering....221
The Activity Rendering API....222
Simple activity renderers....223
Combining activity renderers....224
Rendering Activities with Multiple Dates....229
Using Composite Graphics....232
Installing Custom Activity Renderers....237
Customizing table columns....239
Running the example....240
Tree column icons....241
The PriorityColumn class....242
Adding the column to the table....247

Dynamic columns......248
Interacting with the Gantt charts......253
Class associations for interactors......254
Selecting activities and constraints......255
Moving activity and reservation graphics......256
Duplicating reservation graphics......257
Resizing activity and reservation graphics......258
Interacting with the Gantt sheet using the mouse......259
Creating activities and reservations......260
Creating constraints......261
Popup menus......262
Resource Data charts......265
The architecture of the Resource Data chart......266
The Resource Data chart bean......267
Basic architecture......268
Basic steps in using the Resource Data chart bean - details......270
Comparing the Resource Data chart with IBM® ILOG® JViews Charts......271
Computing and displaying resource data......272
Synchronizing Schedule charts and Resource Data charts......273
Overview......274
Selecting resources for display......275
Computing resource data......277
Rendering resource data......278
The x-axis......279
Calendar view components......281
Calendar view beans......282
Running the Calendar View sample......284
Basic architecture......285
Overview......286
Calendar View models......288
Calendar View renderers......289
Leaf activity and holiday renderers......291
Milestone renderers......294
Deploying as an applet......296
Using JViews products in Eclipse RCP applications......297
Printing......305
Printing Gantt and Schedule charts......307

Overview...308
Introduction...309
The GanttPrintExample demo...310
Printing Framework API...312
How it works...317
Printing a Resource Data chart...319
Introduction...320
The Printing Resource Data chart Example...321
Printing framework API...322
Critical path analysis...325
Critical path analysis overview...326
Example...328
Handling errors...329
Loading data on demand...331
Vertical load-on-demand...333
Overview...334
Running the Database Gantt example...335
Understanding the Database Gantt example...336
Horizontal load-on-demand...339
Overview...340
Running the Database Schedule example...341
Understanding the Database Schedule example...342
Document type definition for SDXL...345
Index...347

Introducing the main classes

Explains how the main classes of JViews Gantt are organized and how they relate to the main functionality in the product.

In this section

Class overview
Describes the high-level chart and calendar components and the main classes of the Gantt API that implement them.

Data model classes
Describes the abstract data model, and the concrete implementations that make up the scheduling model.

Binding the Gantt chart components to the data model
Explains how to bind model view objects to a data model.

Time and duration
Describes how to instantiate and use the classes that represent date, duration and time interval in JViews Gantt.

© Copyright IBM Corp. 1987, 2009

Class overview

JViews Gantt provides a library of classes for displaying an abstract data model of scheduling information as a Gantt chart, a Schedule chart, a Resource Data chart, and monthly or daily calendar views.

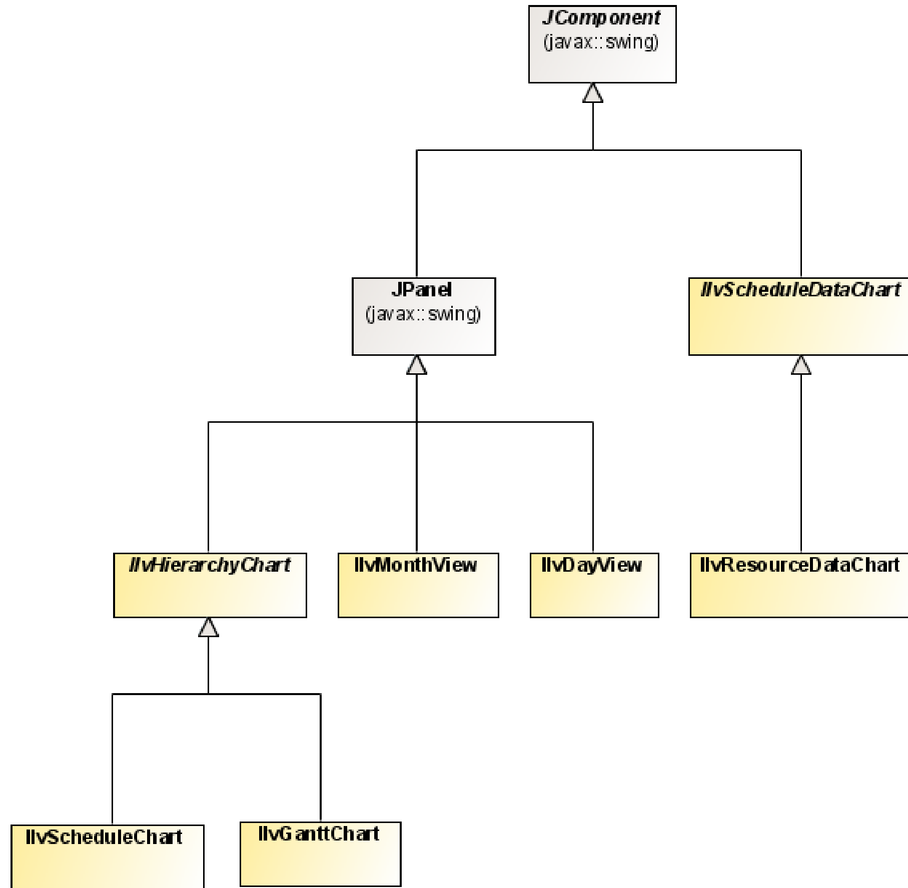
JViews Gantt features three high-level chart components, called the Gantt chart, Schedule chart, and Resource Data chart. The Gantt and Schedule charts are implemented by the `IlvGanttChart` and `IlvScheduleChart` classes, respectively, both subclasses of `IlvHierarchyChart`. The Resource Data chart is implemented by the `IlvResourceDataChart` class, which is a subclass of `IlvScheduleDataChart`.

IBM® ILOG® JViews Gantt also features the following high-level calendar view components:

- Monthly Calendar View
- Daily Calendar View.

These views are implemented by the `IlvMonthView` and `IlvDayView` classes, respectively. The chart and calendar view components encapsulate the Gantt library API and provide a high-level interface to its capabilities. Together with the `IlvGanttModel` interface, the three chart and two calendar view components make up the six main classes of the Gantt API.

The following figure shows the classes of JViews Gantt.

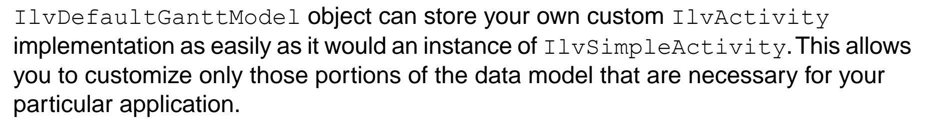


The scheduling data displayed by the chart and calendar view components is defined by the abstract `IlvGanttModel` interface. This interface defines the overall data model and acts as an intelligent container for the other four data model entities.

These data models are:

- Activities, defined by the `IlvActivity` interface.
- Resources are defined by the `IlvResource` interface.
- Activity-to-activity constraints are defined by the `IlvConstraint` interface.
- Assignment of a resource to an activity is defined by the `IlvReservation` interface.

Note: All the data model interfaces and provided implementations are independent of the exact implementation of the other portions of the data model. For example, an



IlvDefaultGanttModel object can store your own custom IlvActivity implementation as easily as it would an instance of IlvSimpleActivity. This allows you to customize only those portions of the data model that are necessary for your particular application.

Data model classes

Describes the abstract data model, and the concrete implementations that make up the scheduling model.

In this section

The interaction between abstract and concrete elements
Explains the interaction between the main interfaces in the data model and the implementation classes supplied for them.

Using the abstract implementation to create a custom data model
Explains where to find information about how to create a custom data model using the abstract implementation as a starting point.

Concrete data model implementations
Describes the data model implementations provided by JViews Gantt.

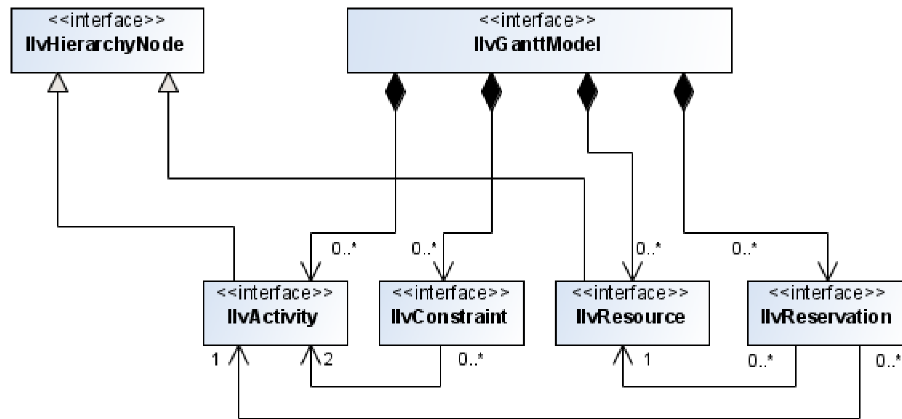
The interaction between abstract and concrete elements

The data model is completely abstract and is defined by the `IlvGanttModel` interface. This interface acts as an intelligent container for four other abstract interfaces that represent the scheduling data itself: `IlvActivity`, `IlvConstraint`, `IlvResource`, and `IlvReservation`. These interfaces are included in the `ilog.views.gantt`.

The following table gives a brief description of each interface.

Data Model Interface	Description
`IlvGanttModel`	Defines the overall JViews Gantt data model and is a container for the other four entities.
`IlvActivity`	Represents an activity or task that must be completed in the schedule.
`IlvResource`	Represents a resource that can be allocated to an activity to enable its completion.
`IlvConstraint`	Represents an activity-to-activity scheduling constraint.
`IlvReservation`	Represents the allocation of a resource to an activity.

The following figure shows the associations between the five interfaces that compose the JViews Gantt data model.



Several levels of implementation are available for each of these abstract interfaces.

- *Using the abstract implementation to create a custom data model*
- *Concrete data model implementations*

The following tables summarize implementation classes supplied for the data model interfaces described in the previous table.

Abstract Implementation	Simple Memory-Based Implementation	Default Memory-Based Implementation
IlvAbstractGanttModel	IlvDefaultGanttModel	IlvDefaultGanttModel
IlvAbstractActivity	IlvSimpleActivity	IlvGeneralActivity
IlvAbstractResource	IlvSimpleResource	IlvGeneralResource
IlvAbstractConstraint	IlvSimpleConstraint	IlvGeneralConstraint
IlvAbstractReservation	IlvSimpleReservation	IlvGeneralReservation

Abstract Implementation	Swing TableModel Implementation	JDBC™ Implementation
IlvAbstractGanttModel	IlvTableGanttModel	IlvJDBCGanttModel
IlvAbstractActivity	IlvTableActivity	IlvTableActivity
IlvAbstractResource	IlvTableResource	IlvTableResource
IlvAbstractConstraint	IlvTableConstraint	IlvTableConstraint
IlvAbstractReservation	IlvTableReservation	IlvTableReservation

In general, there are no hard-coded dependencies between the data model implementation classes. This means that you can choose to use as much of the provided data models as you need while subclassing just the portion that you need to customize for your application.

The notable exceptions are:

- The class `IlvTableGanttModel` that you can use to connect to Swing TableModel instances.

  See *Connecting to Swing TableModel instances* for details.

- The class `IlvJDBCGanttModel` that you can use to connect to a database.

  See *Connecting to data through JDBC* for details.

These data model implementations require that their data entities be instances of `IlvTableActivity`, `IlvTableResource`, `IlvTableConstraint`, or `IlvTableReservation`. The `IlvTableGanttModel` and `IlvJDBCGanttModel` classes automatically create these data instances for you from the contents of the Swing or database tables. Therefore, you do not need to be concerned with explicitly populating an `IlvTableGanttModel` or `IlvJDBCGanttModel` object.

Using the abstract implementation to create a custom data model

An abstract implementation is provided as a starting point for your own custom data model designs. These classes provide the basic event notification framework, but no property or data storage.

You can create your own custom data model in its entirety, but you are recommended to use the abstract classes as a starting point. How to extend the abstract classes for this purpose is an advanced topic not covered in this documentation.

This topic is demonstrated in the database examples available in:

- **`<installdir>/jviews-gantt86/samples/databaseGantt`**
- **`<installdir>/jviews-gantt86/samples/databaseSchedule`**

The abstract implementations are included in the `ilog.views.gantt.model`.

Concrete data model implementations

Describes the concrete data model implementations provided by JViews Gantt. How to extend them is an advanced topic not covered in this section. See *Customizing Gantt charts* for a customization example.

Simple data model implementation

This concrete implementation is completely memory-based and provides the most basic implementation of the Gantt data model. Only the required properties of each data model entity are supported. This implementation is used as the basis for the Default Memory-Based data model described in *Default data model implementation*. It can also be used as a more complete foundation for your own custom data model extensions. The simple data model implementation is included in the `ilog.views.gantt.model` .

The following table shows the corresponding data model interfaces and implementation classes.

Data Model Interface	Simple Memory-Based Implementation
`IlvGanttModel`	`IlvDefaultGanttModel`
`IlvActivity`	`IlvSimpleActivity`
`IlvResource`	`IlvSimpleResource`
`IlvConstraint`	`IlvSimpleConstraint`
`IlvReservation`	`IlvSimpleReservation`

Default data model implementation

The default data-model implementation extends the Simple Data Model implementation and is also completely memory-based. This implementation inherits the required properties of each data model entity and adds support for user-defined properties. It is used throughout the examples, except for the Database example.

The following table shows the corresponding data model interfaces and implementation classes.

Data Model Interface	Default Memory-Based Implementation
`IlvGanttModel`	`IlvDefaultGanttModel`
`IlvActivity`	`IlvGeneralActivity`
`IlvResource`	`IlvGeneralResource`
`IlvConstraint`	`IlvGeneralConstraint`
`IlvReservation`	`IlvGeneralReservation`

See the `ilog.views.gantt.model.general`.

Connection to a JDBC database

The `IlvJDBCGanttModel` implementation of the `IlvGanttModel` interface connects to a database through JDBC™ to get the definition of the activities, resources, constraints, and reservations with simple mapping configuration. It is documented in *Connecting to data through JDBC*. See also the `ilog.views.gantt.model.jdbc`.

Connection to a Swing TableModel

This data model implementation enables you to create your custom data model from Swing `TableModel` instances. It is documented in *Connecting to Swing TableModel instances*. See also the `ilog.views.gantt.model.table`.

The following table shows the corresponding data model interfaces and implementation classes.

Data Model Interface	Swing TableModel Implementation
`IlvGanttModel`	`IlvTableGanttModel`
`IlvActivity`	`IlvTableActivity`
`IlvResource`	`IlvTableResource`
`IlvConstraint`	`IlvTableConstraint`
`IlvReservation`	`IlvTableReservation`

Binding the Gantt chart components to the data model

As explained in Getting to know the Designer in *Using the Designer*, the data model is designed with complete model-view separation.

Use the following method to bind an `IlvGanttChart`, `IlvScheduleChart`, `IlvResourceDataChart`, `IlvMonthView` or `IlvDayView` object to a data model, as illustrated in *Running the samples*:

```
void setGanttModel(IlvGanttModel ganttModel)
```

You can obtain the current data model of the chart by using the method:

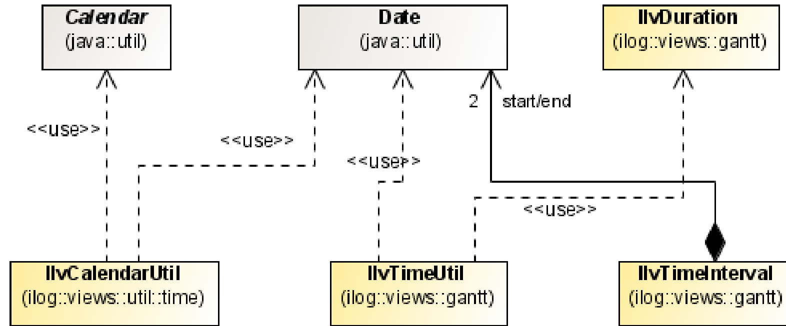
```
IlvGanttModel getGanttModel()
```

Both methods are members of the `IlvHierarchyChart` class (the common superclass of `IlvGanttChart` and `IlvScheduleChart`), and `IlvScheduleDataChart` (the superclass of `IlvResourceDataChart`), `IlvMonthView` and `IlvDayView`.

Time and duration

Throughout the JViews Gantt product, time is represented by the standard `java.util.Date` class, duration by `IlvDuration`, and time intervals by `IlvTimeInterval`.

The following figure shows the classes for Manipulating Time and Duration.



Date

The standard `java.util.Date` class is capable of representing an instant in time with millisecond precision, starting from January 1, 1970. As of JDK 1.1, all methods that can modify a `Date` instance have been deprecated. JViews Gantt considers dates to be immutable and a separate `java.util.Calendar` object must be used to perform date arithmetic. Some useful arithmetic methods, such as `add` and `subtract`, are bundled into the utility classes `IlvCalendarUtil` and `IlvTimeUtil`. These methods always return a new `Date` instance instead of modifying the original object.

You can create a `Date` instance using one of the following constructors:

- Date()
- Date(long millis)

The following methods can be used to compare or perform arithmetic on dates:

- `Date IlvTimeUtil.add(Date date, IlvDuration delta)`
- `Date IlvTimeUtil.subtract(Date date, IlvDuration delta)`
- `IlvDuration IlvTimeUtil.subtract(Date date1, Date date2)`
- `Date IlvCalendarUtil.min(Date a, Date b)`
- `Date IlvCalendarUtil.max(Date a, Date b)`

The following methods can be used to compare or perform arithmetic on dates:

The following example shows how to create a `Date` that represents 8:00 am on April 4, 2001:

```
Calendar calendar = Calendar.getInstance();
calendar.clear();
calendar.set(2001, Calendar.APRIL, 4, 8, 0);
Date date = calendar.getTime();
```

Note: An instance of the `java.util.Calendar` class initializes all its time fields to the current time. You must explicitly clear those `Calendar` fields that you want set to zero. In the previous example, calling the `clear` method ensures that the second and millisecond fields of the `Calendar` object are set to zero.

IlvDuration

The `IlvDuration` class creates duration objects that represent a length of time with millisecond precision. Like `Date`, `IlvDuration` is an immutable class. Therefore, to create a different duration, you must create a new `IlvDuration` object. The class `IlvDuration` has a single constructor that takes the length of time expressed in milliseconds:

```
IlvDuration(long millis)
```

The `IlvDuration` class also has several convenient static constants that represent commonly used time spans:

- `IlvDuration.ONE_SECOND`
- `IlvDuration.ONE_MINUTE`
- `IlvDuration.ONE_HOUR`
- `IlvDuration.ONE_DAY`
- `IlvDuration.ONE_WEEK`

There are also several methods you can use to perform arithmetic on durations:

- `Date add(Date date)`
- `IlvDuration add(IlvDuration delta)`
- `IlvDuration subtract(IlvDuration delta)`
- `IlvDuration multiply(int multiplier)`

The following example shows how to create a duration of three weeks:

```
IlvDuration threeWeeks = IlvDuration.ONE_WEEK.multiply(3);
```

IlvTimeInterval

The `IlvTimeInterval` class creates time objects that represent an interval of time between a start time and an end time.

You can create time intervals by using the constructors:

- `IlvTimeInterval(Date start, Date end)`
- `IlvTimeInterval(Date start, IlvDuration duration)`

The following example shows how to create a time interval that starts on February 15, 2001 and lasts for one week:

```
Calendar calendar = Calendar.getInstance();
calendar.clear();
calendar.set(2001, Calendar.FEBRUARY, 15);
Date start = calendar.getTime();
IlvTimeInterval interval = new IlvTimeInterval(start,
                                               IlvDuration.ONE_WEEK);
```

Unlike the `Date` and `IlvDuration` classes, the `IlvTimeInterval` class is mutable.

You can manipulate a time interval using the following methods:

- `Date getStart()`
- `void setStart(Date t)`
- `Date getEnd()`
- `void setEnd(Date t)`
- `void setInterval(Date start, Date end)`
- `void setInterval(Date start, IlvDuration duration)`
- `IlvDuration getDuration()`
- `void setDuration(IlvDuration duration)`
- `boolean overlaps(IlvTimeInterval interval)`
- `boolean contains(Date time)`

Connecting to data

Explains how to connect to a database through JDBC and how to integrate data in XML files. Indicates how to use an in-memory data model for test purposes and how to go about integrating a custom data model.

In this section

Connecting to data in-memory
Explains how to connect and populate the data model using data in-memory.

Connecting to XML data
Describes SDXL and how to use this language to serialize schedule data.

Connecting to Swing TableModel instances
Provides an overview on how to display Gantt data from a Swing TableModel

Connecting to data through JDBC
Explains the information required for a JDBC™ Gantt data model object, and how to establish the connection between data and model.

Implementing custom data models
Describes the cases where a custom data model may be used and points you to example applications that show how to implement a custom data model.

© Copyright IBM Corp. 1987, 2009

Connecting to data in-memory

Explains how to connect and populate the data model using data in-memory.

In this section

When to use data in-memory
Explains the main use for data in-memory and describes how to connect to it.

Activities and resources
Describes the stages necessary to populate your data model with activities and resources.

Populating the data model
Describes how to populate your data model with activities and resources.

Manipulating activities and resources
Describes the methods available to manipulate activities and resources within the data model.

Activity and resource factories
Explains how create activities and resources based on the factory design pattern.

Constraints
Explains the associations between the classes needed to model constraints, and the prerequisite to adding a constraint and the constraint factory.

Reservations
Explains the associations between the classes needed to model reservations, and the prerequisite to adding a reservation and the reservation factory.

When to use data in-memory

This access mode is often used to supply data for testing dynamic styling. You can use it to simulate situations where you have designed a change in the representation of the business data to flag specific cases, such as the crossing of a threshold. You can find such test data in:

`<installdir>/jviews-gantt86/samples/ganttChart/src/shared/data/SimpleEngineeringProject.java`

Connecting to data in-memory implies populating the data model. You can do so *before* or *after* a chart has been bound to it.

- If you do it before binding a chart, the initial data will be immediately displayed by the chart when it binds to the data model.
- If you do it after binding a chart, the chart will update dynamically to reflect the new data in the data model.

For better performance, the recommended procedure consists in populating your data model before the chart is bound, especially if you have a large dataset.

You populate the data model with:

- *Activities and resources*
- *Constraints*
- *Reservations*

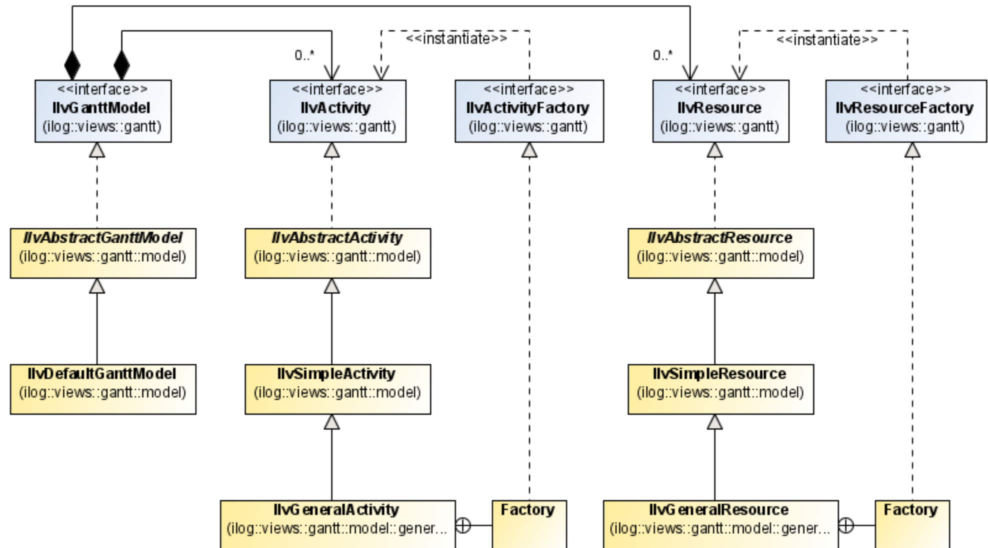
You can clear all entities from the data model by using the `clear()` method defined in the `IlvAbstractGanttModel` base class. This will depopulate all activities, resources, constraints, and reservations from the data model.

Activities and resources

An activity is a task that must be completed in the schedule. One or more resources can be allocated to an activity to enable its completion. Activities are defined by the `IlvActivity` interface and resources by the `IlvResource` interface. Both are stored in a hierarchical structure within the data model and are subinterfaces of the `IlvHierarchyNode` interface.

Each activity can have 0, 1, or more child activities. Similarly, each resource can have 0, 1, or more child resources. An activity or resource with at least one child is called a parent activity or parent resource. Conversely, an activity or resource with no children is called a leaf activity or leaf resource. The top level of hierarchical trees of activities and resources is called the root activity and root resource. Each activity and resource in the data model is a child of its parent, except for the roots, which have no parent.

The following figure shows the associations between the classes needed to model constraints.



Populating the data model

Explains how to set `IlvActivity` and `IlvResource` instances to an `IlvGanttModel` object.

To populate your data model with activities and resources:

1. Populate a data model with activities by establishing the root `IlvActivity` object:

```
IlvGanttModel ganttModel = new IlvDefaultGanttModel();
IlvActivity rootActivity = new IlvGeneralActivity(...);
ganttModel.setRootActivity(rootActivity);
```

2. Add more activities to the data model using either of the following `IlvGanttModel` methods:

   - `void addActivity(IlvActivity newActivity, IlvActivity parent)`
   - `void addActivity(IlvActivity newActivity, IlvActivity parent, int)`

   For example, here you add a child activity to the root activity that was created in the data model:

```
IlvActivity childActivity = new IlvGeneralActivity(...);
ganttModel.addActivity(childActivity, rootActivity);
```

3. Populate the data model with resources by establishing the root `IlvResource` object:

```
IlvGanttModel ganttModel = new IlvDefaultGanttModel();
IlvResource rootResource = new IlvGeneralResource(...);
ganttModel.setRootResource(rootResource);
```

4. Add more resources to the data model using either of the following `IlvGanttModel` methods:

   - `void addResource(IlvResource newResource, IlvResource parent)`
   - `void addResource(IlvResource newResource, IlvResource parent, int)`

   For example, here you add a child resource to the root resource just added to the data model:

```
IlvResource childResource = new IlvGeneralResource(...);
ganttModel.addResource(childResource, rootResource);
```

Manipulating activities and resources

By continuing to add activities and resources in the manner described in *Populating the data model*, you populate the Gantt data model with your scheduling data. The following table shows the `IlvGanttModel` methods that allow you to manipulate the activities and resources within the data model:

Activities	Resources
`addActivity` (two signatures)	`addResource` (two signatures)
`getRootActivity`	`getRootResource`
`moveActivity`	`moveResource`
`removeActivity` (two signatures)	`removeResource` (two signatures)
`setRootActivity`	`setRootResource`

Activity and resource factories

The Gantt chart and Schedule chart beans provide a flexible way to create activities and resources, based on the *factory* design pattern.

The `IlvHierarchyChart` class, which is the superclass of both `IlvGanttChart` and `IlvScheduleChart`, contains an activity factory defined by the `IlvActivityFactory` interface. This interface has only one method:

```
IlvActivity createActivity(IlvTimeInterval interval)
```

In the Gantt chart and Schedule chart examples, whenever you create a new activity using the mouse, the interactor asks the chart factory to create the actual `IlvActivity` object by calling this method. (For information on interactors, see *Interacting with the Gantt charts*.) By default, the chart activity factory is an instance of `IlvGeneralActivity.Factory`.

You can use the following `IlvHierarchyChart` methods to change this:

- `IlvActivityFactory getActivityFactory()`
- `void setActivityFactory(IlvActivityFactory factory)`

In this manner, the decision as to what type of activity to create is dissociated from the interactor, which only determines when the activity should be created based upon mouse events.

The activity factory can also be used to remove the hard-coded dependency on a specific `IlvActivity` implementation from your own code. For example, instead of writing:

```
IlvActivity rootActivity = new IlvGeneralActivity(...);
```

as in the previous section, you could write the following code:

```
IlvActivityFactory activityFactory = myChart.getActivityFactory();
IlvActivity rootActivity = activityFactory.createActivity(...);
```

Note: The class `IlvGeneralActivity.Factory` creates each new activity with a default name and identifier of “New Activity”, located in resource files for easier localization. You will probably want to modify these default attributes before adding the new activity to your data model.

The Gantt chart and Schedule chart examples use this technique to populate the data model.

Note: The Resource Data chart does not support interactive creation of data model objects. Therefore, it does not contain any API for data model factories.

In addition to an activity factory, the class `IlvScheduleChart` also inherits a resource factory from the class `IlvHierarchyChart`. This factory is defined by the `IlvResourceFactory` interface. Like the activity factory, this interface has only one method:

```
IlvResource createResource()
```

By default, the chart resource factory is an instance of `IlvGeneralResource.Factory`.

However, you can use the following `IlvHierarchyChart` methods to change this:

- `IlvResourceFactory getResourceFactory()`
- `void setResourceFactory(IlvResourceFactory factory)`

You can use the resource factory to create resource objects for your data model:

```
IlvResourceFactory resourceFactory = myChart.getResourceFactory();
IlvResource rootResource = resourceFactory.createResource();
```

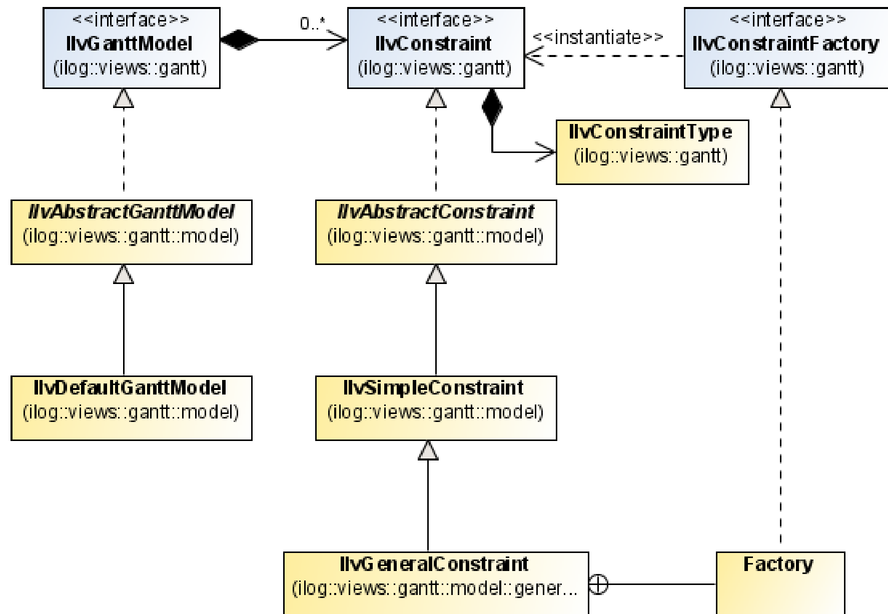
Note: The `IlvGeneralResource.Factory` class creates each new resource with a default name and identifier of “New Resource”, located in resource files for easier localization. You will probably want to modify these default attributes before adding the new resource to your data model.

Constraints

You can create a constraint between two activities. (See Constraints in *Introducing JViews Gantt*.) A constraint is defined by the `IlvConstraint` interface. It also has a type, defined by the `IlvConstraintType` class whose four static constants define the supported constraint types:

- `IlvConstraintType.START_START`
- `IlvConstraintType.START_END`
- `IlvConstraintType.END_START`
- `IlvConstraintType.END_END`

The following figure shows the associations between the classes needed to model constraints.



Prerequisites to adding a constraint

The `IlvConstraintType` class has a private constructor, so that no other instances can be created. Think of this feature as the Java™ equivalent of a C++ enumerated type. Before you can add a constraint to the Gantt data model, the two activities involved must already be members of the data model. For example:

```
IlvGanttModel ganttModel = new IlvDefaultGanttModel();
IlvActivity rootActivity = new IlvGeneralActivity(...);
ganttModel.setRootActivity(rootActivity);
```

```
IlvActivity child1 = new IlvGeneralActivity(...);
ganttModel.addActivity(child1, rootActivity);
IlvActivity child2 = new IlvGeneralActivity(...);
ganttModel.addActivity(child2 rootActivity);
// Create a constraint between child1 and child2
IlvConstraint constraint =
    new IlvGeneralConstraint(child1, child2, IlvConstraintType.END_START);
ganttModel.addConstraint(constraint);
```

If either of the constrained activities is removed from the data model, the constraint will also be removed. This avoids “loose” constraints and maintains the invariant whereby a constraint always links two activities in the same Gantt data model.

The following `IlvGanttModel` methods allow you to manipulate the constraints within the data model:

- `void addConstraint(IlvConstraint newConstraint)`
- `void removeConstraint(IlvConstraint constraint)`
- `Iterator constraintIterator()`
- `Iterator constraintIteratorFromActivity(IlvActivity fromActivity)`
- `Iterator constraintIteratorToActivity(IlvActivity toActivity)`

The constraint factory

As with activities and resources, the `IlvHierarchyChart` class also contains a constraint factory, defined by the `IlvConstraintFactory` interface. Like activity and resource factories, this interface has only one method:

```
IlvConstraint createConstraint(IlvActivity from,
                               IlvActivity to,
                               IlvConstraintType type)
```

By default, the chart constraint factory is an instance of the `IlvGeneralConstraint.Factory` class.

However, you can use the following methods of the class `IlvHierarchyChart` to change this:

- `IlvConstraintFactory getConstraintFactory()`
- `void setConstraintFactory(IlvConstraintFactory factory)`

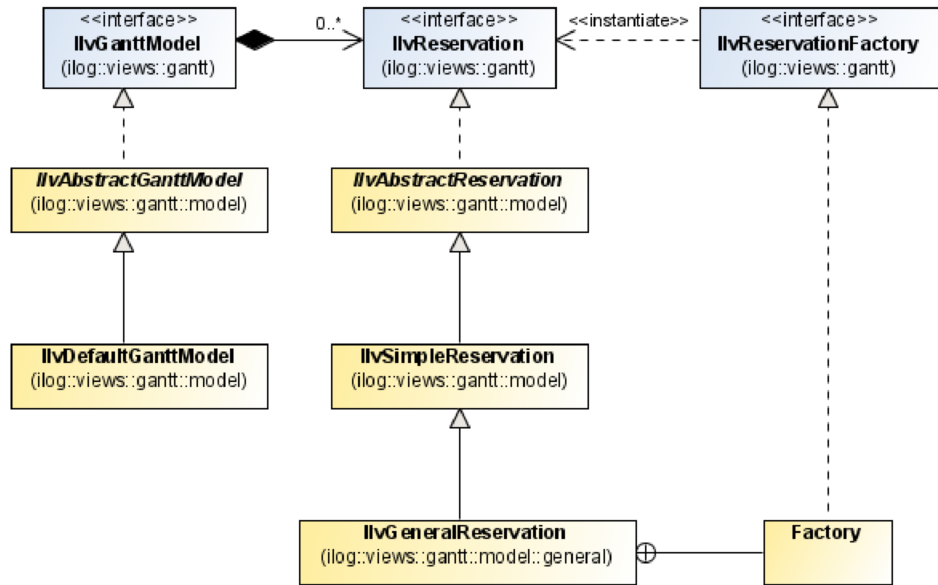
You can use the constraint factory to create constraint objects for your data model:

```
IlvConstraintFactory constraintFactory = myChart.getConstraintFactory();
IlvConstraint constraint =
     constraintFactory.createConstraint
          (activity1, activity2, IlvConstraintType.END_START);
```

Reservations

A reservation is created between a resource and an activity. Another way to think of this is that the activity has “reserved” the resource for its execution. A reservation is defined by the `IlvReservation` interface.

The following figure shows the associations between the classes needed to model reservations.



Prerequisite to adding a reservation

Before you can add a reservation to the Gantt data model, both the resource and the activity must already be members of the data model. For example:

```
IlvGanttModel ganttModel = new IlvDefaultGanttModel();
IlvActivity rootActivity = new IlvGeneralActivity(...);
ganttModel.setRootActivity(rootActivity);
IlvActivity childActivity = new IlvGeneralActivity(...);
ganttModel.addActivity(childActivity, rootActivity);
IlvResource rootResource = new IlvGeneralResource(...);
ganttModel.setRootResource(rootResource);
IlvResource childResource = new IlvGeneralResource(...);
ganttModel.addResource(childResource, rootResource);
// Create a reservation between childActivity and childResource
IlvReservation r = new IlvGeneralReservation(childResource, childActivity);
ganttModel.addReservation(r);
```

If either the activity or the resource is removed from the data model, the reservation will also be removed. This avoids “loose” reservations and maintains the invariant whereby a reservation always links an activity to a resource in the same Gantt data model.

The following `IlvGanttModel` methods allow you to manipulate the reservations within the data model:

- `void addReservation(IlvReservation newReservation)`
- `void removeReservation(IlvReservation reservation)`
- `Iterator reservationIterator()`
- `Iterator reservationIterator(IlvActivity activity)`
- `Iterator reservationIterator(IlvResource resource)`
- `Iterator reservationIterator(IlvResource resource, IlvTimeInterval interval)`

The reservation factory

As in *Activity and resource factories* and *The constraint factory*, the `IlvHierarchyChart` class also contains a reservation factory, defined by the `IlvReservationFactory` interface. As with the other equivalent interfaces, this interface has only one method:

```
IlvReservation createReservation(IlvResource resource, IlvActivity activity)
```

By default, the chart reservation factory is an instance of the `IlvGeneralReservation.Factory` class.

However, you can use the following methods of the `IlvHierarchyChart` class to change this:

- `IlvReservationFactory getReservationFactory()`
- `void setReservationFactory(IlvReservationFactory factory)`

You can use the reservation factory to create reservation objects for your data model:

```
IlvReservationFactory reservationFactory = myChart.getReservationFactory();
IlvReservation reservation =
     reservationFactory.createReservation(aResource, anActivity);
```

Connecting to XML data

Describes SDXL and how to use this language to serialize schedule data.

In this section

The SDXL format
Explains where to find more information about the SDXL format.

The schedule data exchange language
Explains the SDXL language, presents the design criteria of the language as well as some scenarios of how it can be used.

How to write an IlvGanttModel to an SDXL file using serialization
Describes how to write `IlvGanttModel` objects using the `ilog.views.gantt.xml` package.

How to read an IlvGanttModel from an SDXL file using serialization
Describes how to read `IlvGanttModel` objects by using the `ilog.views.gantt.xml` package.

Handling exceptions while reading SDXL files
Describes how to handle exceptions during serialization.

The SDXL format

JViews Gantt allows you to serialize schedule data to Schedule Data Exchange Language (SDXL) files. Use the classes in `ilog.views.gantt.xml`. You can find a description of the SDXL format in Content and structure of an XML data file in *Using the Designer*. You can also find the DTD specification of the SDXL format in *Document type definition for SDXL*.

You can find the DTD file in:

`<installdir>/jviews-gantt86/data/sdxl.dtd`

The schedule data exchange language

The Gantt chart (`IlvGanttChart`) and the Schedule chart (`IlvScheduleChart`) are designed to visualize and to edit schedule data in a JViews Gantt model (`IlvGanttModel`). Users of the Gantt chart and the Schedule chart need first to save their schedule data and then to exchange the schedule data with other users.

SDXL is an application of W3C XML. It is designed to meet the following needs:

- Serialize the schedule data (`IlvGanttModel`) represented by an `IlvGanttChart` or an `IlvScheduleChart`. This allows users to save their schedule data to SDXL files and to load the saved schedule data from SDXL files.

- Exchange the schedule data with other programs developed with or without JViews Gantt. Since SDXL is an application of W3C XML, it can be easily read by other programs that are capable of reading XML files. It can also be translated to other formats by using technologies such as XSL.

Scenarios of how SDXL can be used

Since SDXL is a flexible XML application, its usage is not limited in scope.

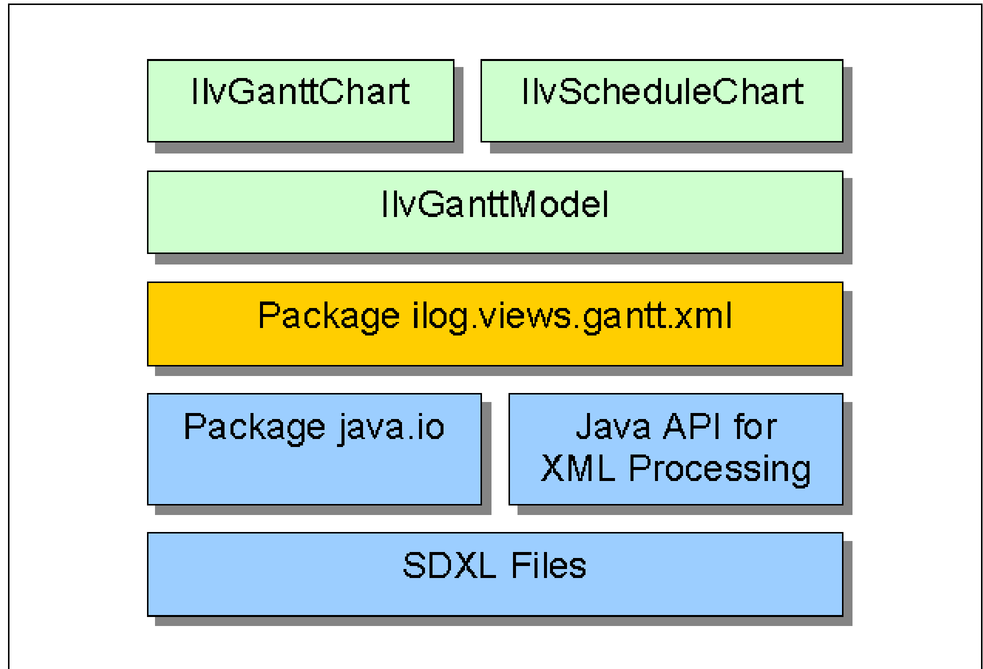
However, to give a general idea, imagine the following scenarios:

- You use a JViews Gantt program to manage your projects. The program is heavy because it is connected to a database and uses the database to store the schedule data. SDXL can help distribute this schedule data. You can save your schedules to SDXL files and distribute them by means of a lightweight JViews Gantt program that does not need database connections.

- You use a JViews Gantt program to manage your schedules. You can save your schedules to SDXL files. An optimization program loads the SDXL files and runs optimization algorithms to make your schedules more efficient. Then, you reload the optimized schedules by using your JViews Gantt program to visualize them.

API for reading and writing SDXL

The following figure shows that the `ilog.views.gantt.xml` contains all the classes that make it possible to serialize schedule data to SDXL files:



The previous figure shows how the `ilog.views.gantt.xml` package can serialize schedule data contained in an `IlvGanttModel` to SDXL files. The `ilog.views.gantt.xml` package is based on `java.io` and JAXP (Java™ API for XML Processing) since it has to use `java.io` APIs and JAXP to read and write SDXL files.

To be able to use the `ilog.views.gantt.xml` package, you need both:

- The JViews Gantt JAR files.
- Some JAR files of the IBM® ILOG® JViews Framework library, provided in

  <installdir>/jviews-framework86/lib/external

To use these libraries, you need a thin and lightweight API that provides a standard way to seamlessly integrate any XML-compliant parser with a Java application. JViews Gantt comes with the Apache™ Xerces parser implementation.

The `ilog.views.gantt.xml` package is designed to serialize schedule data defined by the following classes:

- `IlvSimpleActivity` and `IlvGeneralActivity`
- `IlvSimpleResource` and `IlvGeneralResource`
- `IlvSimpleReservation` and `IlvGeneralReservation`
- `IlvSimpleConstraint` and `IlvGeneralConstraint`
- `IlvDefaultGanttModel`

If your Gantt data model is exclusively defined by these classes, the `ilog.views.gantt.xml` package contains all the classes you need to serialize your schedule data. The following table lists by level the readers and writers available in the package. An interface is given for each reader or writer. This interface defines the functionality the reader or the writer must implement. One of the benefits of using interfaces is that you can interchange readers or writers that implement the same interface. This makes the package flexible and customizable.

For each reader and writer, the package provides two default implementations. The "simple" implementations read and write the default data model classes, while the "general" implementations read and write the data model classes that support user-defined properties.

These can be used as-is.

Level	Functions	Interfaces	Default implementations in the ilog.views.gantt.xml package

Level 1: Element readers and writers	Read/write an activity, a resource, a reservation, or a constraint from/to an element	`IlvActivityReader`
		`IlvActivityWriter`
		`IlvResourceReader`
		`IlvResourceWriter`
		`IlvReservationReader`
		`IlvReservationWriter`
		`IlvConstraintReader`
		`IlvConstraintWriter`
			`IlvSimpleActivityReader`
			`IlvSimpleActivityWriter`
			`IlvSimpleResourceReader`
			`IlvSimpleResourceWriter`
			`IlvSimpleReservationReader`
			`IlvSimpleReservationWriter`
			`IlvSimpleConstraintReader`
			`IlvSimpleConstraintWriter`
			`IlvGeneralActivityReader`
			`IlvGeneralActivityWriter`
			`IlvGeneralResourceReader`
			`IlvGeneralResourceWriter`
			`IlvGeneralReservationReader`
			`IlvGeneralReservationWriter`
			`IlvGeneralConstraintReader`
			`IlvGeneralConstraintWriter`

Level 2: Document reader and writer	Read/write a Gantt data model from/to a document		`IlvGanttDocumentReader` `IlvGanttDocumentWriter`
Level 3: Stream writer	Write a document to an `OutputStream`		`IlvGanttStreamWriter`

The readers and writers are arranged by levels. Level 1 is the lowest level and level 3 is the highest. In most cases, the readers and writers of level N are based on the readers and writers of level N-1. For example, to read a Gantt data model from a document, the

IlvGanttDocumentReader (level 2) uses element readers (level 1), that is, IlvSimpleActivityReader, IlvSimpleResourceReader, IlvSimpleReservationReader, and IlvSimpleConstraintReader.

- IlvGanttDocumentReader is at the highest level (level 2) among the readers. This reader can read an IlvGanttModel instance from a document. The document is the only input of the package. To read a Gantt data model, the user must provide a document object. See *How to read an IlvGanttModel from an SDXL file using serialization* for information on how to create a document from an SDXL file by using JAXP.

- IlvGanttStreamWriter is at the highest level (level 3) among the writers. It is capable of writing an IlvGanttModel instance to an OutputStream object. The OutputStream is the only output point of the package. Users who want to write an SDXL file should provide an OutputStream. See *How to write an IlvGanttModel to an SDXL file using serialization* for information on how to create an OutputStream object for an SDXL file by using the java.io package.

Customizing readers and writers

The default readers and writers provided by JViews Gantt are sufficient to serialize default Gantt data models. If you created a customized Gantt data model, you need to customize the default readers and writers in order to serialize the customized schedule data. All level readers and writers are customizable. You can customize them either by creating subclasses of the default readers and writers implemented, or by writing you own readers or writers that implement the reader or writer interfaces.

How to write an IlvGanttModel to an SDXL file using serialization

This section shows how to use the main classes of the `ilog.views.gantt.xml` package. You will learn how to write an `IlvGanttModel` to an SDXL file and then how to read back an `IlvGanttModel` from an SDXL file.

Two examples show how to serialize schedule data:

- **<installdir>/jviews-gantt86/samples/xmlGantt/src/xml/XMLGanttExample.java** presented in:

  <installdir>/jviews-gantt86/samples/xmlGantt.

- **<installdir>/jviews-gantt86/samples/xmlSchedule/src/xml/XMLScheduleExample.java** presented in:

  <installdir>/jviews-gantt86/samples/xmlSchedule.

To write the contents of an `IlvGanttModel` to an SDXL file, follow these recommended stages:

1. *Creating an org.w3c.dom.Document*: Use your XML Java™ API to create an instance of `org.w3c.dom.Document`.
2. *Creating an IlvGanttDocumentWriter*: Use the `ilog.views.gantt.xml` package to create an instance of `IlvGanttDocumentWriter`.
3. *Writing a Gantt data model to a document*: Use the `IlvGanttDocumentWriter` to write your `IlvGanttModel` to the document you created in Step 1.
4. *Creating an output stream*: Use the `java.io` package to create a `java.io.OutputStream` object for the SDXL file you want to write to.
5. *Creating a stream writer*: Use the `ilog.views.gantt.xml` package to create an instance of `IlvGanttStreamWriter`.
6. *Writing a document to an output stream*: Use the stream writer to write the document to the output stream you created in Step 4.

Creating an org.w3c.dom.Document

This section shows you how to use the `javax.xml.parsers` package to create an `org.w3c.dom.Document` instance. This is one of many ways of creating document objects. You can, of course, use other ways.

To create a new `org.w3c.dom.Document` instance:

1. Import the packages:

```
import javax.xml.parsers.*;
import org.w3c.dom.*;
```

2. Get an instance of the `DocumentBuilderFactory`:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

3. Get an instance of `DocumentBuilder` from the `DocumentBuilderFactory`:

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

4. Use the `DocumentBuilder` to create the document object:

```
Document document = builder.newDocument();
```

See

<installdir>/jviews-gantt86/samples/xmlGantt/src/xml/XMLGanttActions.java

for a concrete implementation.

Creating an IlvGanttDocumentWriter

Use the `ilog.views.gantt.xml` package to create an instance of `IlvGanttDocumentWriter`. Then the document writer is ready to write an `IlvGanttModel` to a document.

The `IlvGanttDocumentWriter` constructor takes a locale as argument that affects how dates are serialized. In the example the document writer is configured to use the current locale.

To create a new `IlvGanttDocumentWriter` instance:

1. Import the package:

```
import ilog.views.gantt.xml.*;
```

2. Create the `IlvGanttDocumentWriter`:

```
IlvGanttDocumentWriter documentWriter =
    new IlvGanttDocumentWriter(Locale.getDefault());
```

Writing a Gantt data model to a document

The `IlvGanttDocumentWriter` has a method called `writeGanttModel`.

To write your Gantt data model:

♦ Call `writeGanttModel`

```
documentWriter.writeGanttModel(document, yourGanttModel);
```

The `document` argument is the document object you created in *Creating an org.w3c.dom.Document*. The argument `yourGanttModel` is the `IlvGanttModel` you want to write to the document.

Creating an output stream

You can use the `java.io` package to create an `OutputStream`. This is one of many ways of creating output streams. You can, of course, use other ways.

To create an `OutputStream` instance:

1. Import the `java.io` package:

```
import java.io
```

2. Create an `OutputStream` for the file you want to write to:

```
String filename = "c:\mysdxl.xml";
FileOutputStream outstream = new FileOutputStream(filename);
```

The argument `filename` is the name of the SDXL file you want to write to.

Creating a stream writer

After creating the `OutputStream`, you need a utility class that helps you write the document to the stream. The `ilog.views.gantt.xml` package provides a stream writer named `IlvGanttStreamWriter`. You can directly create an instance of this class and use it to write your document. The stream writer is ready to write a document object to an `OutputStream`.

To create an `OutputStream` instance:

1. Import the `ilog.views.gantt.xml` package:

```
import ilog.views.gantt.xml
```

2. Create the stream writer:

```
IlvGanttStreamWriter streamWriter = new IlvGanttStreamWriter();
```

Writing a document to an output stream

Now that you have the `OutputStream` instance and the document, you can write your document to the `OutputStream` object.

To do this:

1. Call the `writeDocument` method of the stream writer created in the previous step to write the document to the `OutputStream`.

```
streamWriter.writeDocument(outstream, document);
```

2. Close the output stream:

```
outstrem.close();
```

Your `IlvGanttModel` is written to an SDXL file.

How to read an IlvGanttModel from an SDXL file using serialization

Change the itemized list back to a numbered list in XML.

To read the contents of an SDXL file to an `IlvGanttModel`, follow these recommended stages:

- *Creating an input source*: Use your XML Java™ API to create an `InputSource` for the file you want to read.
- *Parsing an input source*: Use Java XML parser to parse the `InputSource` in order to get an `org.w3c.dom.Document`.
- *Creating an IlvGanttDocumentReader*: Use the `ilog.views.gantt.xml` package to create an instance of `IlvGanttDocumentReader`.
- *Reading a Gantt data model from a document*: Create a target `IlvGanttModel` to receive the schedule data and use the `IlvGanttDocumentReader` to read the document created in Step 2 to the `IlvGanttModel` created in Step 4.
- *Handling exceptions while reading SDXL files*

These stages are seen in more detail in the subsequent sections.

Creating an input source

You can use your XML Java API to create an `InputSource` instance for the SDXL file you want to read. This section presents one of many ways of creating input sources. You can, of course, use other ways.

To create an `InputSource` instance for the SDXL file you want to read:

1. Import the package:

   ```
   import org.xml.sax.*;
   ```

   Suppose you want to read an SDXL file named `/nfs/works/myschedule.xml`:

   ```
   String url = "file:///nfs/works/myschedule.xml";
   ```

2. You can create directly an `InputSource` for the file to read:

   ```
   InputSource source = new InputSource(url);
   ```

The input source is ready to be parsed by your XML parser.

Parsing an input source

The `DocumentBuilder` provided by your XML Java API can parse an input source. You can create an instance of the builder and use it to parse the input source.

To parse an input source:

1. Import the packages:

   ```
   import javax.xml.parsers.*;
   import org.w3c.dom.*;
   ```

2. Get an instance of the `DocumentBuilderFactory`:

```
DocumentBuilderFactory factory = DocumentBuilderFactory.newInstance();
```

3. Get an instance of `DocumentBuilder` from the `DocumentBuilderFactory`:

```
DocumentBuilder builder = factory.newDocumentBuilder();
```

4. Use the `DocumentBuilder` to parse the input source:

```
Document document = builder.parse(source);
```

Now you need an `IlvGanttDocumentReader` instance to read the document.

Creating an IlvGanttDocumentReader

The `ilog.views.gantt.xml` package provides the `IlvGanttDocumentReader` class.

To create an instance of this class:

1. Import the package:

```
import ilog.views.gantt.xml.*;
```

2. Create the `IlvGanttDocumentReader`:

```
IlvGanttDocumentReader documentReader = new IlvGanttDocumentReader();
```

The document reader is ready to read a document.

Reading a Gantt data model from a document

The `IlvGanttDocumentReader` class has a method called `readGanttModel`. You can call this method to read a JViews Gantt model from a document. Before reading the document you must create a target `IlvGanttModel` to receive the schedule data.

To read a Gantt data model from a document:

1. Import the `ilog.views.gantt` and `ilog.views.gantt.model` packages:

```
import ilog.views.gantt.*;
import ilog.views.gantt.model.*;
```

2. Create a default Gantt data model:

```
IlvGanttModel model = new IlvDefaultGanttModel();
```

3. Read the Gantt data model from the document:

```
documentReader.readGanttModel(document, model);
```

The `document` argument is the document object you created in the previous step.

Your `IlvGanttModel` has now been read from an SDXL file.

Handling exceptions while reading SDXL files

Exceptions might be encountered during the reading of the document. The exceptions are reported by `IlvGanttReaderException` objects. The following example shows how to handle such exceptions.

```
try {
  docReader.readGanttModel(document, model);
} catch(IlvGanttReaderException e) {
  //...
```

Connecting to Swing TableModel instances

Provides an overview on how to display Gantt data from a Swing TableModel

In this section

Overview
Describes the classes used to display Gantt data or access a database through JDBC.

Data required by IlvTableGanttModel
Describes the data required for activities, resources, constraints and reservations.

Converting TableModel data to IlvTableGantt Model data
Explains the TableModel instances used to configure an `IUlvTableGanttModel` object.

Configuring the IlvTableGanttModel object correctly
Describes the steps necessary to configure an `IUlvTableGanttModel` object with two `TableModel` instances.

Read/write support
Explains how once you have imported the data, modifications applied to the `IlvTableGanttModel` instance are passed along to the TableModel object.

Dynamic behavior
Describes how to handle changes to the TableModel instances during the lifetime of the application dynamically modify the `IlvTableGanttModel` object.

Reading data from CSV files
Explains how to use the import Gantt data from CSV files using the JViews Gantt and IBM® ILOG® JViews Framework APIs.

Complex mappings

Explains how to map columns to a time-interval property.

Overview

The class `IlvTableGanttModel` is a specific implementation of the `IlvGanttModel` interface that allows you to display Gantt data coming from Swing `TableModel` instances.

A subclass of `IlvTableGanttModel`, called `IlvJDBCGanttModel`, is also provided in the library to enable you to load data from databases through instances of a `TableModel` implementation, `IlvRowSetTableModel`, which can access a database through JDBC™ . If this is the implementation you need, see *Connecting to data through JDBC*.

For a table Gantt data model (class `IlvTableGanttModel`), data pertaining to activities, resources, constraints, and reservations must be contained in a `TableModel` instance. A set of configuration parameters allow you to map this data to the information that the `IlvTableGanttModel` object needs to build its contents.

Data required by IlvTableGanttModel

A table Gantt data model requires the following information:

For activities (IlvTableActivity)

The following table shows the data required by IlvTableGanttModel for activities.

Property description	Time Information			ID	Parent ID	Name
	Start Time	End Time	Time Interval			
Property name	START_TIME_PROPERTY	END_TIME_PROPERTY	TIME_INTERVAL_PROPERTY	ID_PROPERTY	PARENT_ID_PROPERTY	NAME_PROPERTY
Required type	Date	Date	IlvTimeInterval	String	String (can be null)	String

Note: The time information is required either as delimiters (start/end) or as a time interval

For resources (IlvTableResource)

The following table shows the data required by IlvTableGanttModel for resources.

Property description	Quantity	ID	Parent ID	Name
Property name	QUANTITY_PROPERTY	ID_PROPERTY	PARENT_ID_PROPERTY	NAME_PROPERTY
Required type	Float	String	String (can be null)	String

For constraints (IlvTableConstraint)

The following table shows the data required by `IlvTableGanttModel` for constraints.

Property description	Constraint Type	Identifier of the From activity	Identifier of the To activity
Property name	TYPE_PROPERTY	FROM_ACTIVITY_ID	TO_ACTIVITY_ID
Required type	`IlvConstraintType`	String	String

For reservations (IlvTableReservation)

The following table shows the data required by `IlvTableGanttModel` for reservations.

Property description	ID of the activity	ID of the resource
Property name	ACTIVITY_ID_PROPERTY	RESOURCE_ID_PROPERTY
Required type	String	String

The column types in the `TableModel` instance must not necessarily be the same as the ones required by the `IlvTableGanttModel` properties. If they are different, the `IlvTableGanttModel` object must be configured such as to convert the values to the right type, as explained in the next section *Converting TableModel data to IlvTableGantt Model data*.

Converting TableModel data to IlvTableGantt Model data

Suppose you have the following two `TableModel` instances:

Activities

The following table represents a `TableModel` instance for activities. The column name is returned by the method `TableModel.getColumnName(int)` and the column type is returned by the method `TableModel.getColumnClass(int)`.

Column name	Category (0)	ID (1)	StartTime (2)	EndTime (3)	ParentID (4)
Column type	String	String	String (formatted M/d/yy)	String(formatted M/d/yy)	String

You can see that the `TableModel` instance contains:

- the required `ID` and `parent ID` data with the right type;
- start-time and end-time data, but expressed as String objects whereas `IlvTableGanttModel` requires Date objects;
- no name information;
- a `Category` column that does not correspond to any required property.

Constraints

The following table represents a `TableModel` instance for constraints. The column name is returned by the method `TableModel.getColumnName(int)` and the column type is returned by the method `TableModel.getColumnClass(int)`.

Column name	Type (0)	FromActivityID (1)	ToActivityID (2)
Column type	Integer	String	String

You can see that the `TableModel` instance contains the required `FromActivity` and `ToActivity` IDs with the right type. It also contains the constraint type. However, the latter is expressed as an Integer, instead of an `IlvConstraintType` instance. Therefore, you need to map the corresponding values as shown in the following table:

IlvConstraintType	Integer
START_START	1
START_END	2
END_END	3
END_START	4

For a basic use of `IlvTableGanttModel`, that is, when you import the data from the database to the Gantt data model a single time and as read-only, there are no particular requirements on how the `TableModel` instances are implemented. This will be different with more advanced uses (see *Read/write support* and *Dynamic behavior*).

Configuring the IlvTableGanttModel object correctly

To configure your `IlvTableGanttModel` object correctly with these two `TableModel` instances, do the following:

1. **Create the table Gantt data model**

   This is done by instantiating `IlvTableGanttModel`:

```
IlvTableGanttModel tableModel = new IlvTableGanttModel();
```

2. **Create the mapper for activities**

   The activities mapper is taken from the activities table model and the mapping instructions for the required properties.

   The arguments after the table model are:

   - the ID column,
   - the name column,
   - the start time column,
   - the end time column, and
   - the parent ID column.

   Indexes of columns are used in the code sample below but you could also use column names:

```
IlvTableModelMapper activityMapper =
    IlvTableGanttModel.createActivityMapper(activityModel, 1, 1, 2, 3,
4);
```

   As the TableMode instance does not provide name information, the ID column is used instead (index 1).

   The tables in *Constraints* show that the `StartTime` and `EndTime` columns do not use the required type; therefore, you should also, theoretically, register a converter from the column actual type (String) to the type required by `IlvTableGanttModel` (Date). Practically, this is not mandatory in this case because this particular converter is already registered by default. However, if it was necessary, the code would be the following:

```
IlvConvert.addConverter(new IlvConverter() {
  private final DateFormat formatter =
       DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
  public Object convert(Object value, Class toType)
                              throws IlvConvertException {
   try {
    return formatter.parse((String)value);
   } catch (ParseException e) {
    throw new IlvConvertException("error");
   }
}
public Class[] fromTypes() {
```

```
    return new Class[] {String.class};
  }
  public Class[] toTypes() {
    return new Class[] {Date.class};
  }
}
```

3. Add an instance of IlvTableModelPropertyDescriptor to the mapper

This is done in order to map the optional column (that is, the activity category) available in the table:

```
activityMapper.addPropertyDescriptor("OptionalCategory",
   new IlvBasicTableModelPropertyDescriptor("CATEGORY"), null);
```

These code lines map the contents of the Category column of the `TableModel` instance to the activity property `CATEGORY` of the `IlvTableGanttModel` object without trying to convert the values (null as `requiredType` parameter). The values will then be accessible on each activity by a call to the method `getProperty("CATEGORY")` on the `IlvTableActivity` instance.

At this stage, the mapper for activities is fully configured.

4. Pass it to the IlvTableGanttModel

```
tableModel.setActivityMapper(activityMapper);
```

5. Repeat previous steps

Repeat from *Create the mapper for activities* to *Pass it to the `IlvTableGanttModel`* for the constraints.

Here, you are going to use column names instead of indexes to build the mapper. Note that the last parameter is different from the one used for activities. Its role is to map the different values taken by the constraint types in the `TableModel` (1, 2, 3, or 4 in this case) to the values required by the `IlvTableGanttModel` object, namely `IlvTableConstraintType.START_START`, `START_END`, `END_START`, and `END_END`, in this order:

```
IlvTableModelMapper constraintMapper =
IlvTableGanttModel.createConstraintMapper
                                         (constraintModel,
                                         "FromActivityID",
                                           "ToActivityID",
                                                   "Type",
                             new Object[] {new Integer(1),
                                           new Integer(2),
                                           new Integer(4),
                                           new Integer(3)});
```

6. Carry out any other necessary conversion or add the optional properties

In this example, there are none, so you can move on to the next step.

7. Pass the result to the model

The following code example shows how to do this.

```
tableModel.setConstraintMapper(constraintMapper);
```

All the tables you want to deal with are now configured.

8. **Initialize the model**

You now import the data from the `TableModel` instances:

```
try {
  tableModel.initializeMapping();
} catch (IlvTableModelMappingException e) {
  // in case something went wrong
}
```

If you do nothing else after this initialization step, use the `TableModel` instances and the `IlvTableGanttModel` object as read-only. If you want to add read/write and dynamic capabilities, see *Read/write support* or *Dynamic behavior*.

Read/write support

Users often need to interact with the model through a Schedule or Gantt chart, and then, they want the result of the interaction to be persistent. In such cases, the `TableModel` instance connected to the `IlvTableGanttModel` object must additionally implement the interface `IlvTableModel` which allows you to add or remove rows from the `TableModel` instance.

```
public interface IlvTableModel extends TableModel {
  public void addRow(Object[] rowData);
  public void insertRow(int rowIndex, Object[] rowData);
  public void removeRow(int rowIndex);
}
```

If you do not add this interface, exceptions may be raised when the application tries to modify the `IlvTableGanttModel` object.

The Swing class `DefaultTableModel` already contains the required methods. You can therefore easily use it for read/write operations with `IlvTableGanttModel`. All you have to do is mark a subclass with the interface `IlvTableModel`:

```
public BasicTableModel extends DefaultTableModel
     implements IlvTableModel {
  public BasicTableModel() {
    super();
  }
}
```

From then on, each change to the `IlvTableGanttModel` object will update the `TableModel` object accordingly.

If you want the predefined actions on `IlvHierarchyChart` instances to create the right type of activities, resources, constraints, and reservations, you can configure these instances to use the `IlvTableGanttModel`-aware factories by calling:

```
tableGanttModel.configureHierachyChart(hierachyChart);
```

Dynamic behavior

When importing data from a Swing `TableModel` instance to a Gantt data model, you may also need the Gantt data model to update automatically when the contents of the underlying `TableModel` instances are modified *after* the mapper has been initialized. Such updating does not take place by default because additional events need to be thrown which regular implementations of `TableModel` do not throw. If you need this updating feature, you must code your own `TableModel` implementation to fire the additional events described in the `IlvTableModelEvent` class.

As described in *Read/write support*, it is easy to code a `TableModel` instance from the Swing class `DefaultTableModel` so that it supports dynamic modification of the model *after* initialization. It consists in firing the appropriate events before the parent class methods are called:

```
class BasicTableModel extends DefaultTableModel
    implements IlvTableModel  {
    public BasicTableModel() {
      super();
    }
    public void setDataVector(Vector dataVector, Vector columnIdentifiers) {
      IlvTableModelEvent.fireBeforeTableStructureChanged(this);
      super.setDataVector(dataVector, columnIdentifiers);
    }
    public void removeRow(int row) {
      IlvTableModelEvent.fireBeforeTableRowsDeleted(this, row, row);
      super.removeRow(row);
    }
    public void setColumnCount(int columnCount) {
      IlvTableModelEvent.fireBeforeTableStructureChanged(this);
      super.setColumnCount(columnCount);
    }
  }
```

The events will allow the `IlvTableGanttModel` object to update correctly according to the changes made to the `BasicTableModel` object.

Reading data from CSV files

The class `IlvTableGanttModel` is particularly useful because it allows you to import any kind of data to a JViews Gantt application provided you can read this data from a Swing `TableModel` instance.

One way to do so is to use the subclass `IlvJDBCGanttModel` which allows you to load data from a database via a JDBC™ implementation of the class `TableModel`. See *Connecting to data through JDBC* for details.

Another way is to use the class `IlvTableGanttModel` to read Gantt data from CSV (Comma-Separated Values) files. This section shows an example based on the following file, containing activities information:

```
A1, Root Activity, 1/31/04, 10/17/04,
A2, First Child Activity, 1/31/04, 4/2/04, A1
A3, Second Child Activity, 4/2/04, 10/17/04, A2
```

As for a JDBC connection, IBM® ILOG® JViews Framework provides a utility API that allows a Gantt data model to read the contents of a CSV file from a `TableModel` instance. All you have to do is connect that CSV file to your Gantt data model like this:

```
TableModel activities = null;
try {
  activities = IlvCSVReader.getInstance(',').
                    read(new FileReader("file.csv"));
} catch (IOException e) {
}
IlvTableGanttModel model = new IlvTableGanttModel();
IlvTableModelMapper activitiesMapper =
    IlvTableGanttModel.createActivityMapper(activities, 0, 1, 2, 3, 4);
model.setActivityMapper(activitiesMapper);
try {
  model.initializeMapping();
} catch (IlvTableModelMappingException e) {
}
```

Complex mappings

So far, you have seen only simple mappings: one column of a table corresponds to one property in the Gantt data model with, possibly, type conversion if necessary. However, the JViews Gantt API is generic enough to enable more complex mapping.

For example, instead of mapping the `StartTime` and `EndTime` columns to the start-time and end-time properties of activities, you can map the two columns directly to the time-interval property of activities. Referring again to the example used in *Connecting to Swing TableModel instances*, where the start and end times were respectively available in columns (2) and (3).

To map columns to a time-interval property:

1. Write an `IlvTableModelPropertyDescriptor` implementation:

```
class TimeIntervalPropertyDescriptor
    implements IlvTableModelPropertyDescriptor  {
  private final int[] indexes = new int[2];
  private final DateFormat formatter =
       DateFormat.getDateInstance(DateFormat.SHORT, Locale.US);
  public TimeIntervalPropertyDescriptor(int start, int end) {
    indexes[0] = start;
    indexes[1] = end;
  }
  public Object getProperty(TableModel model, int rowIndex)
   throws IlvTableModelMappingException, IlvConvertException {
    try {
      // get the start & end time on the table
      Date start = formatter.parse((String)model.getValueAt(indexes[0],

                       rowIndex));
      Date end  = formatter.parse((String)model.getValueAt(indexes[1],
                       rowIndex));
      // return the IlvTimeInterval
      return new IlvTimeInterval(start, end);
    } catch (ParseException e) {
      throw new IlvConvertException("error");
    }
  }
  public void setProperty(TableModel model, int rowIndex, Object
propertyValue)
    throws IlvConvertException, IlvTableModelMappingException {
    // extract start & end time from the time interval
    Date start = ((IlvTimeInterval)propertyValue).getStart();
    Date end  = ((IlvTimeInterval)propertyValue).getEnd();
    // set the values back in the table
    model.setValueAt(formatter.format(start), indexes[0], rowIndex);
    model.setValueAt(formatter.format(end), indexes[1], rowIndex);
  }
  public int[] getColumns(TableModel model) {
    return indexes;
  }
}
```

2. Build the activity mapper with it:

```
IlvTableModelMapper activityMapper = new IlvTableModelMapper(activityTable)
;
activityMapper.
  addPropertyDescriptor(IlvTableActivity.TIME_INTERVAL_PROPERTY,
                             new TimeIntervalPropertyDescriptor(2, 3),

                         IlvTimeInterval.class);
activityMapper.
  addPropertyDescriptor(IlvTableActivity.ID_PROPERTY,
          new IlvBasicTableModelPropertyDescriptor(1), String.class);
 // ...
```

3. Associate it with the `IlvTableGanttModel` object.

```
tableModel.setActivityMapper(activityMapper);
```

The JViews Gantt API enables you to code even more complex mappings. However, keep in mind that the more complex the mapping, the more time-consuming it is, which may significantly slow your application because the Gantt data model makes frequent calls to these methods.

Connecting to data through JDBC

Explains the information required for a JDBC™ Gantt data model object, and how to establish the connection between data and model.

In this section

Overview
Provides an overview of a class used to display Gantt data retrieved from a database through a JDBC connection.

Writing queries to populate the data model
Explains the information required for a JDBC Gantt data model object.

The harbor example
Describes the data contained in the harbor database and the queries needed to retrieve the activities, resources and reservations contained in it.

Passing the data to the Gantt data model
Describes that steps necessary to establish the connection between data and model.

Overview

The class `IlvJDBCGanttModel` is a specific implementation of the `IlvGanttModel` interface that allows you to display Gantt data retrieved from a database through a JDBC™ connection and optionally to commit modifications back to the database. This class is based on the class `IlvTableGanttModel`, discussed in *Connection to a Swing TableModel*. You may find it useful to read that section if you intend to use advanced functionality of the class `IlvJDBCGanttModel`.

Writing queries to populate the data model

To populate your Gantt data model from the contents of the database, you need to create appropriate queries to get the necessary data for activities, resources, constraints, and/or reservations, and pass it to your `IlvJDBCGanttModel` instance. This is possible only if the database contains sufficient information.

A JDBC™ Gantt data model object requires the following information:

For Activities (IlvTableActivity)

The following table shows the data required by`IlvJDBCGanttModel` for activities.

Property description	Time Information			ID	Parent ID	Name
	Start Time	End Time	Time Interval			
Property name	`START_TIME_PROPERTY`	`END_TIME_PROPERTY`	`TIME_INTERVAL_PROPERTY`	`ID_PROPERTY`	`PARENT_ID_PROPERTY`	`NAME_PROPERTY`
Required type	`Date`	`Date`	`IlvTimeInterval`	`String`	`String` (can be `null`)	`String`

Note: The time information is required either as delimiters (start/end) or as a time interval.

For resources (IlvTableResource)

The following table shows the data required by `IlvJDBCGanttModel` for resources.

Property description	Quantity	ID	Parent ID	Name
Property name	QUANTITY_PROPERTY	ID_PROPERTY	PARENT_ID_PROPERTY	NAME_PROPERTY
Required type	Float	String	String (can be null)	String

For constraints (IlvTableConstraint)

The following table shows the data required by `IlvJDBCGanttModel` for constraints.

Property description	Constraint Type	ID of the from activity	ID of the to activity
Property name	TYPE_PROPERTY	FROM_ACTIVITY_ID	TO_ACTIVITY_ID
Required type	IlvConstraintType	String	String

For reservations (IlvTableReservation)

The following table shows the data required by `IlvJDBCGanttModel` for reservations.

Property description	ID of the activity	ID of the resource
Property name	ACTIVITY_ID_PROPERTY	RESOURCE_ID_PROPERTY
Required type	String	String

Information about object types can be accessed either from a specific database table, or from several tables, or from a single table. In any case, it is mandatory to build one SQL query for each object type needed by the `IlvJDBCGanttModel` object. The property types available in the database must not necessarily be the same as the ones required by `IlvJDBCGanttModel`. If they are different, the `IlvJDBCGanttModel` object must be configured such as to convert the values to the right type (see the class `IlvTableGanttModel` in *Connecting to Swing TableModel instances*).

The harbor example

This section is based on the `harbor.mdb` example, located in:

<installdir>/jviews-gantt86/bin/designer/data/examples/harbor.mdb

The following table provides information about a harbor that expects ships (the activities) at a given dock (the resource) from a start date to an end date.

Column name	ID	ShipName	StartDate	EndDate	DockName
Column type	String	String	Date	Date	String

Activities

For activities (ships), the `parent ID` property is missing, because there is no hierarchical information on the activities. Therefore, the SQL query will always provide `Null` as the parent ID value, like this:

```
select ID, ShipName, StartDate, EndDate, Null as ParentID from HARBOR
```

Resources

For resources, both the parent ID and the ID of the resource itself are missing. For the parent ID, you will use the same solution as for activities. However, the ID itself cannot be null, so you will use the unique `DockName` column value as the ID of the resource. At this stage, it is not necessary to do anything special, you will do so later during the mapping phase (see *Map the result* in the next section *Passing the data to the Gantt data model*).

The Quantity property is also missing. Clearly, you can set it to ‘1’ in this context since each Dock represents a single slot. The docks are listed several times (one by activity). Therefore, you will group the result so as to have unique Docks. You obtain the following query to get the required information:

```
select DockName, '1' as Quantity, Null as ParentID from HARBOR group by
DockName
```

Constraints

There are no constraints.

Reservations

For reservations, you just have to take the activity ID and the resource (Dock) ID, and merge them into a single query:

```
select ID as ActivityID , DockName as DockID from HARBOR
```

The information is already in the required types, so there is no need for any additional conversion.

Passing the data to the Gantt data model

After you have written the appropriate queries to retrieve the data you need through JDBC™ , you establish the connection between data and model.

To do this:

1. **Build the model**
   The following code example shows how to do this.

   ```
   IlvJDBCGanttModel jdbcModel = new IlvJDBCGanttModel();
   ```

   Alternatively, if you want modifications to the Gantt data model to be committed automatically to the database, you can create the model in read-write mode by passing `false` to the read-only constructor parameter:

   ```
   IlvJDBCGanttModel jdbcModel = new IlvJDBCGanttModel(false);
   ```

   In this case, you need to make sure that the database tables are writable.

2. **Create a connection**
   The connection is created to the database for the three queries.

   ```
   Connection databaseConnection = DriverManager.getConnection(dataBaseURL,

   username, password);
   ```

   The parameters required by `DriverManager` depend on the database you are querying. For more information, refer to the documentation of the JDBC driver you use for your database. The following example shows the code for an MS Access Database.

   ```
   Connection databaseConnection = DriverManager.getConnection(
      "jdbc:odbc:Driver={Microsoft Access Driver(*.mdb)};"+
      "DBQ=data/examples/harbor.mdb", null, null);
   ```

3. **Build a query for activities**
   The following code example shows how to do this.

   ```
   String activitiesQuery = "select ID, ShipName, StartDate, EndDate, Null
   as
   ParentID from HARBOR";
   ```

   For more information, see *Writing queries to populate the data model*:

4. **Map the result**
   The result is mapped to the required properties of the `IlvJDBCGanttModel` object

   Specify either the name of the column in the query (as in the code sample below) or its index, using an integer:

   ```
   Map activitiesMapping = new HashMap();
   activitiesMapping.put(IlvTableActivity.ID_PROPERTY, "ID");
   activitiesMapping.put(IlvTableActivity.NAME_PROPERTY, "ShipName");
   activitiesMapping.put(IlvTableActivity.START_TIME_PROPERTY, "StartDate")
   ```

```
;
activitiesMapping.put(IlvTableActivity.END_TIME_PROPERTY, "EndDate");
activitiesMapping.put(IlvTableActivity.PARENT_ID_PROPERTY, "ParentID");
```

5. **Pass the connection, the query, and the mapping information to the model**

```
jdbcModel.setActivitiesQuery(databaseConnection, activitiesQuery,
activitiesMapping);
```

6. **Repeat some of the previous steps.**

   Repeat from *Create a connection* to *Pass the connection, the query, and the mapping information to the model* for resources and reservations:

```
String resourcesQuery  = "select DockName, '1' as Quantity, Null as
ParentID from HARBOR group by DockName";
Map resourcesMapping = new HashMap();
// as said above, we use the DockName as ID
resourcesMapping.put(IlvTableResource.ID_PROPERTY, "DockName");
resourcesMapping.put(IlvTableResource.NAME_PROPERTY, "DockName");
resourcesMapping.put(IlvTableResource.PARENT_ID_PROPERTY, "ParentID");
resourcesMapping.put(IlvTableResource.QUANTITY_PROPERTY, "Quantity");
jdbcModel.setResourcesQuery(databaseConnection, resourcesQuery,
resourcesMapping);

String reservationsQuery = "select ID as ActivityID , DockName as DockID
 from HARBOR";
Map reservationsMapping = new HashMap();
reservationsMapping.put(IlvTableReservation.ACTIVITY_ID_PROPERTY,
   "ActivityID");
reservationsMapping.put(IlvTableReservation.RESOURCE_ID_PROPERTY,
   "DockID");
jdbcModel.setReservationsQuery(databaseConnection, reservationsQuery,
         reservationsMapping);
```

7. **Specify that the configuration is finished**

   Once all the connections, queries, and mapping information has been passed to the model, specify that the configuration is finished and that the model can populate itself from the database:

```
try {
  jdbcModel.initializeMapping();
} catch (IlvTableModelMappingException e) {
  // in case something went wrong
}
```

Depending on the option you choose when you build the model, the resulting model is a read-only model or a read-write model that can be displayed in a Gantt or Schedule chart. If you choose a read-write model, the model can be modified.

Implementing custom data models

If you need to connect specialized business data to your Gantt, Schedule, or Resource Data chart, it is always best to determine whether there is a way to translate or adapt the data to one of the standard formats supported by JViews Gantt.

The arguments after the table model are:

- Perhaps the business data can be accessed via ODBC and you can then use an ODBC-JDBC™ bridge to load the data using JViews Gantt JDBC connection (see *Connecting to data through JDBC*).
- Perhaps the data can be exported to a custom XML format that can then be translated into SDXL format using XSLT (see *Connecting to XML data*).
- However, you may have a special case where implementing a custom data model implementation is the best and most efficient way to connect to your business data. An example of this might be a live connection to business data that must be selectively filtered or modified before it is displayed in a Gantt chart.

JViews Gantt supports the ability for you to design a custom data model implementation and display your custom data in the standard charts. This is possible because of the model-view separation and the loose coupling between the Gantt data model interfaces and the charts. The data model interfaces and classes provided with JViews Gantt are described in *Data model classes*. Although you can implement your custom data model in its entirety, you are recommend to use at least the abstract classes as your starting point. You can also subclass any of the concrete data model implementation classes if this suits your purposes better.

Examples

The database examples illustrate how to implement a custom data model by subclassing the abstract classes. These examples are available in:

- **<installdir>/jviews-gantt86/samples/databaseGantt**
- **<installdir>/jviews-gantt86/samples/databaseSchedule**

In these examples, the `DBROGanttModel` class implements a Gantt data model that connects to a read-only custom database. The database implements the `GanttDBRO` interface instead of the standard JDBC interface.

The Filter example illustrates how to implement a custom data model by subclassing the `IlvFilterGanttModel` base class. The `BasicFilterGanttModel` implementation wrappers another Gantt data model and filters activities for display. This example is available in:

<installdir>/jviews-gantt86/samples/filter.

Styling

Describes how to use Cascading Style Sheets (CSS) for styling your data.

In this section

Styling examples
Lists the examples that show you how to customize the appearance of a chart by applying cascading style sheets.

Using CSS syntax in the style sheet
Explains the origins and syntax of CSS and how to apply CSS to Java™ objects.

Applying CSS to Java objects
Describes how the CSS selector mechanism is used to match a hierarchy of Java objects accessible from a CSS model interface.

Using style sheets
Explains how to use style sheets in JViews Gantt.

The Gantt and Schedule CSS examples
Describes how to run and customize the CSS examples.

The resource data CSS example
Describes how to run and customize this example application.

Styling Gantt and Schedule chart components
Describes how style sheets can be used to customize the appearance of the Gantt and Schedule chart components and their subcomponents.

© Copyright IBM Corp. 1987, 2009

Styling Gantt chart and Schedule chart data

Describes the activities, classes and constraints used to style Gantt and Schedule chart data.

Styling Resource Data chart components

Describes the CSS properties and Java methods used to control Resource Data chart rendering.

Styling the Resource Data chart data

Explains how to control Resource Data charts rendering using CSS and Java.

Styling examples

JViews Gantt allows you to customize the appearance of a chart by applying cascading style sheets (CSS).

The following CSS examples illustrate how styling works and contain several style sheet examples for the Gantt and Schedule charts, respectively:

- **<installdir>/jviews-gantt86/samples/cssGantt**.
- **<installdir>/jviews-gantt86/samples/cssSchedule**.

The following CSS example illustrates how styling works and contains several style sheet examples for the Resource Data chart:

- **<installdir>/jviews-gantt86/samples/cssResourceData**.

Using CSS syntax in the style sheet

Explains the origins and syntax of CSS and how to apply CSS to Java™ objects.

In this section

Overview
Explains the conformity and divergences between the style sheet syntax and the CSS2 specification.

The origins of CSS
Explains the history of cascading style sheets.

The CSS syntax
Explains the CSS syntax briefly.

Overview

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

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

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

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

Declarations have the form:

```
   propertyName : value ;
```

An example of a style rule is:

```
activity[completion > '0.25']{
        background : red;
          foreground : black;
}
```

This rule makes all activities that are more than 25% complete red with black text.

This section introduces and describes CSS briefly and then explains in more detail the version of CSS used in JViews Gantt. It also shows you the typical uses of CSS for customizing activities, constraints, and resource data series.

The origins of CSS

Cascading style sheets (CSS) are a powerful mechanism for customizing HTML rendering inside a Web browser. The CSS2 specification comes from the World Wide Web Consortium (W3C), and has now reached the status of a W3C recommendation.

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

The CSS syntax

This section gives a shortened presentation of CSS syntax. For a full description of CSS syntax, see *http://www.w3.org/TR/REC-CSS2/*.

Style rule

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

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

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

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

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

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

Selector

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

A selector is composed of one or more minimal building blocks. When two or more minimal building blocks are aggregated into a selector, they may be separated by combinators.

A combinator is a single character the semantics of which are described in the following table. Extra spaces are ignored.

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

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

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

For example, the following line:

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

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

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

Declaration

Declarations are key-value couples. The separator is a colon (:). Each declaration is terminated by a semicolon (;). The key should represent a predefined graphic attribute (foreground, size, font, and so forth) and the value is a literal whose type depends on the key (such as red, 10pt, or serif). All key-value pairs are String. You are recommended to quote values with double quotes " " or single quotes ' 'when the values contain non alphanumeric characters.

Priority

The priority of the rules depends on their relative specificity. Specificity is computed as three numbers, a-b-c (in a number system with a large base).

These numbers represent:

- a is the number of ID building blocks in the selector
- b is the number of classes, pseudo-classes, and attributes
- c is the number of element types

The following table shows the information used in priority order, with the most specific first.

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

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

Priority is used as follows:

1. The declarations of all rules that match the same objects are merged.
2. The priority is applied only if there is a conflict (same key value) within the merged declaration block.

Cascading

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

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

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

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

Import statements must precede the first rule in a style sheet. Priorities of the imported rules are computed as if the rules replace the import statements. The following example shows the import.

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

Inheritance

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

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

Applying CSS to Java objects

Describes how the CSS selector mechanism is used to match a hierarchy of Java objects accessible from a CSS model interface.

In this section

Overview
Explains how CSS is applied to Java objects.

The CSS engine
Describes the functions of the CSS engine at load time and run time.

The CSS data model
Describes the information required for the CSS engine and explains the relationship between CSS and Java objects.

CSS recursion
Describes how to style sheet recursion works using Java.

Constructs
Explains how to use CSS constructs.

Expressions
Explains how to use an expression in the place of a literal.

Custom functions
Describes how to use Java to register a custom function as part of an expression.

Registering custom functions
Explains how to register a custom function before it is used in a style sheet.

Divergences from CSS2
Explains changes to the CSS2 mechanism for specific behavior.

Overview

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

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

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

JViews Gantt uses CSS declarations to create and customize graphic objects and renderers for objects in the Gantt data model and to customize components of the chart itself.

Possible customizations are:

- In the Gantt and Schedule charts, activities in the Gantt data model are matched by the CSS selector mechanism to create and customize the activity and reservation graphic renderers. Constraints in the Gantt data model are matched to create and customize the constraint graphics.
- In the Resource Data chart, resource data series are matched by the CSS selector mechanism to customize their rendering.

The CSS engine

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

- At load time: creating and customizing graphic objects and renderers and customizing the chart itself.
- At run time: customizing the graphic objects and renderers according to changes in the Gantt data model.

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

The CSS data model

The input data model represents the seed of the “CSS for Java” engine.

It provides three important kinds of information to the CSS engine, required to resolve the selectors:

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

  This structure consists of the chart itself and parts of the Gantt data model, such as the tree of activities for a Gantt chart.

- Object type, ID, and tag (or user-defined type), which match element type, ID, and CSS classes.

  IDs and types are strings; CSS classes are words separated by a space character. ID is not required to be unique, although it is wise to assume so.

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

The target object is the graphic object or renderer associated with the model object. In the case of the chart itself, the CSS model object and the target graphic object are the same, that is, the chart. For CSS model objects that are part of the Gantt data model, such as activities, the graphic object is the associated activity renderer. The declarations change property values of the graphic object that corresponds to the matching model object, thereby customizing the graphic appearance given by the rendering.

In the Gantt and Schedule charts, the target object to which the CSS declarations are applied is usually:

- an instance of `IlvActivityRenderer` when styling activities, or

- an instance of `IlvConstraintGraphic` when styling constraints.

In the Resource Data chart, the target object is usually an instance of `IlvResourceDataSet` when styling the data series for a resource.

Object Types and Attribute Matching

The following code sample shows a rule that matches the object of type `activity` with the attribute `completion` greater than or equal to `1` and sets the property `background` of the graphic renderer associated with this object (defined elsewhere) to `green`.

Setting a property value for a class

```
activity[completion>='1'] {background : green;}
```

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

The following code example shows a rule that changes the color of objects that are of CSS type `activity` and CSS class `sales` whenever the model attribute `critical` is set to `true`.

Color change behavior dependent on an attribute value

```
activity.sales[critical = true] {color : "gray"}
```

Object identifiers and CSS classes

The CSS ID of a model object can be checked against the # selector of a rule. For activities and resources, the CSS ID is obtained from the `id` property of the object by calling the `getID()` method. The CSS ID of `IlvGeneralConstraint` is obtained from the user-defined property with the reserved name `id`. Other implementations of constraints have an undefined CSS ID.

CSS classes of a model object are matched against the CSS classes in the rule selector. The classes of an `IlvGeneralActivity` or an `IlvGeneralConstraint` object are given by the user-defined property with the reserved name `tags`. Multiple classes are specified by setting the `tags` property as a space-delimited string of CSS class names.

Other activity and constraint implementations do not support CSS class membership.

CSS classes are not necessarily related to data model semantics; they are devices to add to the pattern-matching capabilities in the style sheet. An object belongs to only one type, but can belong to several CSS classes or none. A check on a CSS class is for its presence or absence. Therefore a CSS class can be seen as an attribute without a value or as a Boolean attribute flagged by its presence or absence.

For example, *Matching CSS classes* shows how to change the color of activities that are both critical and related to the Beta test.

Matching CSS classes

```
activity.critical.betatest {
      background : red ;
}
```

This rule matches all activities with a space-delimited `tags` property that contains the strings `critical` and `betatest`.

Class name

The `class` property is a reserved keyword indicating the class name of the generated graphic object or renderer. The class property must be specified somewhere in the rule hierarchy for every activity and constraint leaf rule. However, the class declaration is applied only when there is a creation request. If the model state is changed, the graphic objects and renderers are customized by applying only new declarations from new matching rules of the style sheet. Therefore, the class declaration is ignored if it is not declared in the subset of rules matched by the change in the model.

For activities, the right side of a class declaration is a class name that will be loaded by the system class loader. It may be:

- An implementation of the `IlvActivityRenderer` interface. For example:

  ```
  activity {
    class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
    thickness : 3;
     background : yellow;
  }
  ```

- An implementation of the `IlvActivityRendererFactory` interface.

- An instance of `IlvGraphic`.

For information and examples, see *Styling activities*.

For constraints, the right side of a class declaration may be:

- An instance of `IlvConstraintGraphic`.
- An implementation of the `IlvConstraintGraphicFactory` interface

For information and examples, see *Styling constraints*.

Pseudo-classes and pseudo-elements

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

Activities support the `selected`, `leaf`, `milestone`, and `parent` pseudo-classes. Constraints support the `selected` pseudo-class.

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

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

Model indirection

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

Setting a property to an attribute value

```
activity {
   class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
   background : powderblue;
   label : '@id';
   toolTipText : @'|"<html><center><b>"+@id+"<br>"+@name+"</b>
                         </center></html>"';
}
```

The `label` property labels the activity bar with the ID of the activity. The tooltip is rendered with HTML formatting and displays the activity name and ID on separate lines bold and centered.

Such indirection is also used in the opposite direction, that is, to retrieve the name of the model attribute that controls a graphic property. This allows user interactions to modify the data model correctly. Two special names, `@id` and `@tags`, represent values of the user-defined properties with the reserved names `id` and `tags`, returned by calls to the method `getProperty (java.lang.String)`.

Resolving URLs

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

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

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

CSS recursion

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

Constructs

@# Construct

Prefix the value with '@#' to create new Beans when required as shown in *Creating a Bean in a declaration*.

Creating a Bean in a declaration

```
activity {
  class :     ilog.views.gantt.graphic.renderer.IlvActivityCompositeRenderer;
renderer[0] : @Subobject#barRenderer;
}

#barRenderer {
   class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
    thickness : 1;
}
```

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

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

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

In *Creating a Bean in a declaration*, an `IlvBasicActivityBar` object is created, with the `thickness` property set to `1`. This new object is assigned to the `renderer[0]` property of the `activity` object, which is an instance of `IlvActivityCompositeRenderer`.

@= and @+ Constructs

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

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

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

The need for these refinements arises from a performance issue. The `@#` operator creates a new object each time a declaration is resolved. Usually a declaration is applied whenever a property changes. Under certain circumstances, the creation of objects may lead to expensive

processing, so JViews Gantt provides an optional mechanism to minimize the creation of objects during property changes.

@| Construct

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

@ Construct

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

The @ construct for preventing a property from being modified

```
activity {
    class : "ilog.views.gantt.graphic.renderer.IlvActivityBar" ;
    bottomMargin : "0.3" ;
}

activity:parent {
    bottomMargin : @ ;
}
```

These two rules say that the `bottomMargin` property value should be set to 0.3, unless the activity has the CSS pseudo-class `parent`. Without the `"@"` capability, the default value of `bottomMargin` would have to be written down in the CSS.

Expressions

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

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

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

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

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

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

`'@|"name is: " + @name'`-> "name is: Bob", if the value of current object attribute `name` is `Bob`. Note the use of quotes to keep the space characters.

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

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

There are some default functions provided by JViews Gantt: `formatDate`, `formatDuration`, and `activityProperty`.

The `formatDate` function formats a `java.util.Date` object, passed as the second argument, into a `String`, with a `SimpleDataFormat` string as the first argument.

The `formatDuration` function formats an `IlvDuration` object, passed as the second argument, into a `String`, with one of the following supported constants as the first argument: `TIME_UNIT_MEDIUM`, `TIME_UNIT_SHORT`, or `LARGEST_UNIT_MEDIUM`.

The `activityProperty` function retrieves the value of a user-defined property from an activity. This function is useful for styling constraint graphics based on the value of the From activity or To activity associated with the constraint.

The following sample shows how the `activityProperty` function provides an additional level of indirection, which allows you to retrieve the `id` property of activities that are themselves the `fromActivity` and `toActivity` properties of the constraint.

Styling Constraint Graphics Based on From or To Activity

```
constraint {
  class : 'ilog.views.gantt.graphic.IlvConstraintGraphic';
  toolTipText : activityProperty(@fromActivity,"id")+"to"+
                        activityProperty(@toActivity,"id");
```

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

Custom functions

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

The following code example shows the signature of the main method.

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

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

♦ The parameter `type` is the expected type of the function, when known. A `null` value is possible. Implementation should take care to return an object of this type; otherwise the conversion will only be performed if it can be (that is, if it is a simple conversion between primitive types or to `String`).

♦ The other parameters are the `model`, `node`, `target`, and `closure` at invocation time; `model` is the current CSS object model, `node` is the current CSS model object being customized, and `target` is the graphic object or renderer being customized. Not all functions need these parameters. (See, for example, *Calling the Custom Function Average*.)

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

The following sample shows an example of a function that computes the average value of its parameters.

Custom function example: average of parameters

```
import ilog.views.util.styling.IlvCSSFunction;

class Average extends IlvCSSFunction {
   //default constructor
   public Average() { }

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

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

   // Returns the average of arguments.
   public Object call(Object[] args, Class type, IlvCSSModel model,
                      Object node, Object target, Object closure) {
      // Assume only double, for the sake of simplicity.
      double result = 0d;
      for (int i=0; i<args.length; i++) {
         if (args[i] != null) {
```

```
            result += Double.parseDouble(args[i].toString());
         }
      }
      result /= args.length;
      return new Double(result);
   }
}
```

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

Calling the Custom Function Average

```
constraint {
    lineWidth : @|avrg(@param1,@param2);
}
```

Registering custom functions

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

To do this:

- Call `registerFunction` in `IlvHierarchyChart` or `IlvScheduleDataChart`.

Divergences from CSS2

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

Cascading

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

Pseudo-classes and pseudo-elements

The pseudo-class construct is fully implemented and used to represent renderer-specific states or GUI items.

The list of predefined pseudo-classes is as follows:

- selected
- parent
- milestone
- leaf

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

Attribute matching

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

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

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

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

Syntax enhancement

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

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

The closing semicolon (`;`) is optional.

Null value

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

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

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

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

Empty string

The null syntax does not allow you to specify an empty string in the style sheet. The following code example shows that you can create an empty string.

Creating an empty string

```
activity {
   toolTipText : @#emptyString ;
}
#emptyString {
    class : 'java.lang.String';
}
```

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

Sharing an empty string

```
activity {
   toolTipText : @=emptyString ;
}
#emptyString {
    class : 'java.lang.String';
}
```

Using style sheets

Explains how to use style sheets in JViews Gantt.

In this section

Applying styles
Explains the classes used to apply a style sheet to a chart and describes how the internal mechanism functions.

Disabling styling
Explains the parameters to pass to specific methods to disable styling.

Applying styles

The `IlvHierarchyChart` class, the common superclass of `IlvGanttChart` and `IlvScheduleChart`, implements the `IlvStylable` interface. The `IlvScheduleDataChart` class, the superclass of `IlvResourceDataChart`, also implements the `IlvStylable` interface. This interface defines several methods that can be used to control styling. The following example shows the typical code involved in applying a style sheet to a chart.

```
try {
  chart.setStyleSheets(new String[]{"simple.css"});
} catch (IlvStylingException x) {
  System.err.println("Cannot load style sheets: " + x.getMessage());
}
```

You can integrate a style sheet generated with the Designer in this way. See Integrating a style sheet into an application in *Using the Designer*.

The following table shows the `IlvHierarchyChart` and `IlvScheduleDataChart` methods that can be used to control styling.

The following table shows the methods for controlling styling.

Where used	Methods

Style Sheets	`getStyleSheet()`
	`setStyleSheet(java.lang.String)`
	`getStyleSheets(int)`
	`setStyleSheets(int, java.lang.String)`
	`getStyleSheets()`
	`setStyleSheets(java.lang.String[])`

Debugging	`getStyleSheetDebugMask()` `setStyleSheetDebugMask(int)`

When style sheets are set on a chart, the initial state of the chart is saved internally. When new style sheets are set or styling is disabled completely, the chart is first restored to its saved state. Then, the new style sheets are interpreted in order to customize the chart. This ensures that when you set new style sheets, they will customize the chart beginning from a known state. This also prevents undesired compound customizations that would result from successively applying multiple sets of style sheets.

As a consequence, you should keep two points in mind when you apply style sheets to a Gantt, Schedule, or Resource Data chart and you use Java™ code to customize a chart by calling its APIs:

- If the Java code customizes the chart *before* you set style sheets, the style sheets may override or suppress the Java customization. When you set new style sheets or disable styling completely, the customization performed by the Java code is restored, because it was saved as part of the state of the chart.

- If the Java code customizes the chart *after* you set style sheets, the Java code may override or suppress customizations performed by the style sheets. When you set new style sheets or disable styling completely, the customization performed by the Java code is lost because it was not saved as part of the state of the chart.

Disabling styling

When you globally disable styling, the chart is told that no styles are specified and it removes any overhead related to styling. Note that this is different from setting an empty style sheet on the chart, since the chart will still try to match CSS rules in this case.

To disable styling:

1. Pass `null` to the `setStyleSheets(java.lang.String[])` method of `IlvHierarchyChart` and `IlvScheduleDataChart`.

The CSS samples are provided with JViews Gantt to show how you can use style sheets with CSS syntax to customize the appearance of your Gantt, Schedule, or Resource Data charts.

The Gantt and Schedule CSS examples

Describes how to run and customize the CSS examples.

In this section

Running the examples
Explains where to find and how to run the Activity Chart and Resource Chart CSS rendering samples.

Scheduling data
Explains the XML data loaded by the example.

Customizing a Gantt chart using a simple style sheet
Explains how to write a CSS stylesheet to customize a Gantt chart.

Running the examples

The Activity Chart and Resource Chart CSS rendering samples are provided with JViews Gantt to show how you can use CSS to customize the appearance of your charts.

The files and source code of the Activity Chart and Resource Chart CSS rendering samples can be found in:

- **`<installdir>/jviews-gantt86/samples/cssGantt`**.
- **`<installdir>/jviews-gantt86/samples/cssSchedule`**.

To run a sample:

1. Ensure that the `Ant` utility is properly configured. If not, read Starting the samples for instructions on how to configure `Ant` for JViews Gantt.
2. Go to the directory where the sample is installed and type:

```
ant run
```

Scheduling data

The Activity Chart and Resource Chart CSS rendering samples display scheduling data that is initially loaded from the XML file:

<installdir>/jviews-gantt86/samples/cssGantt/data/data.xml

This XML scheduling data file defines activities that contain additional user-defined properties. These properties can be used during styling to match against CSS declarations or for display in activity renderers.

The following example shows a part of the data file that defines two activities.

```
<activity id="A-1.1.1" name="Compile customer list" start="6-10-2000 4:53:58"
  end="7-10-2000 4:53:58">
  <property name="type">marketing</property>
  <property name="completion">0.90</property>
</activity>
<activity id="A-1.1.2" name="Contact customers" start="7-10-2000 4:53:58"
  end="9-10-2000 4:53:58">
  <property name="completion">1.0</property>
</activity>
```

Both activities contain a `completion` property that has a numeric value from `0` to `1`. In addition, the first activity contains a property named `type` that has the value "`marketing`". The CSS rendering samples read the XML file and populate the Gantt data model with instances of `IlvGeneralActivity`, `IlvGeneralResource`, `IlvGeneralConstraint`, and `IlvGeneralReservation`. Although you can apply styling to any Gantt data model implementation, you can only reference user-defined properties in your style sheets if your data model objects implement the `IlvUserPropertyHolder` interface. The *general* implementations provided in the `ilog.views.gantt.model.general` package implement this interface.

Note: For information on how to read an `IlvGanttModel` from an XML data file, see *How to read an IlvGanttModel from an SDXL file using serialization*.

Customizing a Gantt chart using a simple style sheet

This section describes the contents of a simple style sheet and how it customizes a Gantt chart.

> **Note**: The following steps are based on the Gantt CSS example found in:
>
> **<installdir>/jviews-gantt86/samples/cssGantt**
>
> They also apply to the Schedule CSS example in:
>
> **<installdir>/jviews-gantt86/samples/cssSchedule**.

To customize a Gantt chart:

1. Create an empty CSS file in the `data` directory of the Gantt CSS sample. The file must have a `.css` extension. For example:

   `<installdir>/samples/cssGantt/data/my-first-stylesheet.css`

2. Run the Activity chart CSS rendering sample (see *Running the examples*) and your new style sheet will appear in the list of available style sheets. The Activity chart CSS rendering sample makes all the style sheets available in the `cssGantt/data` directory. Similarly, the Resource chart CSS rendering sample makes all the style sheets available in the `cssSchedule/data` directory.

3. While the Activity chart CSS rendering sample is running, load the empty CSS file into the text editor of your choice.

4. Every time you edit the CSS file in your text editor, save your changes.

   You can then test the changes you have made by switching to the Activity chart CSS rendering sample and reapplying the style sheet to the chart. Reselect your style sheet from the list of available style sheets.

5. In the CSS file, first specify some properties of the Gantt chart:

```
chart {
  rowHeight: 25;
  ganttSheetToolTipsEnabled: true;
  dividerOpaqueMove: true;
}
```

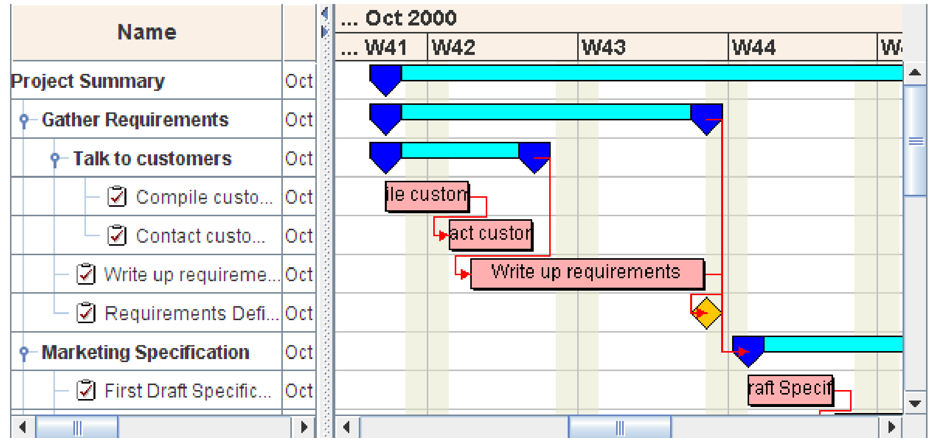
   This increases the default row height of the chart, ensures that tooltips are enabled in the Gantt sheet, and enables `opaqueMove` mode for the vertical divider that separates the table from the sheet.

6. Add some CSS rules that give the table header and the time scale an attractive background color and a bold font:

```
table {
  headerFont: arial,bold,14;
  headerBackground: linen;
```

```
}

timeScale {
  font: arial,bold,14;
  background: linen;
}
```

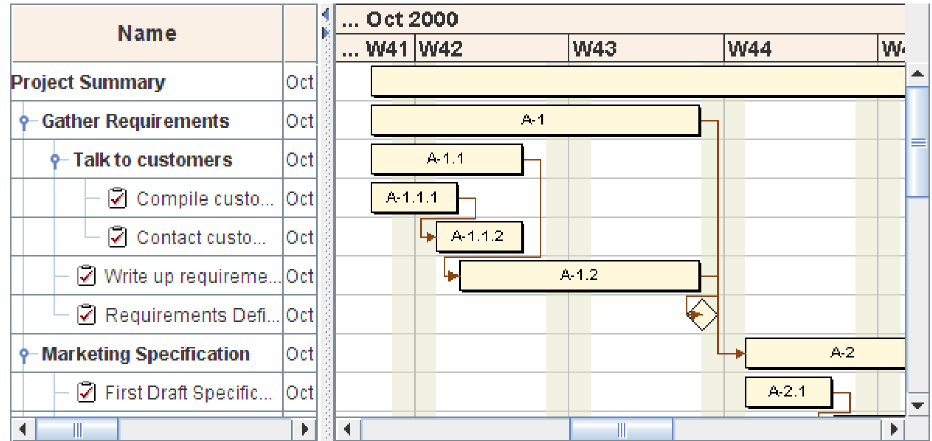
The following figure shows how the Gantt chart looks with this style sheet.



7. Finally, style the activity and constraint graphics by adding additional CSS rules. Specify that all activities are to be displayed as a simple rectangle. The ID of each activity will be displayed in the center of the rectangle in a small font. The color of the constraint links will be changed to a shade of brown that matches well with the rest of the theme.

```
activity {
  class: 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
  background: cornsilk;
  label: "@id";
  font: arial,plain,10;
}

activity:milestone {
  class: 'ilog.views.gantt.graphic.renderer.IlvBasicActivitySymbol';
  shape: DIAMOND;
  background: black;
  foreground: cornsilk;
  label: @
  font: @
}

constraint {
 class: 'ilog.views.gantt.graphic.IlvConstraintGraphic';
 foreground: saddlebrown;
}
```

The following figure shows how the Gantt chart looks now with the completed style sheet.



Note: You can use the Designer to generate a style sheet file that you can load into an application. See Integrating a style sheet into an application in *Using the Designer*.

The resource data CSS example

Describes how to run and customize this example application.

In this section

Running the Example
Explains where you find the source code for the Resource Data CSS sample and how to run it.

Scheduling data
Describes the classes used to handle the in-memory data used for this sample.

Customizing a Resource Data style sheet
Describes the contents of a simple style sheet and explains how it customizes a Resource Data chart.

Two kinds of rules
Explains the two kinds of rules used in the sample style sheet.

Running the Example

The Resource Data CSS sample is provided with JViews Gantt to show how you can use CSS to customize the appearance of your chart. This sample appears as Load Chart Rendering (CSS) in the sample summary page found in:

<installdir>/jviews-gantt86/samples

The files and source code of the Resource Data CSS sample can be found in the directory:

<installdir>/jviews-gantt86/samples/cssResourceData

To run the sample:

1. Ensure that the Ant utility is properly configured. If not, read Starting the samples for instructions on how to configure Ant for JViews Gantt.

2. Go to the directory where the sample is installed and type:

```
ant run
```

Scheduling data

The Resource Data CSS sample uses an in-memory data model that is created by the class:

<installdir>/jviews-gantt86/samples/cssResourceData/src/shared/data/SimpleEngineeringProject.java

The `SimpleEngineeringProject` class implements a Gantt data model that simulates the scheduling of a typical engineering project. The Resource Data CSS sample instantiates the data model with factories that it uses to populate itself with instances of `IlvGeneralActivity`, `IlvGeneralResource`, `IlvGeneralConstraint`, and `IlvGeneralReservation`. As mentioned in *The Gantt and Schedule CSS examples*, although you can apply styling to any Gantt data model implementation, you can only reference user-defined properties in your style sheets if your data model objects implement the `IlvUserPropertyHolder` interface. The general implementations provided in the `ilog.views.gantt.model.general` package implement this interface.

Customizing a Resource Data style sheet

To customize a Resource Data chart:

1. Create an empty CSS file in the data directory of the Resource Data CSS sample. The file must have a .css extension:

   ```
   <installdir>/samples/cssResourceData/data/my-first-stylesheet.css
   ```

2. Run the Resource Data CSS sample (see *Running the Example*) and your new style sheet will appear in the list of available style sheets. The Resource Data CSS sample makes all the style sheets available in the `cssResourceData/data` directory.

3. While the Resource Data CSS sample is running, load the empty CSS file into the text editor of your choice.

4. Every time you edit the CSS file in your text editor, save your changes.

   You can then test the changes you have made by switching to the Resource Data CSS sample and reapplying the style sheet to the chart. Reselect your style sheet from the list of available style sheets.

5. In the CSS file, first specify some properties of the Resource Data chart:

   ```
   chart {
     opaque : true;
     background : lemonchiffon;
     headerText : "My Chart";
   }
   ```

   This sets the background color of the chart and adds a header centered above it. The chart is transparent by default, so you must explicitly set its `opaque` property to `true` in order for the background color to display.

6. Add a CSS rule so that the time scale is transparent and the chart background color shows through:

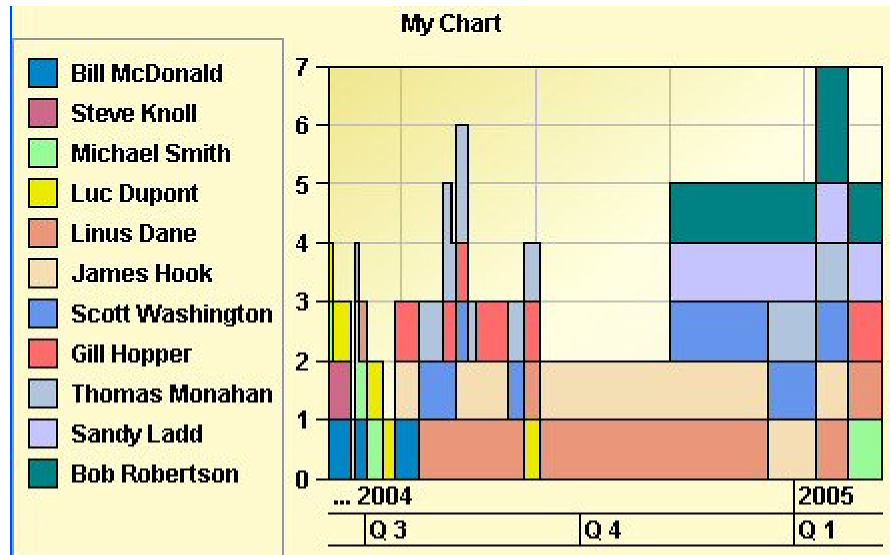
   ```
   #timeScale {
     opaque : false;
   }
   ```

7. Add some CSS rules to give the plot area background a gradient fill and provide some insets surrounding the plot area:

   ```
   chartArea {
     plotStyle : @#plotStyle;
     border : @#emptyBorder;
   }
   Subobject#plotStyle {
     class : 'ilog.views.chart.IlvStyle(strokePaint, fillPaint)';
     strokePaint : black;
     fillPaint : 'khaki\\lightyellow\\linen';
   }
   Subobject#emptyBorder {
     class : 'javax.swing.border.EmptyBorder(borderInsets)';
   ```

```
  borderInsets : 6,6,6,6;
}
```

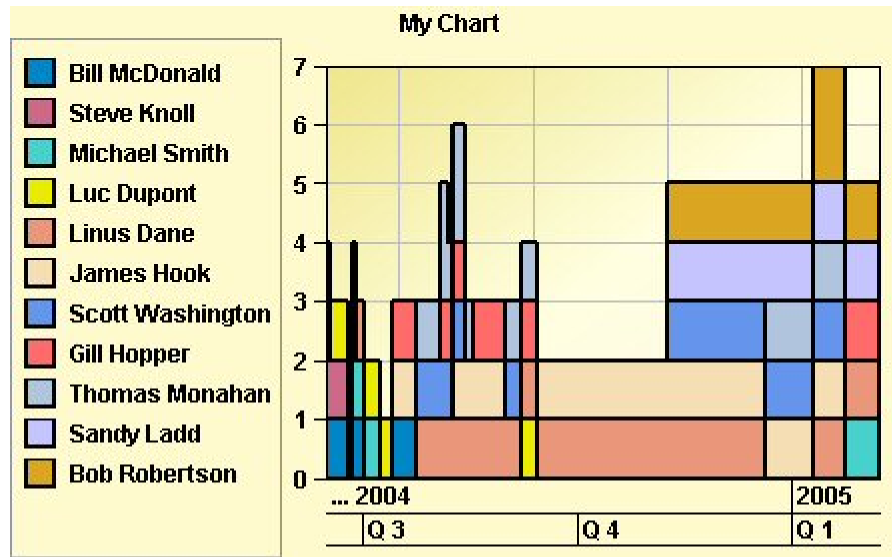
The following figure shows how the Resource Data chart looks with this style sheet.



8. Finally, style the charts data series by writing additional CSS rules:

```
series {
   lineWidth : 2.0;
}
series[name="Bob Robertson"] {
   color1 : "goldenrod" ;
}
series[name="Michael Smith"] {
   color1 : "mediumturquoise" ;
}
```

The following figure shows how the Resource Data chart looks now with the completed style sheet.


My Chart
Bill McDonald
Steve Knoll
Michael Smith
Luc Dupont
Linus Dane
James Hook
Scott Washington
Gill Hopper
Thomas Monahan
Sandy Ladd
Bob Robertson
7
6
5
4
3
2
1
0
... 2004
2005
Q 3
Q 4
Q 1

Two kinds of rules

From the previous style sheet sample, you can distinguish two sets of CSS rules:

- Rules that customize the appearance of the chart and its constituent GUI components. These rules are applied to the properties of the chart and its child components.
  - For `IlvGanttChart` and `IlvScheduleChart` objects, the child components are the table, the time scale, and the Gantt sheet. These rules are described in *Styling Gantt and Schedule chart components*.
  - For the `IlvResourceDataChart` objects, the child components are the chart area, the legend, the scales, and the grids. These rules are described in *Styling Resource Data chart components*.
- Rules that control how Gantt data model entities, such as activities, constraints, and reservations, are rendered in the Gantt sheet of the `IlvGanttChart` and `IlvScheduleChart`, and are rendered as data series of the `IlvResourceDataChart`. These rules are described in *Styling Gantt chart and Schedule chart data*.

Styling Gantt and Schedule chart components

Describes how style sheets can be used to customize the appearance of the Gantt and Schedule chart components and their subcomponents. These chart components are explained in more detail in *The Gantt beans*.

The following table lists the CSS elements that are defined to reference the different parts of the chart components:

The following table shows the Gantt and Schedule chart CSS elements.

CSS Model Object Type and ID	Description	Target object class	Bean properties	Type
chart	The Gantt or Schedule chart component	IlvGanttChart IlvScheduleChart	constraintLayerVisible	boolean
			displayingConstraints	boolean
			dividerLocation	int
			dividerOpaqueMove	boolean
			dividerSize	int
			ganttSheetBackground	Color
			ganttSheetToolTipsEnabled	boolean
			ganttSheetVisible	boolean
			horizontalScrollBarVisible	boolean
			insideBorder	Border
			maxVisibleTime	Date
			minVisibleTime	Date
			mouseWheelEnabled	boolean
			mouseWheelPreferredOrientation	int
			multipleRowActivityGraphicsEnabled	boolean
			rootRowVisible	boolean
			rowHeight	int
			tableBackground	Color
			tableFont	Font
			tableForeground	Color
			tableGridColor	Color
			tableHeaderBackground	Color
			tableHeaderFont	Font
			tableHeaderForeground	Color
			tableVisible	boolean
			timeScale	IlvTimeScale
			timeScaleBackground	Color
			timeScaleFont	Font
			timeScaleForeground	Color
			verticalPosition	int

CSS Model Object Type and ID	Description	Target object class	Bean properties	Type
			verticalScrollBarPolicy	int
			verticalScrollMode	int
			visibleDuration	IlvDuration
			visibleIntervalAnimationSteps	int
			visibleTime	Date
		IlvScheduleChart	activityLayout	IlvActivityLayout
			reservationCacheLoadFactor	float
			reservationCacheLoadThreshosld	float
			reservationCachingEnabled	boolean
sheet	The Gantt sheet	IlvGanttSheet	antialiasing	boolean
			background	Color
			backgroundPatternLocation	URL
			defaultGhostColor	Color
			defaultXORColor	Color
			displayingConstraints	boolean
			horizontalGrid	IlvGanttGridRenderer
			hoverHighlightingMode	int
			multipleRowActivityGraphicsEnabled	boolean
			parentActivityEditable	boolean
			parentActivityMovable	boolean
			refreshMilestoneRenderer	boolean
			refreshParentActivityRenderer	boolean
			toolTipsEnabled	boolean
			verticalGrid	IlvGanttGridRenderer
horizontalGrid	The horizontal grid of the Gantt sheet	IlvHorizontalGanttGrid	evenRowsBackground	Color
			filled	boolean
			foreground	Color
			oddRowsBackground	Color
verticalGrid	The vertical grid of the Gantt sheet	IlvWeekendGrid See the note after this	foreground	Color
			printWeekendsOpaque	boolean

CSS Model Object Type and ID	Description	Target object class	Bean properties	Type
		table.	weekendColor	Color
			weekendDisplayed	boolean
timeScale	The time scale	IlvTimeScale	background	Color
			font	Font
			foreground	Color
			opaque	boolean
table	The Gantt table	IlvJTable	background	Color
			columnMargin	int
			columns	String
			font	Font
			foreground	Color
			gridColor	Color
			headerBackground	Color
			headerFont	Font
			headerForeground	Color
			showsRootHandles	boolean

Note: By default, the `verticalGrid` CSS model object type implements the `IlvWeekendGrid` subclass of `IlvVerticalGanttGrid`. If you replace `IlvWeekendGrid` with your own subclass, the CSS will return the properties of your own subclass.

Similarly, the `horizontalGrid` CSS model object type implements the class `IlvHorizontalGanttGrid`. If you replace `IlvHorizontalGanttGrid` with your own implementation, the CSS will return the properties of your class.

These CSS model objects can be used to modify the Bean properties of the corresponding target object. The following example shows how you can control the row height of the chart and the colors and fonts of the table and the time scale.

```
chart {
  rowHeight: 25;
}

table {
  headerFont: arial,bold,14;
```

```
  headerBackground: linen;
}

timeScale {
  font: arial,bold,14;
  background: linen;
}
```

The CSS ID is the same as the CSS model object type for each of the chart components. Therefore, the following CSS rules are equivalent to the ones above. Here, the ID of each chart component, instead of its type, is specified as the selector for each rule:

```
#chart {
  rowHeight: 25;
}

#table {
  headerFont: arial,bold,14;
  headerBackground: linen;
}

#timeScale {
  font: arial,bold,14;
  background: linen;
}
```

Note: The chart components have no assigned CSS classes or pseudo-classes.

Styling Gantt chart and Schedule chart data

Describes the activities, classes and constraints used to style Gantt and Schedule chart data.

In this section

Overview
Explains how to use style sheets to specify the rendering attributes of activities and constraints in the Gantt sheet.

Styling activities
Explains in detail the model objects used to identify activities in the Gantt data model and how to use them.

Styling constraints
Describes the model object type identifier constraints in the Gantt data model that will be styled by the CSS engine.

Overview

Style sheets can also be used to specify the rendering attributes of activities and constraints in the Gantt sheet. The selector for each CSS rule specifies which activities or constraints are being rendered in the Gantt data model. The target object to which the CSS declarations are applied is usually an instance of `IlvActivityRenderer` or an instance of `IlvConstraintGraphic` respectively. This is explained in more detail in the following sections.

Styling Gantt data works best if you use data model implementation classes that implement the `IlvUserPropertyHolder` interface such as the general data model implementation classes, provided in the `ilog.views.gantt.model.general` (see *Default data model implementation*). These classes support user-defined properties. This allows the CSS engine to match properties of the data model objects against CSS attribute selectors and to perform model indirection when evaluating the CSS declarations. For information, see *Selector* and *Model indirection*.

Note: If you do not use the `IlvActivity` and `IlvConstraint` data model implementations that implement the `IlvUserPropertyHolder` interface, you will not be able to use attribute selectors or perform model indirection in your style sheets.

Styling activities

Explains in detail the model objects used to identify activities in the Gantt data model and how to use them.

In this section

Activity model objects
Describes the object types used for styling.

Activity renderer target objects
Describes how to specify the required constructor arguments in the CSS declaration.

Activity ID selectors
Describes the things to look for when you use ID selectors in your style sheet.

IlvGeneralActivity properties
Describes the styling features available when you use an `IlvGeneralActivity` instance.

IlvGeneralActivity CSS classes
Describes how activity properties are used to list the CSS classes and object it belongs to.

Activity CSS pseudoclasses
Describes the activity pseudoclasses you can use in rule selectors.

The formatDate and formatDuration functions
Explains the predefined date and duration formatting functions you can use as part of an expression in your style sheet.

Activity model objects

The `activity` model object type identifies activities in the Gantt data model that will be styled by the CSS engine. The target object to which the CSS declarations will be applied can be an instance of `IlvActivityRenderer`, `IlvActivityRendererFactory`, or `IlvGraphic`. The class of the target object must always be specified and is declared in the style sheet using the reserved class property name. This is explained in more detail in *Class name*. The following extremely simple CSS rule will display all activities using an `IlvBasicActivityBar` renderer.

```
activity {
  class: 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
}
```

As shown previously, you can then add more declarations to the CSS rule that specify Bean properties of the `IlvBasicActivityBar` target object you want to customize:

```
activity {
  class      : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
  thickness  : 3;
  background : yellow;
}
```

The following table summarizes the CSS model objects, tokens, and functions that are applicable for styling activities. Each item of the table is further discussed in the subsequent sections.

Model object	An instance of `IlvActivity`. An `IlvGeneralActivity` provides the most flexibility.
Model indirection	Properties of `IlvGeneralActivity`, or of `IlvActivity` implementations that implement the `IlvUserPropertyHolder` interface. Not supported for other `IlvActivity` implementations.
Target object class	`IlvActivityRenderer` or `IlvActivityRendererFactory` or `IlvGraphic`.
CSS model object type	activity
CSS ID	The ID property of the activity: `getID()` or

	getProperty(java.lang.String).
CSS declaration properties	Bean properties of the target object.
CSS classes	The tags property of IlvGeneralActivity, or of IlvActivity implementations that implement the IlvUserPropertyHolder interface: IlvGeneralActivity.getProperty("tags") Not supported for other IlvActivity implementations.
CSS pseudoclasses	parent leaf milestone selected
CSS attribute selectors	Properties of IlvGeneralActivity, or of IlvActivity implementations that implement the IlvUserPropertyHolder interface. Not supported for other IlvActivity implementations.
CSS custom functions	formatDate() formatDuration()

Activity renderer target objects

As shown in *Styling activities*, the target object to which the CSS declarations are applied can be an instance of `IlvActivityRenderer`, `IlvActivityRendererFactory`, or `IlvGraphic`. If you specify an `IlvGraphic` class, it will be instantiated and then used in an `IlvActivityGraphicRenderer` wrapper by the Gantt CSS engine. Most `IlvGraphic` implementations provided in the JViews Gantt distribution have constructors with no arguments. However, for those `IlvGraphic` implementations that have no such zero-argument constructors, you need to specify the required constructor arguments in the CSS declaration. The following example shows how to specify a filled `IlvRectangle` graphic as an activity renderer.

Note: The example below is in fact purely didactic and does not really apply to the class `IlvRectangle` since this class does have a constructor with no argument.

```
activity {
  class          : 'ilog.views.graphic.IlvRectangle(definitionRect)';
  definitionRect : @=dummyRect;
  fillOn         : true;
  background     : lightseagreen;
}

Subobject#dummyRect {
  class : ilog.views.IlvRect;
}
```

Notice how a dummy `IlvRect` object is provided as an argument to the `IlvRectangle` constructor. The initial value of this rectangle is unimportant because the Gantt library will subsequently resize the graphic to represent the time duration of the activity.

If you specify an `IlvActivityRendererFactory` instance as your target object, the Gantt CSS engine will ask the factory to create the activity renderer. However, you are recommended not to use the renderer factories that are provided in the distribution because they are not well suited to CSS styling. This is because the provided factories create renderer instances that are shared among activities. In an application that does not use CSS styling, this minimizes object creation and memory usage. However, this also defeats the ability of the Gantt CSS engine to apply individualized rendering Customization. If you have written your own activity renderer factory that does not share renderer instances, then it should work well with CSS styling.

In most cases, you will simply specify an `IlvActivityRenderer` implementation as your target object. The following table lists the renderers provided in the distribution that provide the most flexibility when used with CSS styling:

The following table shows the renderers for CSS styling of activities.

Renderer	Bean properties	Type
IlvBasicActivityBar	background	Color
	bottomMargin	float
	font	Font
	foreground	Color
	label	String
	style	enum
	thickness	int
	toolTipText	String
	topMargin	float
IlvBasicActivityLabel	background	Color
	bottomMargin	float
	font	Font
	foreground	Color
	horizontalAlignment	enum
	label	String
	offset	float
	toolTipText	String
	topMargin	float
	verticalAlignment	enum
IlvBasicActivitySymbol	alignment	enum
	background	Color
	bottomMargin	float
	foreground	Color
	shape	enum
	toolTipText	String
	topMargin	float
IlvActivityCompositeRenderer	renderer	IlvActivityRenderer

Of course, other renderers provided in the distribution can also be specified in the style sheet, as well as any custom activity renderers that you may have written yourself.

The following code example shows how to use the class IlvActivityCompositeRenderer to create a more complex renderer from simpler ones:

Creating complex renderers

```
activity {
  class   : 'ilog.views.gantt.graphic.renderer.IlvActivityCompositeRenderer';

  renderer[0] : @#bar;
  renderer[1] : @#startSymbol;
  renderer[2] : @#endSymbol;
}

Subobject#bar {
  class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
  background   : powderblue;
  bottomMargin : 0.3;
}

Subobject#startSymbol {
  class     : 'ilog.views.gantt.graphic.renderer.IlvBasicActivitySymbol';
  alignment : START;
}

Subobject#endSymbol {
  class     : 'ilog.views.gantt.graphic.renderer.IlvBasicActivitySymbol';
  alignment : END;
}
```

Activity ID selectors

As shown in *Styling activities*, the ID property of activities in the Gantt data model can be used as CSS ID selectors. For example, if your data model has an activity with an ID of “A7345”, you could specify a rule that customizes the rendering of that specific activity like this:

```
#A7345 {
  class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
  background : orange;
  label : 'I am a special activity';
}
```

There are several things you should be cautious of when you use ID selectors in your style sheet:

- ♦ Each activity should have an ID that is unique across all objects of the data model.
- ♦ Activity ID selectors can only be specified in the style sheet using alphanumeric characters. The activities defined in section *Scheduling data* have IDs that contain non-alphanumeric characters, such as the hyphen. Therefore, this data model is not suitable for use with CSS ID selectors.
- ♦ For performance reasons, the Gantt CSS engine assumes that CSS model object IDs are immutable. Therefore, if the ID of an activity in your data model changes, the Gantt CSS engine will not automatically re-interpret the ID selector rules. Although ILOG does not recommend that you create a data model implementation where the IDs of data objects change dynamically, you can overcome this limitation by reapplying the style sheet and thereby forcing its complete re-interpretation.

IlvGeneralActivity properties

If your Gantt data model uses the `IlvGeneralActivity` implementation, or `IlvActivity` implementations that implement the `IlvUserPropertyHolderIlvUserProperty` interface, you will have the most flexibility when you write CSS declarations to style activities. `IlvGeneralActivity` allows you to specify predefined and user-defined activity properties as CSS attribute selectors. It also allows you to perform model indirection in the value part of your CSS declarations. The samples provided, described in *The Gantt and Schedule CSS examples*, populate their data model with `IlvGeneralActivity` instances. You can therefore use these samples to test and experiment with the styling features described in this section.

IlvGeneralActivity CSS classes

The Gantt CSS engine interprets the “tags” property of an `IlvGeneralActivity`, or another `IlvActivity` implementation that implements the `IlvUserPropertyHolder` interface, as the space-separated list of the CSS classes it belongs to. For example, the default data model of the samples provided defines a `tags` value of “`critical`” for some of the activities:

```
<activity id="A-1.3" name="Requirements Defined" start="21-10-2000 0:0:0"
  end="21-10-2000 0:0:0">
  <property name="tags">critical</property>
</activity>
```

You can then specify the following rules that will highlight all activities that are members of the “critical” class in a different color:

```
activity {
  class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
  background : powderblue;
  label      : '@id';
}

activity.critical {
  background : plum;
}
```

Activity CSS pseudoclasses

As shown in *Styling activities*, the Gantt CSS engine defines several activity pseudoclasses that you can use in your rule selectors.

The pseudoclasses are:

- parent - Indicates that the activity has at least 1 child activity.
- leaf - The opposite of parent, indicates that the activity has no children.
- milestone - Indicates that the activity has zero duration.
- selected - Indicates that the activity is selected.

Most of the previous CSS examples given so far use an `IlvBasicActivityBar` renderer for all activities. You may have already noticed that this renderer becomes nearly invisible when it attempts to render a milestone activity that has zero duration. The following rules illustrate how you can provide a symbol renderer for these activities by using the `milestone` pseudoclass in the selector:

```
activity {
  class : 'ilog.views.gantt.graphic.renderer.IlvBasicActivityBar';
  background : powderblue;
  label      : '@id';
}

activity:milestone {
  class  : 'ilog.views.gantt.graphic.renderer.IlvBasicActivitySymbol';
  shape  : DIAMOND;
  foreground : yellow;
  label  : @;
}
```

Note: The `label` line uses the special @ value to ignore the label property declaration that the milestone rule has inherited.

The formatDate and formatDuration functions

As shown in *Styling activities*, the Gantt CSS engine provides two predefined functions you can use as part of an expression in your style sheet: `formatDate` and `formatDuration`.

formatDate

The `formatDate` function lets you format a `Date` property value using a standard pattern string defined by the `java.text.SimpleDateFormat` class. The syntax of this function is.

```
formatDate(<SimpleDateFormat pattern>, <Date>)
```

The following example shows a declaration used to set the tooltip to the formatted start time of the activity:

```
toolTipText : '@|"Start: " + formatDate("MM/dd/yy",@startTime)';
```

formatDuration

Similarly, the `formatDuration` function lets you format an `IlvDuration` value using an `IlvDurationFormat` constant. The syntax of this function is:

```
formatDuration(<IlvDurationFormat constant>, <IlvDuration>)
```

The following example shows a declaration used to set the tooltip to the formatted duration of the activity:

```
toolTipText: '@|"Duration: " + formatDuration(LARGEST_UNIT_MEDIUM, @duration)
';
```

You can see more complex usage of these functions by examining the `standard-look.css` style sheet that is provided in:

<installdir>/jviews-gantt86/samples/cssGantt/data/standard-look.css

Styling constraints

Describes the model object type identifier constraints in the Gantt data model that will be styled by the CSS engine.

In this section

Constraint model objects
Explains in detail the model objects used to identify constraints in the Gantt data model and how to use them.

Constraint graphic target objects
Describes the bean properties of `IlvConstraintGraphic` that can be customized with CSS styling.

Constraint ID selector
Explains how the id property can be interpreted as the CSS ID attribute and used in ID selectors.

IlvGeneralConstraint properties
Explains how `IlvGeneralConstraint` can be used to specify predefined and user-defined constraint properties as CSS attribute selectors.

IlvGeneralConstraint CSS classes
Explains how the `tags` property is interpreted by the Gantt CSS engine.

Constraint CSS pseudoclasses
Explains how pseudoclass are defined so that you can use them in rule selectors.

The activityProperty function

Describes the predefined functions that you can use as part of an expression in your style sheet.

Constraint model objects

The `constraint` model object type identifies constraints in the Gantt data model that will be styled by the CSS engine. The target object to which the CSS declarations will be applied can be an instance of `IlvConstraintGraphic` or `IlvConstraintGraphicFactory`. The class of the target object must always be specified and is declared in the style sheet using the reserved `class` property name. The following example shows an extremely simple CSS rule that will render all constraints using the standard `IlvConstraintGraphic` class.

```
constraint {
  class: 'ilog.views.gantt.graphic.IlvConstraintGraphic';
}
```

You can then add additional declarations to the CSS rule that specify Bean properties of the `IlvConstraintGraphic` target object that you want to customize:

```
constraint {
  class      : 'ilog.views.gantt.graphic.IlvConstraintGraphic';
  foreground : green;
  lineWidth  : 2;
}
```

The following table summarizes the CSS model object types, tokens, and functions that are applicable when styling constraints. Each item of the table is further discussed in the subsequent sections.

Model object	An instance of `IlvConstraint`. An `IlvGeneralConstraint` provides the most flexibility.
Model indirection	Properties of `IlvGeneralConstraint`, or of `IlvConstraint` implementations that implement the `IlvUserPropertyHolder` interface. Not supported for other `IlvConstraint` implementations.
Target object class	`IlvConstraintGraphic` or `IlvConstraintGraphicFactory`
CSS model object type	constraint
CSS ID	The ID property of IlvGeneralConstraint: `getProperty(java.lang.String)`. Not supported for other `IlvConstraint` implementations.
CSS declaration properties	Bean properties of the target object.
CSS classes	The tags property of `IlvGeneralConstraint`, or of `IlvConstraint` implementations that implement the `IlvUserPropertyHolder` interface: `IlvGeneralConstraint.getProperty("tags")`.

	Not supported for other `IlvConstraint` implementations.
CSS pseudo classes	selected
CSS attribute selectors	Properties of `IlvGeneralConstraint`, or of `IlvConstraint` implementations that implement the `IlvUserPropertyHolder` interface. Not supported for other `IlvConstraint` implementations.

CSS custom functions	`activityProperty()`
	
	`formatDate()`
	
	`formatDuration()`

Constraint graphic target objects

As shown in *Constraint model objects*, the target object to which the CSS declarations will be applied can be an instance of `IlvConstraintGraphic` or `IlvConstraintGraphicFactory`. If you specify an `IlvConstraintGraphicFactory` as your target object, the Gantt CSS engine will ask the factory to create the constraint graphic. In most cases, you will simply specify an `IlvConstraintGraphic` as your target object. The following table lists the bean properties of `IlvConstraintGraphic` that can be customized with CSS styling:

The following table shows the Bean properties for constraint graphics.

Bean properties	Type	Allowed values
arrowSize	`float`	
connectionType	`enum`	`TIME_INTERVAL_CONNECTION` `BOUNDING_BOX_CONNECTION`

endCap	`int`	`ilog.views.IlvStroke.CAP_BUTT`
		`ilog.views.IlvStroke.CAP_ROUND`
		`ilog.views.IlvStroke.CAP_SQUARE`

foreground	`Color`	
horizontalExtremitySegmentLength	`float`	

lineJoin	`int`	`ilog.views.IlvStroke.JOIN_BEVEL`
		`ilog.views.IlvStroke.JOIN_MITER`
		`ilog.views.IlvStroke.JOIN _ROUND`

lineStyle	`float`	
lineWidth	`float`	
oriented	`boolean`	
toolTipText	`String`	

Constraint ID selector

Constraints in the Gantt data model do not define an ID property as part of the basic `IlvConstraint` interface. However, if you are using the `IlvGeneralConstraint` implementation, or an `IlvConstraint` implementation implementing the `IlvUserPropertyHolder` interface, the `id` property will be interpreted as the CSS ID attribute and can be used in ID selectors. For example, if your data model defines the following constraint in its XML data file:

```
<constraint from="A723" to="A39" type="End-Start">
  <property name="id">C86</property>
</constraint>
```

You could then specify a rule that customizes the rendering of that specific constraint like this:

```
#C86 {
  class       : 'ilog.views.gantt.graphic.IlvConstraintGraphic';
  foreground  : magenta;
  toolTipText : 'I am a special constraint';
}
```

Note: The same limitations discussed in *Activity ID selectors* apply to constraint ID selectors.

IlvGeneralConstraint properties

If your Gantt data model uses the `IlvGeneralConstraint` implementation, or an `IlvConstraint` implementation implementing the `IlvUserPropertyHolder` interface, you will have the most flexibility when you write CSS declarations to style constraints. `IlvGeneralConstraint` allows you to specify predefined and user-defined constraint properties as CSS attribute selectors. It also allows you to perform model indirection in the value part of your CSS declarations. The samples provided, described in *The Gantt and Schedule CSS examples*, populate their data model with `IlvGeneralConstraint` instances. You can therefore use these samples to test and experiment with the styling features described in this section.

IlvGeneralConstraint CSS classes

The Gantt CSS engine interprets the `tags` property of an `IlvGeneralConstraint`, or another `IlvConstraint` implementation that implements the `IlvUserPropertyHolder` interface, as the space-separated list of the CSS classes it belongs to. This is identical in concept to *IlvGeneralActivity CSS classes*. For example, let us set the `tags` property of a constraint in our data model so that the constraint belongs to the `delay` and `critical` classes:

```
anIlvGeneralConstraint.setProperty("tags", "delay critical");
```

You can then add a rule to the style sheet that highlights all constraints that are both delayed and that are critical in a different color:

```
constraint.critical.delay {
  foreground : red;
  lineWidth : 5;
}
```

Constraint CSS pseudoclasses

As shown in *Constraint model objects*, the Gantt CSS engine defines the `selected` pseudoclass that you can use in your rule selectors. The following example shows some rules that increase the width of the constraint graphic to indicate when it is selected.

```
constraint {
  class      : 'ilog.views.gantt.graphic.IlvConstraintGraphic';
  foreground : green;
  lineWidth  : 1;
}

constraint:selected {
  lineWidth  : 3;
}
```

The activityProperty function

As shown in *Constraint model objects*, the Gantt CSS engine provides three predefined functions that you can use as part of an expression in your style sheet. Section *The formatDate and formatDuration functions* already discusses two of the functions. The `activityProperty` function lets you refer to a property of the constraint `from` activity or `to` activity. The syntax of this function is:

```
activityProperty(<IlvGeneralActivity>, <property name>)
```

The following example shows a declaration used to set the tooltip of the constraint graphic to contain the names of the constraint's `from` and `to` activities.

```
toolTipText : '@|"<html>From: "+activityProperty(@fromActivity,"name")+"<br>To:
                              "+activityProperty(@toActivity,"name")+"</html>"';
```

You can see more complex usage of this function by examining the `standard-look.css` style sheet that is provided in the `<Installdir>samples/gantt/css/data` directory.

Styling Resource Data chart components

Describes the CSS properties and Java methods used to control Resource Data chart rendering.

In this section

Overview
Describes the Resource Data chart component and its subcomponents.

Styling the Chart Area component
Explains how to use the chartArea model object to control the appearance of a portion of a chart.

Styling the Chart Legend
Explains how the chartLegend model object type to control the appearance of the Chart Legend.

Styling the chart renderer
Describes how to use the chartRenderer CSS model object to control the global appearance of the renderer used by the chart.

Styling the chart scales
Explains how to reference a specific scale in a CSS rule.

Styling the chart grids
Explains how to reference a specific grid in a CSS rule.

Overview

You can also use style sheets to customize the appearance of the Resource Data chart component and its subcomponents. These chart components are explained in more detail in *The Resource Data chart bean*. The following table lists the CSS model object types that are defined to reference the different parts of the chart component.

CSS Model Object Type	CSS ID	Description	Target Object Class
chart	Chart	The Resource Data chart component	`IlvResourceDataChart`
chartArea	chartArea	The chart area component	`IlvChart.Area`
chartLegend	chartLegend	The chart legend	`IlvLegend`
chartRenderer	chartRenderer	The chart renderers	`IlvChartRenderer`
chartScale	timeScale yScale	The chart scales	`IlvTimeScale` `IlvScale`
chartGrid	xGrid yGrid	The chart grids	`IlvGanttGridRenderer` `IlvGrid`

Note: Chart components have no assigned CSS classes or pseudoclasses

The `chart` model object type identifies the Resource Data chart component and can be used to control the global appearance of the chart. The following table lists the Bean properties of the `IlvResourceDataChart` class that can be set in the declarations of a CSS style rule.

Bean properties	Type
antiAliasing	boolean
antiAliasingText	boolean
background	Color
border	Border
chartAreaBorder	Border
dataRangePolicy	IlvDataRangePolicy
decorations	List
defaultColors	Color[]
footer	JComponent
footerText	String
foreground	Color
header	JComponent
headerText	String
interactors	IlvChartInteractor
legend	IlvLegend
legendPosition	String
legendVisible	boolean
maxVisibleTime	Date
minVisibleTime	Date
plotAreaBackground	Color
renderer	IlvChartRenderer
resourceDisplayMode	int
scalingFont	boolean
timeScale	IlvTimeScale
visibleDuration	IlvDuration
visibleTime	Date
xGrid	IlvGanttGridRenderer
yAxisReversed	boolean
yGrid	IlvGrid
yGridVisible	boolean
yScale	IlvScale

Bean properties	Type
yScaleTitle	`String`
yScaleTitleRotation	`double`
yScaleVisible	`boolean`

For example, the following CSS rules show you how to control the header, the borders, and the colors of the chart:

```
chart {
  opaque : true;
  foreground : black;
  background : lightyellow;
  plotAreaBackground : slateblue;
  border : @#chartBorder;
  header : @#header;
}

Subobject#chartBorder {
  class : 'javax.swing.border.LineBorder(lineColor)';
  lineColor : black;
}
Subobject#header {
  class : 'javax.swing.JLabel';
  text : "My Resource Chart";
}
```

The CSS ID of the Chart Component is the same as its CSS model object type. Therefore, the following CSS rule is equivalent to the first rule above. Here, the ID of the chart component, instead of its type, is specified as the selector for the rule:

```
#chart {
  opaque : true;
  foreground : black;
  background : lightyellow;
  plotAreaBackground : slateblue;
  border : @#chartBorder;
  header : @#header;
}
```

Styling the Chart Area component

The `chartArea` model object type identifies the Chart Area component and can be used to control the appearance of the portion of the chart.

The following table lists the Bean properties of the `IlvChart.Area` class that can be set in the declarations of a CSS style rule.

Bean properties	Type
background	`Color`
backgroundPaint	`Paint`
border	`Border`
bottomMargin	`int`
filledPlottingArea	`boolean`
foreground	`Color`
font	`Font`
leftMargin	`int`
opaque	`boolean`
plotBackground	`Paint`
plotStyle	`IlvStyle`
rightMargin	`int`
topMargin	`int`

For example, the following CSS rules show you how to control the borders of the chart area:

```
chartArea {
  border : @#emptyBorder;
}
Subobject#emptyBorder {
  class : 'javax.swing.border.EmptyBorder(borderInsets)';
  borderInsets : 6,6,6,6;
}
```

The CSS ID of the Chart Area component is the same as its CSS model object type. Therefore, the following CSS rule is equivalent to the first rule above. Here, the ID of the chart area component, instead of its type, is specified as the selector for the rule:

```
#chartArea {
  border : @#emptyBorder;
}
```

Styling the Chart Legend

The `chartLegend` model object type identifies the Chart Legend and can be used to control its appearance. The following table lists the Bean properties of the `IlvLegend` class that can be set in the declarations of a CSS style rule.

Bean properties	Type
antiAliasing	`boolean`
antiAliasingText	`boolean`
border	`Border`
background	`Color`
floating	`boolean`
floatingLayoutDirection	`int`
followChartResize	`boolean`
font	`Font`
foreground	`Color`
interactive	`boolean`
location	`Point`
movable	`boolean`
paintingBackground	`boolean`
symbolSize	`Dimension`
symbolTextSpacing	`int`
title	`String`
transparency	`int`

For example, the following CSS rules show you how to control the text and symbol rendering, and the interactive docking capability of the chart legend:

```
chartLegend {
  antiAliasing : true;
  antiAliasingText : true;
  font : 'Arial,plain,10';
  symbolSize : "12,12";
  movable: true;
}
```

The CSS ID of the Chart Legend is the same as its CSS model object type. Therefore, the following CSS rule is equivalent to the rule above. Here, the ID of the chart legend, instead of its type, is specified as the selector for the rule:

```
#chartLegend {
  antiAliasing : true;
  antiAliasingText : true;
  font : 'Arial,plain,10';
  symbolSize : "12,12";
  movable: true;
}
```

Styling the chart renderer

The chartRenderer CSS model object type can be used to control the global appearance of the renderer used by the chart. The Resource Data chart supports a single renderer at index 0. The chartRenderer model object type defines an index attribute with value 0 that allows you to select the renderer for styling. The default renderer of the chart is an instance of IlvStairChartRenderer. For example, the following rule changes the mode of the renderer:

```
chartRenderer[index=0] {
  autoTransparency : true;
  mode : SUPERIMPOSED;
}
```

Styling the chart scales

There are two ways to reference a specific scale in a CSS rule. You can use:

- the `axisIndex` attribute of the `chartScale` CSS model object type. This index equals -1 for the x-axis, 0 for the y-axis.
- the `timeScale` CSS ID for the x-axis and the `yScale` CSS ID for the y-axis.

X-axis scale

The default x-axis scale is an instance of the class `IlvGanttTimeScale`. It can be selected using the `chartScale[axisIndex="-1"]` model object type selector or the `#timeScale` ID selector. The Bean properties of the class `IlvTimeScale` that can be set in the declarations of a CSS style rule are discussed in *Styling Gantt and Schedule chart components*.

Note: If you replace `IlvGanttTimeScale` with your own `IlvTimeScale` subclass, the CSS will return the properties of your own subclass.

Y-axis scale

The y-axis scale is an instance of the class `IlvScale`. It can be selected using the `chartScale [index="0"]` model object type selector or the `#yScale` ID selector. The following table lists the Bean properties of the `IlvScale` class that can be set in the declarations of a CSS style rule.

Property	Type	Allowed values
annotations	IlvScaleAnnotation	
autoCrossing	boolean	
autoWrapping	boolean	
axisStroke	Stroke	
axisVisible	boolean	
category	boolean	
crossingValue	double	
drawOrder	int	
foreground	Color	
labelAlignment	int	LEFT, CENTER, RIGHT
labelColor	Color	
labelFont	Font	
labelOffset	int	
labelRotation	float	
labelVisible	boolean	
majorTickSize	int	
majorTickVisible	boolean	
minorTickSize	int	
minorTickVisible	boolean	
skipLabelMode	int	CONSTANT_SKIP,ADAPTIVE_SKIP
skippingLabel	boolean	
stepsDefinition	IlvStepsDefinition	
tickLayout	int	TICK_INSIDE, TICK_OUTSIDE,TICK_CROSS
title	String	
titleOffset	int	
titlePlacement	int	
titleRotation	float	
visible	boolean	

Styling the chart grids

There are two ways to reference a specific grid in a CSS rule. You can use:

- the `axisIndex` attribute of the `chartGrid` CSS model object type. This index equals -1 for the x-axis, 0 for the y-axis.
- the `xGrid` and `yGrid` CSS IDs.

X-axis grid

The default x-axis grid is an instance of `IlvWeekendGrid`. It can be selected using the `chartGrid[axisIndex="-1"]` model object type selector or the `#xGrid` ID selector. The Bean properties of the `IlvWeekendGrid` class that can be set in the declarations of a CSS style rule are discussed in *Styling Gantt and Schedule chart components*.

> **Note**: If you replace `IlvWeekendGrid` with your own `IlvGanttGridRenderer` implementation, the CSS will return the properties of your own subclass.

Y-axis grid

The y-axis grid is an instance of `IlvGrid`. It can be selected using the `chartGrid[index="0"]` model object type selector or the `#yGrid` ID selector.

The following table lists the Bean properties of the `IlvGrid` class that can be set in the declarations of a CSS style rule:

Property	Type
drawOrder	`int`
majorLineVisible	`boolean`
visible	`boolean`
minorLineVisible	`boolean`
majorStroke	`Stroke`
majorPaint	`Paint`
minorStroke	`minorPaint`

Styling the Resource Data chart data

Explains how to control Resource Data charts rendering using CSS and Java.

In this section

Overview
Describes how each resource in a Resource Data chart is represented and rendered.

Selector patterns
Describes the objects that are defined to reference the Resource Data chart data model.

Attributes of model objects
Describes attributes defined for the series and point model object types.

CSS classes
Describes how properties are used to identify the CSS class an object belongs to.

Properties
Explains the properties you can use to customize the rendering of data series and data points.

Properties for data series
Describes how to modify the Bean properties of a renderer that displays the corresponding data set.

Overview

Each resource displayed by the Resource Data chart is represented by a data series that consists of individual data points. Style sheets can be used to specify the rendering attributes of the whole data series or single data points. This section explains the expected selector patterns for the CSS rules, in *Selector patterns*, and the Bean properties that can be used in the declarations of these rules, in *Attributes of model objects*.

Selector patterns

Two CSS model object types are defined to reference the data model of Resource Data charts:

- series: used to match the whole series (represented by `IlvResourceDataSet` instances in the data model).
- point: used to match individual data points. The `point` model objects are direct descendants of the `series` model objects.

Attributes of model objects

This section presents and discusses the attributes defined for the `series` and `point` model object types.

Series model object type

The following table lists the attributes defined for the `series` model object type:

Bean property	Type	Description
name	`String`	The name of the data set, as returned by the `getName()` method. This will be the name of the resource that the data set represents.
index	`int`	The index of the data set

For example, the following CSS rule sets a gradient fill on the data series for the resource named "Linus Dane":

```
series[name="Linus Dane"] {
  color1: paleturquoise|darkblue;
}
```

If the resource represented by the data series implements the `IlvUserPropertyHolder` interface, such as the class `IlvGeneralResource`, the resource properties can be accessed as additional properties of the data series. Assuming that the ID of the Linus Danes series from the previous example is "LD", the following CSS rule is equivalent to the previous one:

```
series[id="LD"] {
  color1: paleturquoise|darkblue;
}
```

Point model object type

The following table lists the attributes defined for the `point` model object type.

Bean property	Type	Description
x	`int`	The x-value of the data point, as returned by the `getXData(int)` method.
y	int	The y-value of the data point, as returned by the `getYData(int)` method.
index	`int`	The index of the data point in the data set.
label	`String`	The label of the data point, as returned by the `getDataLabel(int)` method.

Here are a few examples of selector patterns that use attribute matching:

```
// Matches the series representing the resource named "Gill Hopper".
series[name="Gill HopperSales"] { ... }
```

```
// Matches all the series, except the first one.
series[index<>"0"] { ... }

// Matches all data points with a y-value greater than 1.
point[y>1]{ ... }

// For the series representing the resource named "Bob Robertson",
// matches the data points whose y-value
// is greater than 1. The '>' transition is used to denote that
// point is a child of series.
series[name="Bob Robertson"] > point[y>1] { ... }
```

By using model indirection, you can reference the attributes on the right side of a declaration. Remember that if the resource implements the `IlvUserPropertyHolder` interface, such as the `IlvGeneralResource` class does, the properties of the resource are available as additional properties of the data series.

For example, suppose that each resource defines an `overloadColor` attribute. You can then define the following rule:

```
// For all data points with a y-value greater than 1, assign a color
// equal to the value of the resource ‘overloadColor’ property.
point[y>1] {
  color1: @overloadColor;
}
```

Note that you can also reference the attributes of a series and its resource within the declarations of a point model object.

CSS classes

The CSS engine of the Resource Data chart interprets the `tags` property of an `IlvGeneralResource` instance, or of any resource that implements the `IlvUserPropertyHolder` interface, as the space-separated list of the CSS classes it belongs to. For example, if the data model defines a `tags` value of "Europe" for some of the resources, you can then specify the following rule that will color all data series that represent resources which are members of the "Europe" class in a different color:

```
series.Europe {
  color1 : red;
}
```

This is similar to the way the Gantt and Schedule charts interpret the `tags` property of activities that implement the `IlvUserPropertyHolder` interface. This is discussed in the *IlvGeneralActivity CSS classes*.

Properties

Discusses the properties which you can use to customize the rendering of data series and data points. Properties for the `point` model objects are also available for `series` model objects.

The following table shows the properties for data points.

Name	Type	Default value
color1	java.awt.Paint	null
color2	java.awt.Paint	null
endCap	int (enumerated)	java.awt.BasicStroke.CAP_BUTT
lineJoin	int (enumerated)	java.awt.BasicStroke.JOIN_BEVEL
lineStyle	float[]	null
lineWidth	float	1
miterLimit	float	10
stroke	java.awt.Stroke	null
annotation	IlvDataAnnotation	null
visible	boolean	true

Colors

The `color1` property corresponds to the primary color and `color2` to the secondary color.

The meaning of these colors depends on whether the point is displayed by a filled renderer (see `isFilled()`):

- For renderers that are filled, the primary color corresponds to the fill color and the secondary color corresponds to the stroke color.
- For renderers that are not filled, the primary color corresponds to the stroke color and the secondary color is not used. However, it is set as the fill color of the `IlvStyle` instance used by the renderer.

Stroke style

The stroke that is used by the graphical representation of a data point can be specified either:

- by setting the `stroke` property.

  You can do this by using an @-construct to reference a `java.awt.Stroke` instance,

  or

- by setting the various line attributes: `endCap`, `lineJoin`, `lineStyle`, `lineWidth` and `miterLimit`.

> **Note**: If the `stroke` property is set, these properties are ignored.

For example, the following rules are equivalent:

```
series  {
    lineWidth: 2;
    endCap: CAP_ROUND;
    lineJoin: JOIN_ROUND;
}
```

and:

```
series  {
    stroke: @=stroke1;
}

Subobject#stroke1 {
    class : 'java.awt.BasicStroke(lineWidth, endCap, lineJoin)';
    lineWidth : 2;
    endCap : CAP_ROUND;
    lineJoin : JOIN_ROUND;
}
```

Visibility

The `visible` property allows you to toggle the visibility of data points. For example:

```
// Hide the series whose resource is "CPU #1".
series[name="CPU #1"]  {
    visible: false;
}

// For all series, hide the points whose y-value is negative.
point[y < 0]  {
    visible: false;
}
```

Annotation

The `annotation` property lets you connect an instance of `IlvDataAnnotation` to a data point. For information on data annotations, see *Annotations* in *Developing with the SDK* for JViews Charts.

The following example shows a rule that associates an icon with a set of data points.

```
// For the "CPU #1" series, set an icon on the points
// whose y-value is greater than 50.
```

```
series[name="CPU #1"] > point[y>50]  {
    annotation: @=upperAnnotation;
}

Subobject#upperAnnotation {
    class: 'ilog.views.chart.graphic.IlvDefaultDataAnnotation(URL, position,
offset)';
    URL: url('gif/ok.gif');
    position: NORTH;
    offset: 2;
}
```

Properties for data series

As shown in the previous examples, the `series` model object selects data series based upon attributes of the `series` or the resource it represents. The Bean properties that are styled by the `series` rule can belong to either the data points of the series or to the `IlvChartRenderer` instance that displays the series. On top of the data point properties, the `series` model object can be used to modify the Bean properties of the renderer that displays the corresponding data set. The Resource Data chart default renderer is an instance of `IlvStairChartRenderer`. For example, you can define the following rule:

```
// Specify that the series whose resources have a "hidden" CSS class are not
// displayed by the legend.
series.hidden {
  visibleInLegend: false;
}
```

For information on the available properties, please refer to the documentation of the `IlvChartRenderer` class and its subclasses in the *Java API Reference Manual*.

Note: Properties specified in a rule using the `series` model object usually override the settings specified by the `chartRenderer` model object.

Gantt charts

Explains how to handle rendering and interaction in the Gantt and Schedule charts.

In this section

The architecture of the Gantt charts
Describe the classes behind activity or resource based Gantt charts and how they are associated.

The Gantt beans
Explains the way in which the Gantt chart bean and Schedule chart bean encapsulate the Gantt library.

Basic steps for using the Gantt chart and Schedule chart beans
Describes the steps needed to incorporate a chart into the code of your application.

Running the samples
Describes how to run the Gantt chart samples supplied with JViews Gantt.

Using Gantt and Schedule charts
This section describes how to control the appearance of the charts, manipulate rows, and control scrolling.

Using the Gantt sheet
This section describes how to render data in the Gantt sheet.

Using the time scale
Explains how to compose your time scale by specifying its rows.

© Copyright IBM Corp. 1987, 2009

Customizing Gantt charts
Describes the sample applications used in this documentation.

Customizing activity rendering
Explains how to customize the visual representation of one or more activities.

Customizing table columns
Explains how to customize an existing column in the table portion of the Gantt chart or Schedule chart and also how to define a new type of column and add it to the table.

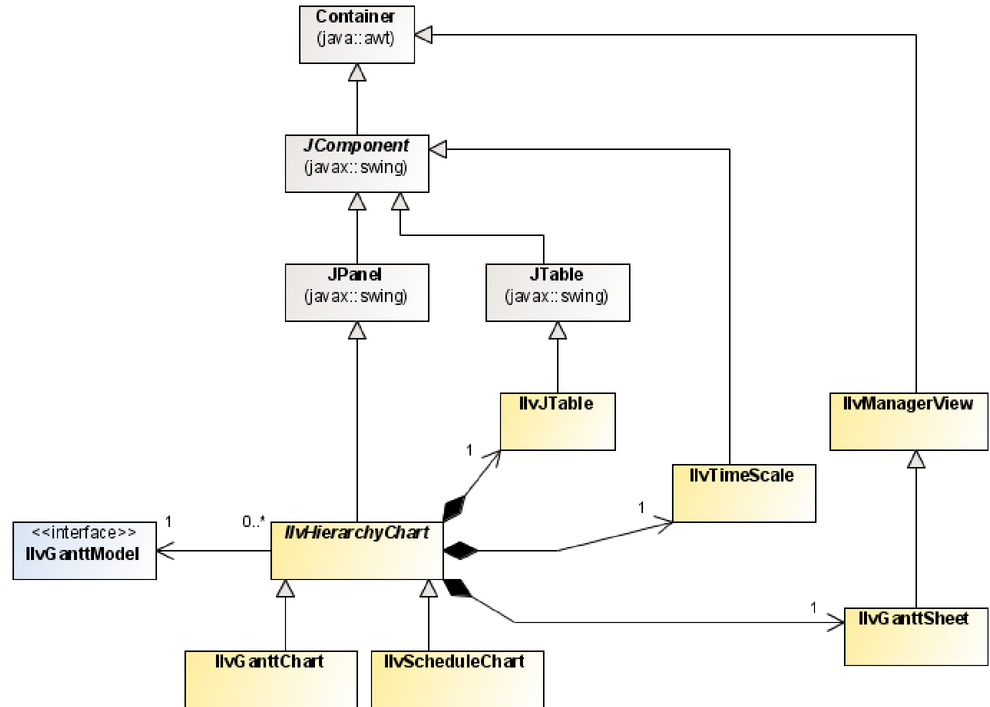
Interacting with the Gantt charts
Describes the association between classes and interactors, explains how predefined interactors work and how to use them.

Interacting with the Gantt sheet using the mouse
Explains how to use the mouse to create activity and reservation graphics in a Schedule chart or constraints in a Gantt chart.

The architecture of the Gantt charts

Gantt charts can be activity based (`IlvGanttChart`) or resource based (`IlvScheduleChart`). The classes for the different types of Gantt chart are encapsulated by the high level Beans described in *The Gantt beans*.

The following figure shows the associations between the classes that represent the different types of Gantt chart and the other main classes for handling Gantt charts in the JViews Gantt API.



The Gantt beans

Explains the way in which the Gantt chart bean and Schedule chart bean encapsulate the Gantt library.

In this section

Overview
Explains the similarities and differences between the high-level Gantt beans.

Structure
Describes the beans used to arrange and coordinate interface components.

Properties
Describes the Gantt bean properties used to control appearance.

Overview

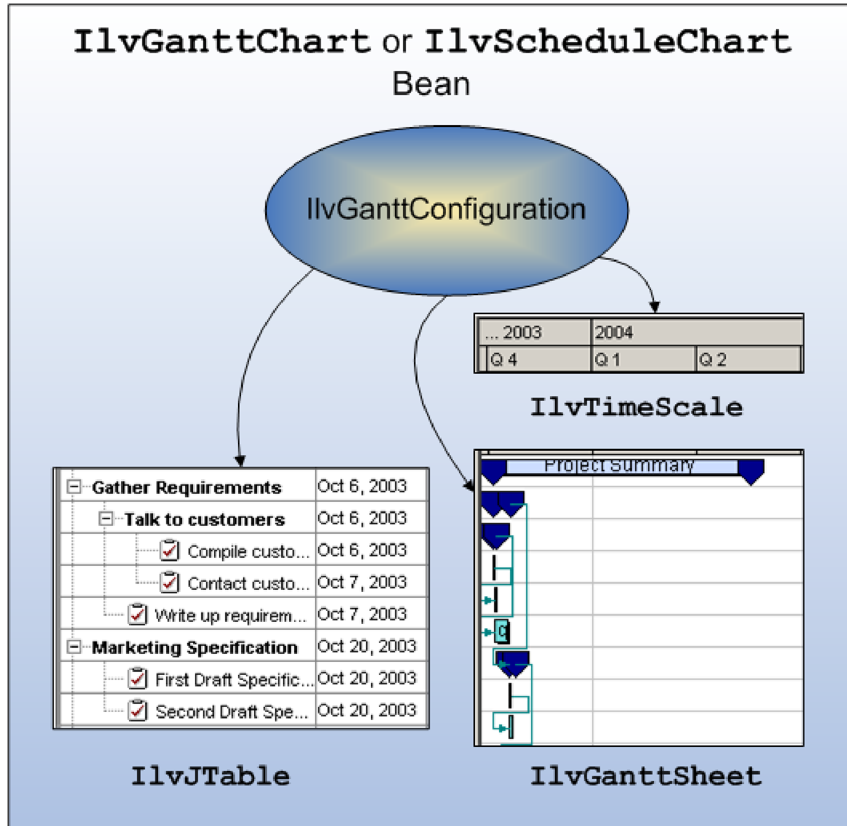
JViews Gantt features two high-level Beans, called Gantt chart bean and Schedule chart bean. Their API is based on the classes `IlvGanttChart` and `IlvScheduleChart`, both subclasses of `IlvHierarchyChart`.

The Beans encapsulate the Gantt library. Although the library can be used without the Beans, you will find it easier to rely on these Beans. Together with the `IlvGanttModel` interface, the two Beans make up the main classes for handling Gantt and Schedule charts in the JViews Gantt API.

Both chart Beans have a similar architecture and have many properties and attributes in common.

Structure

The full Gantt library allows you to arrange user interface components, such as tables, trees, time scales, and Gantt sheets in almost any layout to display the Gantt data. The coordination of the user interface components is handled by the `IlvGanttConfiguration` class (`ilog.views.gantt` package). The Beans can be described as a predefined combination of a configuration (`IlvGanttConfiguration`), a table (`IlvJTable`), a Gantt sheet (`IlvGanttSheet`), and a time scale (`IlvTimeScale`) as shown in the following table.



Properties

Gantt Beans have several properties that control their appearance, such as font, background and foreground color, and hiding or showing the table. The data model is attached to the Beans through the `setGanttModel(ilog.views.gantt.IlvGanttModel)` method, inherited by `IlvGanttChart` and `IlvScheduleChart` from their base class `IlvHierarchyChart` (`ilog.views.gantt` package).

More detailed properties, such as column width or column order, can be handled through the API of the table itself. To do so, you can retrieve a reference to the table through the `getTable()` method, inherited from the base class `IlvHierarchyChart`.

The object returned by this method is an instance of the class `IlvJTable`, which is a subclass of the standard Swing class `JTable`. Therefore, any customization allowed on a `JTable` object is also possible on `IlvJTable` objects.

Similarly, detailed properties of the Gantt sheet, such as the visual aspect of the vertical and horizontal grids, can be manipulated through the API of the sheet itself. You can retrieve a reference to the sheet through the `getGanttSheet()` method of the Bean, also inherited from the base class `IlvHierarchyChart`.

Basic steps for using the Gantt chart and Schedule chart beans

The Gantt chart and Schedule chart beans provide two different views of a Gantt data model.

The basic steps needed to incorporate either chart into the code of your application are very similar:

1. *Stage 1 – Importing the Gantt chart packages*: Import the necessary Gantt packages.
2. *Stage 2 – Creating the Gantt data model*: Create a Gantt data model object by instantiating the interface `IlvGanttModel` and fill it with activities, constraints, resources, and reservations.
3. *Stage 3 – Creating the Gantt chart bean instance*: Instantiate a Gantt chart bean from the `IlvGanttChart` class or a Schedule chart bean from the `IlvScheduleChart` class.
4. *Stage 4 – Binding the Gantt chart to the data model*: Attach the data model to the chart instance.
5. *Stage 5 – Customizing the chart*: Customize the default settings and appearance of the chart, if necessary.
6. *Stage 6 – Adding the Gantt chart to the user interface*: Add the chart instance to the user interface of your application or applet.

Two basic sample Java™ applications are provided to illustrate these steps. The first example demonstrates how to use the activity-oriented Gantt chart:

`<installdir>/jviews-gantt86/samples/ganttChart/src/ganttChart/GanttExample.java`

The second example demonstrates how to use the resource-oriented Schedule chart:

`<installdir>/jviews-gantt86/samples/scheduleChart/src/scheduleChart/ScheduleExample.java`.

Running the samples describes the common steps that are necessary to compile and run both applications and provides the sample code. You can use either of these applications as a starting point for your own work with JViews Gantt. In fact, these basic chart applications are used as the basis for many of the other examples supplied with JViews Gantt.

Running the samples

Describes how to run the Gantt chart samples supplied with JViews Gantt.

In this section

Gantt chart
Describes how to construct and display a chart and run the samples.

Running the sample as an application
Describes how to run the sample as an application.

Schedule chart
Outlines the steps necessary to run the sample application.

Deploying a Gantt application
Explains which JAR file contains the JViews Gantt classes.

Gantt chart

The basic steps for using the Gantt chart bean are illustrated in the **Activity Chart (SDK)** sample. You can find the corresponding source code in:

<installdir>/jviews-gantt86/samples/ganttChart/src/ganttChart/GanttExample.java

Most of the code in the Gantt chart sample is for handling the menus and status bar for the application.

The small portion of code necessary to construct and display the chart itself is outlined here:

```
...
import ilog.views.gantt.*;
import ilog.views.gantt.model.*;
 ...
public class GanttExample extends JApplet
{
  protected IlvGanttChart gantt;
   ...
  public init(Container container)
   {
    super.init(container);
    // Creates the Gantt chart
    gantt = new IlvGanttChart();
    // Creates the Gantt data model
    IlvGanttModel model = createGanttModel();
    // Sets the data model of the Gantt chart
    gantt.setGanttModel(model);
     ...
    // Add the Gantt chart to the panel
    container.add(gantt, BorderLayout.CENTER);
     ...
  }
   ...
  protected IlvGanttModel createGanttModel()
   {
    IlvGanttModel model = new IlvDefaultGanttModel();
    populateGanttModel(model);
    return model;
   }

   protected void populateGanttModel(IlvGanttModel model)
   {
    ... /* Add activities to the data model here */
   }
   ...
  // Initialize example when run as an applet.
  public void init()
  {
    init(getContentPane());
  }

  public static void main (String[] args)
```

```
  {
    JFrame frame = new JFrame("Gantt Chart Example");
    GanttExample ganttChart = new GanttExample();
    ganttChart.init(frame.getContentPane());

    // Exit when the main frame is closed.
    frame.setDefaultCloseOperation(WindowConstants.DISPOSE_ON_CLOSE);
    frame.addWindowListener(new WindowAdapter()
    {
      public void windowClosed(WindowEvent e)
      {
        System.exit(0);
      }
    });

    // Pack the main frame and make it visible.
    frame.pack();
    frame.setVisible(true);
  }
  ...
}
```

To run the samples:

1. Make sure that the Ant utility is properly configured. If not, read Starting the samples for instructions on how to configure `Ant` for JViews Gantt:
2. Go to the directory where the sample is installed and type:

   ```
   ant run
   ```

Running the sample as an application

To run the sample as an application, do the following:

- *Stage 1 – Importing the Gantt chart packages*
- *Stage 2 – Creating the Gantt data model*
- *Stage 3 – Creating the Gantt chart bean instance*
- *Stage 4 – Binding the Gantt chart to the data model*
- *Stage 5 – Customizing the chart*
- *Stage 6 – Adding the Gantt chart to the user interface*

Stage 1 – Importing the Gantt chart packages

To import the Gantt chart packages:

1. In **`<installdir>/jviews-gantt86/samples/ganttChart/src/ganttChart/GanttExample.java`**, import the packages that are common to all the Gantt samples:

```
import shared.*;
import shared.data.*;
import shared.swing.*;
```

2. Import the necessary Gantt chart packages:

```
import ilog.views.gantt.*;
import ilog.views.gantt.action.*;
import ilog.views.gantt.model.*;
import ilog.views.gantt.property.*;
import ilog.views.gantt.swing.*;
```

3. Import the various Swing and AWT packages necessary to build the rest of the user interface of the samples:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Both the Gantt chart sample and the Schedule chart sample derive from a common superclass, `AbstractGanttExample`, that is itself a subclass of `AbstractExample`. These classes contain the code that is shared between all the Gantt samples. Because the samples is written so that it can be run both as an applet or as an application, `AbstractExample` extends the Swing class `JApplet` and `GanttExample` provides a static `main` method to launch the frame window:

```
public class GanttExample extends AbstractGanttExample
{
  ...
  public static void main(String[] args)
  {
    GanttExample ganttChart = new GanttExample();
```

```
    JFrame frame = new ExampleFrame(ganttChart);
    frame.setVisible(true);
  }
  ...
}
```

Stage 2 – Creating the Gantt data model

The data model should contain resources and reservations that will be displayed by the Gantt chart. Refer to the *Connecting to data* for detailed information on how to instantiate different Gantt data model implementations and connect to your business data.

To add resources and reservations to the data model:

1. Create a Gantt data model that implements the `IlvGanttModel` interface.

   ```
   IlvGanttModel model = ...
   ```

2. The Gantt chart and Schedule chart samples use an in-memory data model that is created by the following classes:

   ♦ **<installdir>/jviews-gantt86/samples/ganttChart/src/shared/data/SimpleEngineeringProject.java**

   ♦ **<installdir>/jviews-gantt86/samples/scheduleChart/src/shared/data/SimpleEngineeringProject.java**

3. The `SimpleEngineeringProject` class implements a Gantt data model that simulates the scheduling of a typical engineering project. The Gantt chart sample instantiates the data model like this:

   ```
   IlvGanttModel model = createGanttModel();
   ...
   protected IlvGanttModel createGanttModel() {
     return new SimpleEngineeringProject(chart);
   }
   ```

Stage 3 – Creating the Gantt chart bean instance

Create an instance of the Gantt chart bean:

♦ Implement the `createChart` method using the following code.

```
protected IlvHierarchyChart createChart()
  {
  return new IlvGanttChart();
}
```

Stage 4 – Binding the Gantt chart to the data model

Bind the Gantt chart bean to the data model to enables the chart to display the contents of the data model:

♦ Add the following code to your application:

```
chart.setGanttModel(model);
```

Stage 5 – Customizing the chart

The chart is customized in the `customizeChart` method.

The **<installdir>/jviews-gantt86/samples/ganttChart/src/ganttChart/GanttExample.java** file shows the following basic customizations:

- Several activities are expanded so that all their child activities are initially visible. For example:

```
chart.expandAllRows(anActivity);
```

- The width of the table columns are increased. For example:

```
chart.getTable().getColumn("Name").setPreferredWidth(180);
```

- The time interval initially displayed by the chart is set:

```
chart.setVisibleTime(aDate);
chart.setVisibleDuration(new IlvDuration(...));
```

- Animation of zoom-in and zoom-out is enabled:

```
chart.setVisibleIntervalAnimationSteps(4);
```

It is easy to perform other customizations of the chart also.

For example:

1. Change the default height of the displayed rows:

```
chart.setRowHeight(25);
```

2. Change the font used to label the horizontal time scale:

```
chart.setTimeScaleFont(new Font(...));
```

Stage 6 – Adding the Gantt chart to the user interface

The Gantt chart must now be displayed:

- Add it to the center of the container panel provided as the argument to the `init` method, which is set to have a `BorderLayout` attribute:

```
container.add(gantt, BorderLayout.CENTER);
```

The container panel will be the `contentPane` of the `JApplet` object when the sample is run as an applet or it will be the `contentPane` of the frame when the sample is run as an application.

Schedule chart

The basic steps for using the Schedule chart bean are shown in the **`Resource Chart (SDK)`** sample. They are almost exactly the same as those for the Gantt chart bean, described in *Gantt chart*. You can find the corresponding source code in:

`<installdir>/jviews-gantt86/samples/scheduleChart/src/scheduleChart/ScheduleExample.java`

To run the sample as an application:

1. Make sure that the Ant utility is properly configured. If not, read Starting the samples for instructions on how to configure `Ant` for JViews Gantt:
2. Go to the directory where the sample is installed and type:

```
ant run
```

The Schedule chart sample is almost identical to the Gantt chart sample. The main difference is that an `IlvScheduleChart` object is created instead of an `IlvGanttChart` object. Here are the key lines of code that are different in the Schedule chart sample:

```
...
public class ScheduleExample extends AbstractExample
{
  ...
  protected IlvHierarchyChart createChart()
  {
    return new IlvScheduleChart();
  }
 ...
  public static void main(String[] args)
  {
    ScheduleExample scheduleChart = new ScheduleExample();
    JFrame frame = new ExampleFrame(scheduleChart);
    frame.setVisible(true);
  }

 ...
}
```

Deploying a Gantt application

The classes for JViews Gantt are located in the JAR file:

`<installdir>/jviews-gantt86/lib/jviews-gantt-all.jar`.

Using Gantt and Schedule charts

This section describes how to control the appearance of the charts, manipulate rows, and control scrolling.

In this section

Chart visual properties
Elaborates the properties available to control the appearance of Gantt and Schedule chart beans.

Expanding or collapsing and hiding or showing rows
Explains how activities and resources are displayed in a Gantt chart.

Controlling row structure and visibility
Annotates the methods use to control rows in a chart.

Scrolling in the Gantt sheet
Describes how to control horizontal and vertical scrolling in the Gantt sheet.

Chart visual properties

The Gantt and Schedule chart beans have many properties that control their appearance. For example, to change the font used in the column headers of the table, you can use:

```
myChart.setTableHeaderFont(new Font(...));
```

If you want to change the foreground color of the horizontal time scale to blue you can use:

```
myChart.setTimeScaleForeground(Color.blue);
```

The following table shows the `IlvHierarchyChart` methods that control the visual properties of the charts.

Property	Methods
Background color	`Color getTableBackground()`
	`void setTableBackground(Color color)`
	`Color getTableHeaderBackground()`
	`void setTableHeaderBackground(Color color)`
	`Color getGanttSheetBackground()`
	`void setGanttSheetBackground(Color color)`
	`Color getTimeScaleBackground()`
	`void setTimeScaleBackground(Color color)`
Fonts	`Font getTableFont()`
	`void setTableFont(Font font)`
	`Font getTableHeaderFont()`
	`void setTableHeaderFont(Font font)`
	`Font getTimeScaleFont()`
	`void setTimeScaleFont(Font font)`
Foreground color	`Color getTableForeground()`
	`void setTableForeground(Color color)`
	`Color getTableHeaderForeground()`
	`void setTableHeaderForeground(Color color)`
	`Color getTimeScaleForeground()`
	`void setTimeScaleForeground(Color color)`
Table grid	`Color getTableGridColor()`
	`void setTableGridColor(Color color)`
Row height	`int getRowHeight()`
	`void setRowHeight(int rowHeight)`
Divider	`int getDividerLocation()`
	`void setDividerLocation(int location)`
	`int getDividerSize()`
	`void setDividerSize(int size)`
	`boolean isDividerOpaqueMove()`
	`void setDividerOpaqueMove(boolean opaqueMove`

Expanding or collapsing and hiding or showing rows

For the rest of this discussion, an activity in an `IlvGanttChart` or a resource in an `IlvScheduleChart` is referred to as a data node. This convention allows the common behavior of both charts to be described in a concise manner.

The following terms describe the visibility of data nodes and the rows they are displayed on:

♦ An expanded data node is one that is visible and shows its child nodes, making them visible also.

♦ A collapsed data node is one that hides its child nodes. A collapsed node may or may not be visible, depending on whether its parent node itself is expanded or not. If a data node has no child nodes, its expanded or collapsed status is undefined.

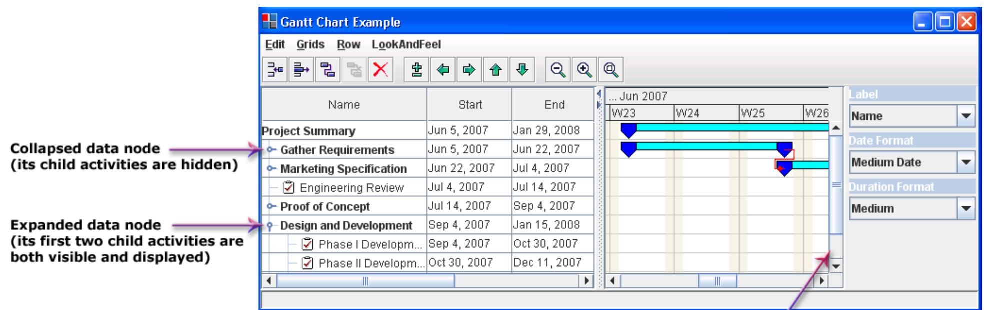
♦ A visible data node is a child of an expanded parent. It is represented by a row, but the user will see that row only if the display area is large enough.

♦ A displayed data node is one that is both visible—that is, its parent node is expanded—and currently within the display area, where it can be seen.

♦ A hidden data node is the opposite of visible. It is a child of a collapsed parent and is not represented by a row.

Note: Scrolling through a window changes the display status of a row, not its visibility status.

The following figure shows the expanded/collapsed and visible/display statuses.



Controlling row structure and visibility

You can use the `IlvHierarchyChart` methods listed in the following table to access and control the expand, collapse, and visibility status of the rows in a chart.

Note: `IlvHierarchyNode` is the abstract superclass of both activities and resources.

The value used by the API will be an activity for an `IlvGanttChart` and a resource for an `IlvScheduleChart`.

The following table shows the methods to control the collapse, expand, and visibility status.

Property	Methods
	`IlvHierarchyNode getRootRow()`
Expand/Collapse	`boolean isRowExpanded(IlvHierarchyNode row)`
	`void expandRow(IlvHierarchyNode row)`
	`void expandAllRows()`
	`void expandAllRows(IlvHierarchyNode row)`
	`void collapseRow(IlvHierarchyNode row)`
Visibility	`int getVisibleRowCount()`
	`int getVisibleRowIndex(IlvHierarchyNode row)`
	`IlvHierarchyNode getVisibleRow(int rowIndex)`
	`boolean isRowVisible(IlvHierarchyNode row)`
	`Iterator visibleRowsIterator(IlvHierarchyNode rootRow)`
	`void makeRowVisible(IlvHierarchyNode row)`
	`Rectangle getVisibleRowBounds(int row)`
	`Rectangle getVisibleRowBounds(IlvHierarchyNode row)`
	`int getVisibleRowIndexAtPosition(int position)`
	`IlvHierarchyNode getVisibleRowAtPosition(int position)`
Displayed	`int getDisplayedRowIndexAtPosition(int position)`
	`IlvHierarchyNode getDisplayedRowAtPosition(int position)`
	`void makeRowDisplayed(IlvHierarchyNode row)`

Scrolling in the Gantt sheet

Describes how to control horizontal and vertical scrolling in the Gantt sheet.

Horizontal scrolling

The time interval displayed by the Gantt sheet and the time scale above it can be modified by using the following methods:

- `Date getVisibleTime()`
- `void setVisibleTime(Date time)`
- `IlvDuration getVisibleDuration()`
- `void setVisibleDuration(IlvDuration duration)`
- `IlvTimeInterval getVisibleInterval()`
- `void setVisibleInterval(Date time, IlvDuration duration)`
- `Date getMinVisibleTime()`
- `void setMinVisibleTime(Date min)`
- `Date getMaxVisibleTime()`
- `void setMaxVisibleTime(Date max)`

The time interval displayed by the Gantt sheet and the time scale above it can be modified by using the following methods:

For example, you can scroll a chart horizontally to the beginning of an activity:

```
IlvActivity activity = ...
Date startTime = activity.getStartTime();
myChart.setVisibleTime(startTime);
```

In the Gantt and Schedule charts, a horizontal scroll bar is displayed below the Gantt sheet. By default, the scroll bar is visible.

You can change this by using the following methods of the class `IlvHierarchyChart`:

- `boolean isHorizontalScrollBarVisible()`
- `void setHorizontalScrollBarVisible(boolean visible)`

The horizontal scroll bar has two operating modes:

- In the default unbounded mode, there is no upper or lower limit to the scrolling. This is indicated by the `getMinVisibleTime()` and `getMaxVisibleTime()` methods returning `null`. The user can use the scroll bar to move forward or backwards in time without limit. However, the scroll bar slider remains in the center of the scroll bar and retains a fixed size.

- The bounded mode is enabled when the methods `setMinVisibleTime(java.util.Date)` and `setMaxVisibleTime(java.util.Date)` have been called with non-null values. In this mode, the scroll bar is limited to the specified time interval and the slider size and position is proportional to the displayed time span.

Vertical scrolling

Vertical scrolling of the Gantt and Schedule charts can be performed using the vertical scroll bar on the right side of the Gantt sheet. By default, this scroll bar is visible only when it is needed.

You can use the following methods to change this behavior:

- `int getVerticalScrollBarPolicy()`
- `void setVerticalScrollBarPolicy(int policy)`

The class `IlvHierarchyChart` has three static constants that define the supported scroll bar policies:

- `IlvHierarchyChart.VERTICAL_SCROLLBAR_AS_NEEDED`
- `IlvHierarchyChart.VERTICAL_SCROLLBAR_NEVER`
- `IlvHierarchyChart.VERTICAL_SCROLLBAR_ALWAYS`

For example, if you want the vertical scroll bar to be always visible, you can write:

```
myChart.setVerticalScrollBarPolicy(myChart.VERTICAL_SCROLLBAR_ALWAYS);
```

You can use the following methods to scroll the chart vertically:

- `int getMaxVerticalPosition()`
- `int getVerticalPosition()`
- `void setVerticalPosition(int position)`
- `int getVerticalExtent()`

Using the Gantt sheet

This section describes how to render data in the Gantt sheet.

In this section

Gantt sheet architecture
Explains the purpose and structure of the Gantt sheet component.

Rendering the data in the Gantt sheet
Describes how to render dates in Activity and Resource Gantt sheets.

Time indicators
Describes the stages you must follow to highlight a specific time in a Gantt sheet.

Activity layouts
Describes the properties of the different layout types.

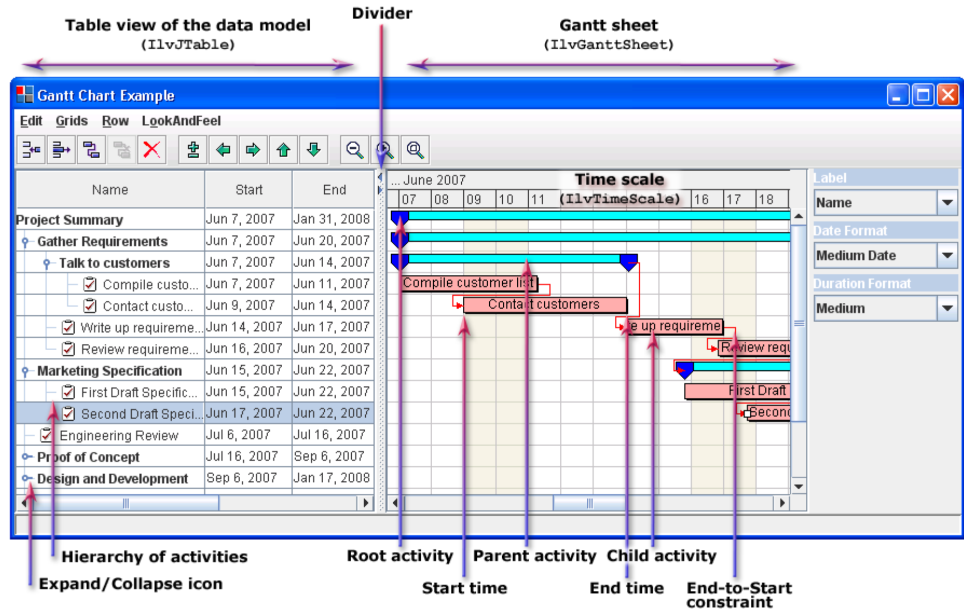
Gantt sheet architecture

The right part of a Gantt chart or Schedule chart is a graphic component called the Gantt sheet, which is an instance of the class `IlvGanttSheet`. It is designed both for the graphical display of the Gantt data in a Gantt data model and as an interactive interface that allows end users to manipulate the Gantt data by means of interactors implemented for this purpose.

The Gantt sheet is a user interface component designed for two main purposes:

- To display the data of a given Gantt data model graphically, namely:
  - Activities and constraints in a Gantt chart
  - Reservations in a Schedule chart
- To let end users interact with the current instance of the `IlvGanttModel` interface by means of a number of interactors developed for this purpose.

The following figure shows the Gantt sheet in a Gantt chart.



Gantt rows

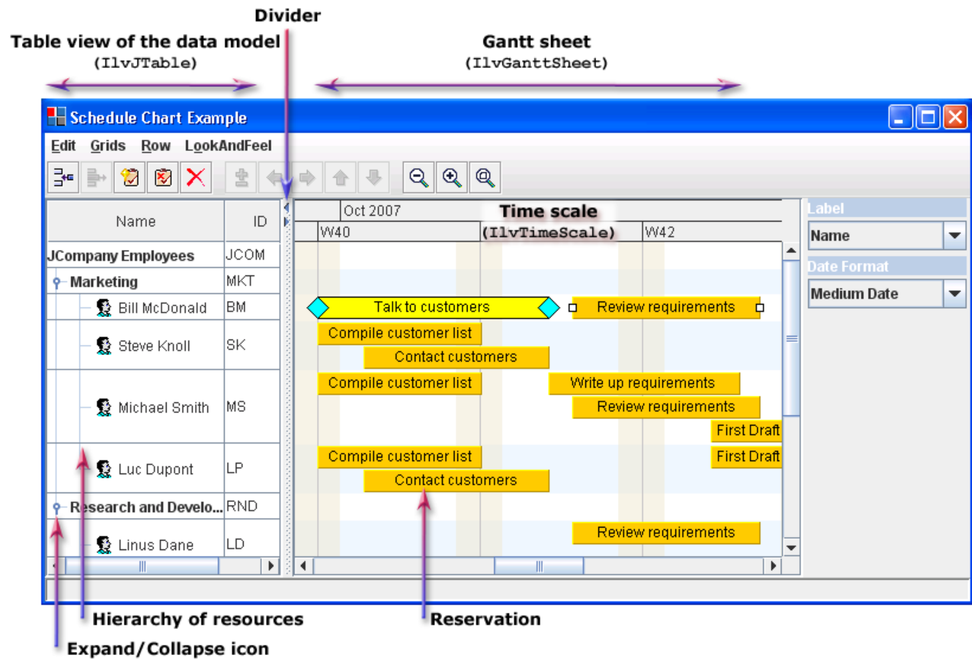
A Gantt sheet consists of several rows, which are instances of the `IlvGanttRow` class.

Rows have the following properties:

- They can be enumerated by a call to one of the methods `getGanttRowCount()` or `ganttRowIterator()` of the `IlvGanttSheet` class.

- They can be visible or hidden. When some rows are hidden, you can use the methods `getVisibleGanttRowCount()` and `getVisibleGanttRowAt(int)` to enumerate the visible ones.
- A Gantt row contains one or more activity graphics to represent activities.

The following figure shows the Gantt sheet in a Schedule chart.



Activity graphics

Activity graphics are instances of the class `IlvActivityGraphic`. They are designed to represent the associated activity, which can be accessed by calling the method `getActivity()`. An activity graphic is drawn as the result of a call to an activity renderer. (See the *Activity renderers* section.) The activity renderer defined for an activity graphic can be accessed or changed by the `getActivityRenderer()` and `setActivityRenderer(ilog.views.gantt.graphic.renderer.IlvActivityRenderer)` methods of the `IlvActivityGraphic` class.

Activity renderers

Activity renderers are objects that implement the `IlvActivityRenderer` interface to render activities. These objects work in association with the `IlvActivityGraphic` class; when an activity graphic needs to be drawn, the `draw(java.awt.Graphics, ilog.views.gantt.graphic.IlvActivityGraphic, ilog.views.IlvTransformer)` method of the `IlvActivityRenderer` interface is called.

Rendering the data in the Gantt sheet

The data of a given Gantt data model is rendered differently depending on whether the Gantt sheet is in a Gantt chart or in a Schedule chart.

This section therefore distinguishes:

♦ *Activity Gantt sheet*

♦ *Resource Gantt sheet*

Activity Gantt sheet

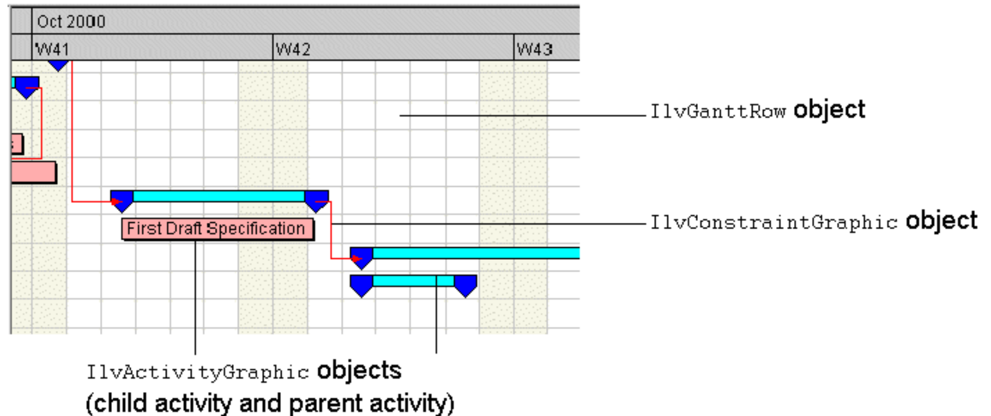
In a Gantt chart, a Gantt row is configured to display only one activity. In other words, there cannot be more than one activity graphic in a Gantt row and the activity graphic cannot be moved to another row.

In a Gantt chart, the Gantt sheet is also configured to show constraints (instances of the interface `IlvConstraint`) by default. See *Constraints* for details.

In a Gantt data model, two activities can be linked by a constraint. In the Gantt sheet, constraints are represented by instances of the class `IlvConstraintGraphic`.

For a given constraint graphic, the associated `IlvConstraint` object can be obtained by calling the method `getConstraint()`. If one or both of the two activity graphics are not visible, the constraint graphic will not be visible either.

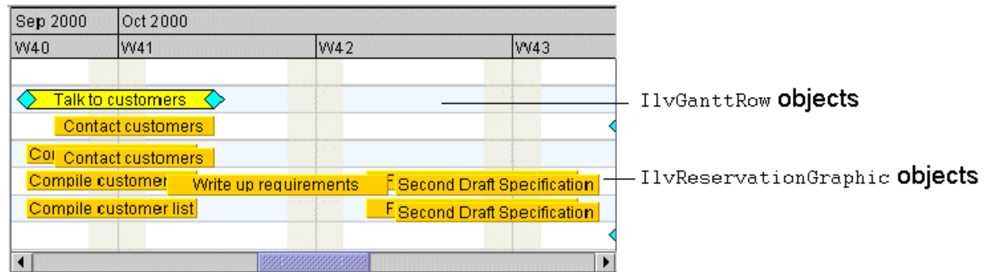
The following figure shows a Gantt sheet in a Gantt chart.



Resource Gantt sheet

In a Schedule chart, the Gantt sheet shows how each resource listed on the left has been scheduled. In other words, the Gantt rows in a Schedule chart display resource reservations. Because a resource can be reserved for more than one activity on a given time span, there can be more than one reservation on one row.

The following figure shows a Gantt Sheet in a Schedule chart.



Reservation graphics are instances of the class `IlvReservationGraphic`. They are designed to render the reservation of resources, which are instances of the interface `IlvReservation`. The class `IlvReservationGraphic` is a subclass of the class `IlvActivityGraphic`. For a given reservation graphic, the associated `IlvReservation` object can be accessed by calling the method `getReservation()`.

In the general case, one activity may reserve several resources and appear as several reservation graphics in the Schedule chart. For this reason, constraints between activities are not displayed by default in the Schedule chart. If each activity reserves at most one resource, constraint links can be displayed in the Schedule chart by calling the `setDisplayingConstraints(boolean)` method.

Time indicators

Time indicators let you highlight a specific time, such as the current time, over the grids in your Gantt sheet. Time indicators can be rendered by any kind of graphics, but the default graphic is a red line. Time indicators are drawn in a specific layer of the Gantt sheet.

This section refers to the following **Highlighting a specific time in the Gantt sheet** code example. You can find the source code for this example in:

`<installdir>/jviews-gantt86/codefragments/application/timeIndicator/src/TimeIndicator.java`

To add a time indicator to a Gantt sheet you need to:

1. Define your own renderer, if the default renderer is not suitable.

   See *Defining your renderer*.

2. Instantiate a subclass of `IlvTimeIndicator`.

   See *Creating a time indicator*.

3. Customize the time indicator.

   See *Customizing a time indicator*.

4. Add the time indicator to the Gantt sheet.

   See *Adding a time indicator to the Gantt sheet*.

You can also control the visibility of time indicators through the specific layer on which they are placed.

Defining your renderer

A time indicator is rendered by an instance of a subclass of `IlvGraphic`. The default renderer is an instance of `IlvLine` with color red and width of 2.

To customize your renderer, do one of the following:

1. Customize the default renderer using the following code:

   Customizing the default renderer

   ```
   //Add the current time indicator. Customize it with some dashes , a width,
    and
   //a color.
   IlvLine line = new IlvLine(0, 0, 0, 100);
   line.setLineWidth(6);
   float[] lineStyle = {10.0f, 15.0f};
   line.setLineStyle(lineStyle);
   ```

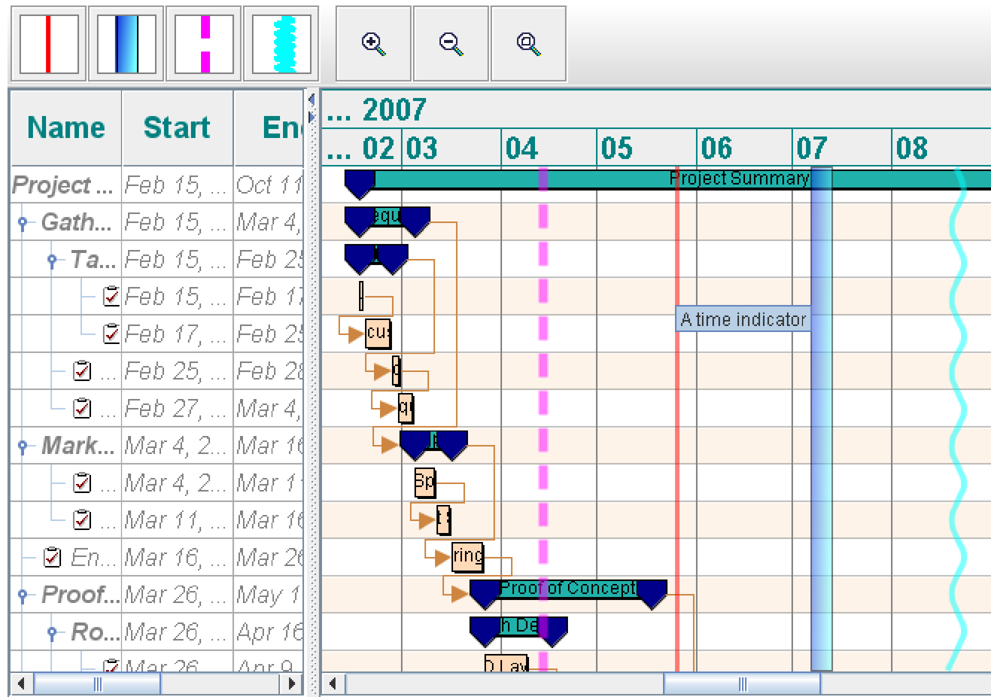
2. Define new renderers, such as `IlvGeneralPath`, to get a more attractive appearance.

   Customizing a renderer with IlvGeneralPath

   ```
   //Add a fixed time indicatorat 12 weeks from today with a customized
   renderer,
   //using an IlvGeneralPath with some gradient paint.
   IlvGeneralPath path = new IlvGeneralPath();
   ```

```
Color fillColor = new Color(20, 50, 225);
Color highlight = fillColor;
for (int i = 1; i <= 5; i++) {
  highlight = highlight.brighter();
}
Color shadow = fillColor;
for (int i = 1; i <= 2; i++) {
  shadow = shadow.darker();
}
Color[] colors = {fillColor, highlight, shadow};
float[] stops = {0, 0.25f, 1};
Point2D start = new Point2D.Float(0, 0);
Point2D end = new Point2D.Float(4, 1);
Paint fillPaint = new IlvLinearGradientPaint(start, end, stops, colors,
true);
path.setFillPaint(fillPaint);
```

The following figure shows a selection of time indicators obtained by running the example.



Creating a time indicator

A time indicator uses an instance of `IlvGraphic` to represent a specific time on the Gantt sheet.

The following classes are provided as subclasses of `IlvTimeIndicator`:

♦ `IlvFixedTimeIndicator` for representing a specific time.

- `IlvCurrentTimeIndicator` for representing the current time. A timer is used to update the Gantt sheet as time passes.

To create customized time indicators do one of the following:

1. Create a default time indicator at a specific time.

   Creating a Default Time Indicator at a Specific Time

   ```
   //Create a default fixed time indicator at 6 weeks from today.
   //It will create a red line.
   //The description "A time indicator" is given (and will be shown as a
   tooltip)
   Date now = new Date();
   Date time = IlvTimeUtil.add(now, IlvDuration.ONE_WEEK.multiply(6));
   IlvFixedTimeIndicator timeIndicator =
             new IlvFixedTimeIndicator(time, "A time indicator");
   ```

2. Create a time indicator at a specific time with a new renderer, such as an `IlvGeneralPath`.

   Creating a time indicator at a specific time with a new renderer

   ```
   IlvGeneralPath path = new IlvGeneralPath();
   ...
   IlvFixedTimeIndicator timeIndicator = new IlvFixedTimeIndicator(time,
   path,
                                 "A second time indicator");
   ```

3. Create a time indicator of the current time, for example, with a customized renderer.

   Creating an indicator of the current time with a customized renderer

   ```
   IlvCurrentTimeIndicator currentTimeIndicator =
       new IlvCurrentTimeIndicator(line);
   ```

Current time indicators have a default description that can be overridden.

Customizing a time indicator

The following table shows the properties of the time indicator that can be customized.

Property	Description
time	Type `java.util.date` for the time that this time indicator represents. You cannot set the time for `IlvCurrentTimeIndicator`, since it is calculated automatically.
maxWidth	The maximum width of the renderer. The default value is 2.
alpha	The alpha value of the layer that holds the graphic object. The default value is 0.5F.
toolTipText	The text displayed as a tool tip. The default text is the description of the time indicator. A null value disables the tool tip.

When you instantiate your own renderer, there is a lot of scope for customizing the rendering of the time indicator. It can also be customized later. Alternatively, you can customize the default renderer.

Customization is achieved through the following properties applied to the underlying graphics:

- foreground
- fillOn
- background
- strokeOn

Note: Setting these properties on some subclasses of `IlvGraphic` has no effect.

You can also control the refresh frequency of `IlvCurrentTimeIndicator`.

To set the `maxWidth`, `strokeOn`, and `foreground` properties.

- Add the following code to your application:

  Setting time indicator properties

```
timeIndicator = new IlvFixedTimeIndicator(time, spline, "A fourth time

                                 indicator");
timeIndicator.setMaxWidth(15);
timeIndicator.setStrokeOn(true);
timeIndicator.setForeground(Color.cyan);
```

Adding a time indicator to the Gantt sheet

You can add a time indicator to the Gantt sheet in the following way:

```
sheet.addTimeIndicator (timeIndicator);
```

Other methods of `IlvGanttSheet` allow you to have some control of time indicators or to manipulate them.

For example:

- `setTimeIndicatorLayerVisible(boolean)` to change the visibility of time indicators.
- Collection `getTimeIndicators()` to obtain all the time indicators.
- IlvTimeIndicator `getTimeIndicator(java.util.Date)` to obtain the time indicator of a specific time or the current time indicator if a null date is passed.
- `replaceTimeIndicator(ilog.views.gantt.graphic.IlvTimeIndicator, ilog.views.gantt.graphic.IlvTimeIndicator, boolean)` to change the time indicator.

The action `IlvScrollToTimeIndicatorAction` is provided to allow you to scroll easily to additional time indicators.

To instantiate this action:

- Add the following code to your application:

Instantiating IlvScrollToTimeIndicator action

```
scrollToTimeIndicatorAction = new IlvScrollToTimeIndicatorAction(
                             sheet, timeIndicator, name, icon,
                             accelerator, shortDescription,
                             longDescription);
```

Activity layouts

A Gantt row in a Schedule chart is configured to render one or more resource reservations corresponding to one or more activities. If the default layout is enabled and the same resource is reserved for more than one activity, the multiple reservation graphics may overlap on the corresponding row. To avoid this and ensure a neat arrangement of the reservation graphics, an `IlvActivityLayout` object is used by an `IlvGanttRow` to compute the positions and z-order of its activity graphics.

The following activity layout implementations are provided:

- *Simple layout*
- *Pretty layout*
- *Tile layout*
- *Cascade layout*

You can also create your own custom implementation of the `IlvActivityLayout` interface or subclass one of the provided implementations.

Use the following methods of `IlvScheduleChart` to access the activity layout of the chart:

- `IlvActivityLayout getActivityLayout()`
- `void setActivityLayout(IlvActivityLayout layout)`

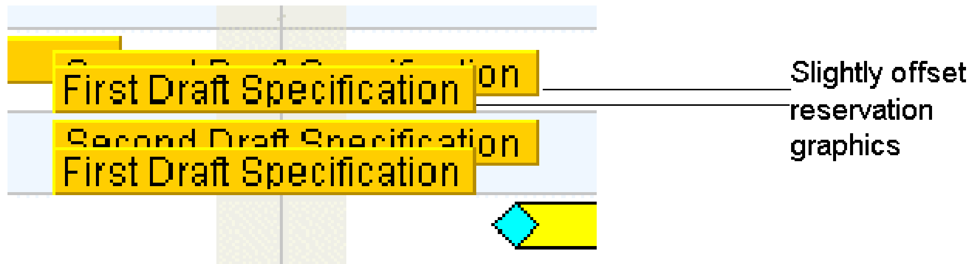
Simple layout

All activity graphics on a given Gantt row have the same `y` position. They are all aligned on the top of the Gantt row and have the same height. The layout does not change the stacking order of the activity graphics (`z` axis). See `IlvActivitySimpleLayout`.

Pretty layout

The following figure shows the result of the Pretty layout option (see the fourth row, for example). The overlapping reservation graphics are arranged with a slight vertical offset. Also, the reservation graphics are stacked so that the higher one (the one that has the greater `y` position) is displayed behind the lower one and both reservation graphics are visible. See `IlvActivityLogisticLayout`.

The following figure shows reservation graphics in pretty layout.

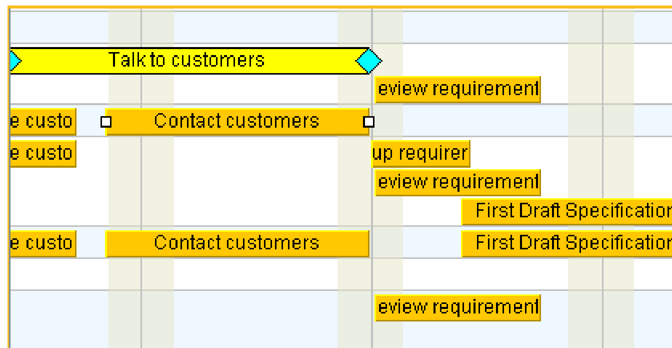


Cascade layout

The Cascade layout is similar to the Pretty layout except that it does not change the stacking order (z axis) of the activity graphics. See `IlvActivityLogisticLayout`.

Tile layout

The following figure shows reservation graphics in Tile layout. The height of activity or reservation graphics does not change when they are displaced.



The Gantt row is divided into subrows, each one accommodating an activity or reservation graphic. The Gantt row can retain a constant height and the height of the subrows is determined by dividing this constant height by the number of subrows. Optionally, the row height can change to accommodate the layout of the graphics and the subrows maintain a constant height. In this case, attempting to set individual row heights on the chart does not work, since the layout overrides this behavior. See `IlvActivityTileLayout`.

Using the time scale

Explains how to compose your time scale by specifying its rows.

In this section

Changing the rows of a time scale
Explains how to customize time scale rows,

Visibility policy
Explains how to control row visibility for an `IlvTimeScale`.

Controlling row visibility
Describes the classes and interfaces provided to control row visibility and explains how to customize row visibility.

Nonlinear time scale
Explains how to emphasize specific time periods by granting them greater screen width than others.

Changing the rows of a time scale

A time scale (class `IlvTimeScale`) is composed of rows (class `IlvTimeScaleRow`). You can compose your time scale by specifying its rows. The Gantt or Schedule chart has a predefined time scale (class `IlvGanttTimeScale`) that has two visible rows and which adjusts the row contents according to the zoom level.

To customize time scale rows, you need to:

1. Create some new time scale rows.
2. Set these rows on the time scale.

The following sections are based on the Relative Time Scale sample found in:

<installdir>/jviews-gantt86/samples/relativeTimeScale.

This sample demonstrates the capability of the time scale component, by customizing the time scale rows. The time scale rows are numbered relative to a reference date.

In the sample, two new rows are created, the new week and day rows. They are numbered starting from a reference date, rather than from the absolute date (which is the default numbering).

To create new rows:

1. Use the relative week row implementation is based on the `IlvWeekTimeScaleRow` class. It can be found in

   <installdir>/jviews-gantt86/samples/relativeTimeScale/src/relativeTimeScale/IlvRelativeWeekTimeScaleRow.java

2. Use the relative day row implementation is based on the `IlvDayTimeScaleRow` class. It are found in:

   <installdir>/jviews-gantt86/samples/relativeTimeScale/src/relativeTimeScale/IlvRelativeDayTimeScaleRow.java

The code for setting these rows are found in:

<installdir>/jviews-gantt86/samples/relativeTimeScale/src/relativeTimeScale/GanttChart.java

To set these rows:

1. Create a time scale and the relative time scale rows:

```
IlvTimeScale timescale = new IlvTimeScale();
weekRow = new IlvRelativeWeekTimeScaleRow();
dayRow = new IlvRelativeDayTimeScaleRow();
```

2. Set the relative time scale reference date. In the example, it is set to be the closest hour:

```
Calendar calendar = Calendar.getInstance();
IlvCalendarUtil.dayFloor(calendar);
referenceDate = calendar.getTime();
```

```
weekRow.setReferenceDate(referenceDate);
dayRow.setReferenceDate(referenceDate);
```

3. Add the rows to the time scale, and set the time scale on the Gantt chart.

```
timescale.addRow(weekRow);
timescale.addRow(dayRow);
chart.setTimeScale(timescale);
```

Visibility policy

To control which rows are visible, the concept of visibility policy has been introduced on the Gantt time scale.

The `IlvTimeScaleVisibilityPolicy` is an interface which lets you adjust the visible rows for the specified time scale through the `adjustRows(java.awt.Graphics, ilog.views.gantt.scale.IlvTimeScale.PaintContext)` method.

When you set a visibility policy on a time scale through the `setVisibilityPolicy(ilog.views.gantt.scale.IlvTimeScaleVisibilityPolicy)` method, the time scale will ask the visibility policy to determine which rows are visible when the time scale visible time interval changes.

The visibility policy dictates that:

- If the policy is set to null, which is the default value, the visibility of the rows is based on their visible property.
- If the policy is not null, the visibility of the rows is adjusted by calling the `adjustRows(java.awt.Graphics, ilog.views.gantt.scale.IlvTimeScale.PaintContext)` method.

Controlling row visibility

To enable control of row visibility, the following classes and interfaces are provided:

- The class `IlvBasicTimeScaleVisibilityPolicy` implements a default visibility policy for `IlvTimeScale`. This policy determines which rows are visible when the time scale is resized, based on some visibility predicates applied to the paint context object. This policy keeps track of each visibility predicate, with its set of visible rows, in a specified order. When asked to adjust the rows visibility, the first visibility predicate that evaluates to `true` is the one used for adjusting the rows.
- The `IlvVisibilityPredicate` interface provides a way to specify a predicate, which should be evaluated.
- The class `IlvTimeWidthVisibilityPredicate` provides a way to specify a condition based on the size of a time unit, which can be pixel-based or character-length-based.
- The class `IlvVisibleTimeScaleRows` provides a way to specify a predicate associated to a list of rows.

The customVisibility code fragment, found in **<installdir>/jviews-gantt86/codefragments/timescale/customVisibility** shows how to use these classes for controlling visibility. This code fragment can also be run as an application.

To customize row visibility:

1. Create the time scale.

```
IlvTimeScale timescale = new IlvTimeScale();
```

2. Create the necessary rows and customize them.

```
// A customized quarter row
IlvQuarterTimeScaleRow quarterRow = new IlvQuarterTimeScaleRow();
quarterRow.setTextColor(Color.white);
quarterRow.setTextFont(new Font("default", Font.PLAIN, 14));
quarterRow.setTextPosition(IlvBasicTimeScaleRow.CENTER);
...
```

3. Add the rows to the time scale.

```
timescale.addRow(quarterRow);
...
```

4. Create the visibility policy.

```
IlvBasicTimeScaleVisibilityPolicy visibilityPolicy =
        new IlvBasicTimeScaleVisibilityPolicy();
```

5. Create the list of visibility conditions.

   Each visibility condition is characterized by a predicate and the list of visible rows when this predicate is true.

```
// First visibility condition is to have the 4 rows visible when at
// least 4 characters can be seen for the month.
// Create the predicate
IlvTimeWidthVisibilityPredicate cond1 =
        new IlvTimeWidthVisibilityPredicate(
            Calendar.DAY_OF_MONTH,
            IlvTimeWidthVisibilityPredicate.CHARACTER,
            4);
cond1.setFont(new Font("SansSerif", Font.BOLD, 18));
// Create the first visibility condition with the predicate
IlvVisibleTimeScaleRows visCond1 = new IlvVisibleTimeScaleRows(cond1);
// Add the visible rows.
visCond1.addRow(quarterRow);
visCond1.addRow(monthRow);
visCond1.addRow(dayRow);
visCond1.addRow(halfDayRow);
...
```

6. Add these visibility conditions to the visibility policy.

```
visibilityPolicy.addVisibleTimeScaleRows(visCond1);
```

7. Set the visibility on the time scale.

```
timescale.setVisibilityPolicy(visibilityPolicy);
```

Nonlinear time scale

In JViews Gantt, the conversion from time to horizontal screen coordinates is performed by an implementation of the `IlvTimeConverter` interface. The default implementation, `IlvLinearTimeConverter`, performs a linear conversion. This implies that all time units of equal duration, such as days and hours, will occupy equal width on the screen. However, it is also possible to define a nonlinear time converter that will emphasize specific time periods by granting them greater screen width than others.

Note: When a nonlinear time converter is set on a Gantt or Schedule chart, scrolling performance is reduced compared to using a linear time converter.

The following table shows the three methods to be implemented in the `IlvTimeConverter` interface.

Method	Description
double getUnits(Date time)	Converts time to `IlvManager` x-axis coordinates
Date getTime(double units)	Converts `IlvManager` x-axis coordinates to time
boolean isLinear()	Indicates whether the time converter is linear

The `getUnits` and `getTime` methods must implement a complementary and reversible conversion. In other words, for any `Date` t:

```
t.equals(getTime(getUnits(t)))
```

and for any manager x-coordinate u:

```
u == getUnits(getTime(u))
```

Any nonlinear time converter must implement the `isLinear` method to return `false`. Use the source code for the nonlinear `WeekTimeConverter` class as the starting point for developing your own custom time converter implementation.

An example implementation of a nonlinear time converter is provided in the Nonlinear Time Scale sample. This sample emphasizes days during the work week by giving them greater screen width than days on the weekend.

To view the sample:

1. Open the file:

   `<installdir>/jviews-gantt86/samples/nonlinearTimeScale`

2. Follow the instructions to run the Nonlinear Time Scale Example.

   The source code of this example is found in:

   `<installdir>/jviews-gantt86/samples/nonlinearTimeScale/src/nonlinearTimeScale/NonLinearTimeScaleExample.java`

To create a nonlinear time scale:

1. Create a nonlinear time converter instance:

```
IlvTimeConverter converter = new WeekTimeConverter();
```

2. Set the nonlinear time converter on the Gantt or Schedule chart:

```
chart.setTimeConverter(converter);
```

Customizing Gantt charts

Describes the sample applications used in this documentation.

In this section

Customization examples
Describes the sample applications on which this documentation is based.

Running the Custom Gantt example
Explains how to run the Custom Gantt example.

Customization overview
Describes the customizations possible using the API.

Customizing the Gantt data model
Explains how to add a user-defined priority property to each activity in the Custom Gantt chart example and then how to create all activity instances with a default priority value by using a factory.

Customization examples

The information in this section is based on the Custom Gantt Chart example.

The **Rendering a Custom Data Model** example is supplied with JViews Gantt to illustrate several advanced customization techniques that can be applied to the charts. The **Rendering a Custom Data Model** example application extends the **Activity Chart (SDK)** example. This application file simplifies the source code so that it only contains the customizations that override the default behavior of the Gantt chart.

Running the Custom Gantt example

The source code file of the Custom Gantt example application is named CustomGanttExample.java and can be found here:

<installdir>/jviews-gantt86/samples/customData/src/customData/CustomGanttExample.java

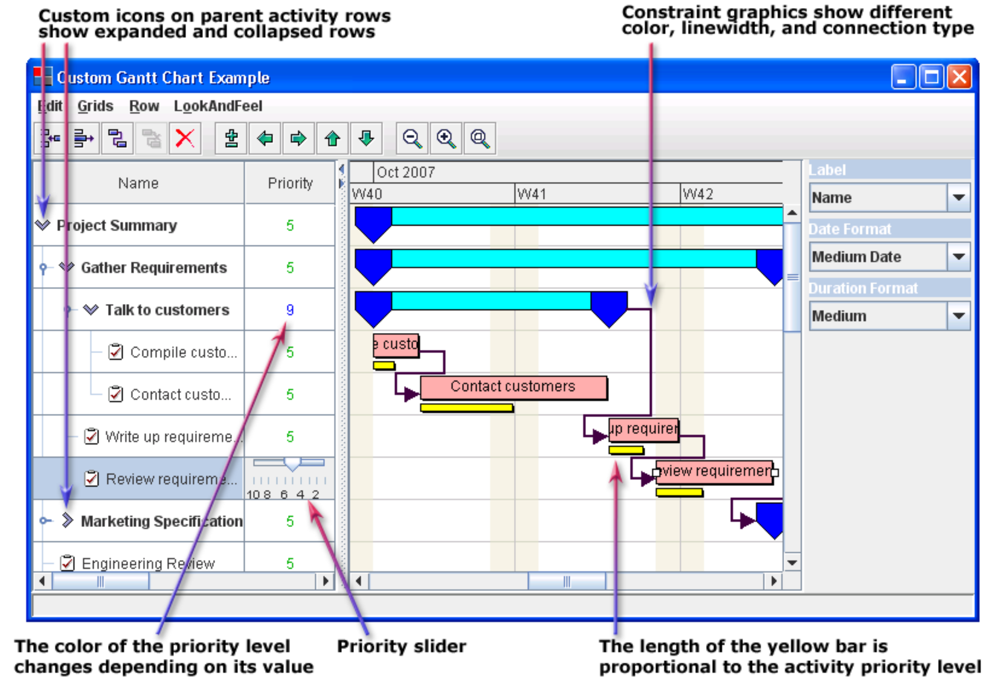
To run the example as an application:

1. Make sure that the Ant utility is properly configured. If not, read Starting the samples for instructions on how to configure Ant for JViews Gantt:
2. Go to the directory where the example is installed and type:

```
ant run
```

Customization overview

The following figure shows what the application looks like when the Custom Gantt example is launched.



This example illustrates several techniques that you can use to customize the Gantt and Schedule charts for your own application needs.

The following customizations are discussed in subsequent sections:

- A numerical user-defined priority property has been added to each activity in the Gantt data model. See *Customizing the Gantt data model*.
- Each leaf activity is rendered with a customized graphic, which represents the activity priority as a horizontal yellow bar. See *Customizing activity rendering*.
- Each parent activity is displayed in the tree column with a custom icon that depends on whether the activity is expanded or collapsed. See *Customizing table columns*.
- A column has been added to the table to display the priority level of each activity. Column rendering has been customized so that priority levels are displayed in different colors depending on their value. A slider has been substituted for the default text field mechanism for editing the priority values. See *Customizing table columns*.

Customizing the Gantt data model

In this section, you learn how to add a user-defined priority property to each activity in the Custom Gantt chart example and then how to create all activity instances with a default priority value by using a factory. You can apply the concepts set out through the tutorial to add your own properties to activities, resources, constraints, or reservations.

Defining your own priority property

The `SimpleEngineeringProject` class implements the data model for this sample. It can be found at:

<installdir>/jviews-gantt86/samples/customData/src/shared/data/ SimpleEngineeringProject.java

This class populates the data model with activities, resources, constraints, and reservations that are created by the data factories of the chart. Because the default data factories of `IlvHierarchyChart` `IlvHierarchyChart` create "general" data objects, the `SimpleEngineeringProject` data model is populated with instances of `IlvGeneralActivity`, `IlvGeneralResource`, `IlvGeneralConstraint`, and `IlvGeneralReservation`.

To define your own priority property:

1. Add a numerical priority property to any activity in the data model by simply using code like this:

```
IlvGeneralActivity activity = ...
activity.setProperty("priority", new Integer(5));
```

2. Encapsulate this behavior in the static utility methods of the class `PriorityProperty`:

```
public static final String PRIORITY_PROPERTY = "priority";
public static final int HIGHEST_PRIORITY = 1;
public static final int LOWEST_PRIORITY = 10;

...

public static void setPriority(IlvActivity activity, int priority) {
  setPriority(activity, new Integer(priority));
}

public static void setPriority(IlvActivity activity, Number priority) {
  if (!(activity instanceof IlvUserPropertyHolder))
    throw new IllegalArgumentException("Priority cannot be set on " +
activity);
  if (priority == null)
    throw new IllegalArgumentException("Priority cannot be null");
  int priorityValue = priority.intValue();
  if (priorityValue > LOWEST_PRIORITY)
    priority = new Integer(LOWEST_PRIORITY);
  if (priorityValue < HIGHEST_PRIORITY)
    priority = new Integer(HIGHEST_PRIORITY);
  ((IlvUserPropertyHolder)activity).setProperty(PRIORITY_PROPERTY,
```

```
priority);
}
```

This ensure that you always use the “priority” property name, that you handle errors in the unlikely case that the activity does not support user-defined properties, and that the priority value is within an acceptable range, you

3. Use the following code to set the value of an activity property instead:

```
IlvGeneralActivity activity = ...
PriorityProperty.setPriority(activity, 5);
```

Creating activity instances

Now that you have a method that helps you to set a numerical `priority` property on an activity instance, you need to have the example application create all activity instances with a default priority value. You do this by creating a custom activity factory and then by telling the chart to use this factory instead of the default.

1. Implemented a custom activity factory by creating the `CustomGanttExample.CustomActivityFactory` inner class and its `createActivityImpl` method, which creates activity instances of `CustomActivity` upon request by the application:

Creating an activity factory to instantiate activities

```
class CustomActivityFactory extends SimpleActivityFactory {
...
  protected IlvActivity createActivityImpl(IlvTimeInterval interval,
                                           int activityNum) {
    IlvGeneralActivity activity =
      new IlvGeneralActivity("CA #" + activityNum,
                             "Custom Activity #" + activityNum,
                             interval);
    PriorityProperty.setPriority(activity, PriorityProperty.
DEFAULT_PRIORITY);
    return activity;
  }
```

2. Get the Custom Gantt chart sample to use the new factory by overriding its `customizeFactories` method:

Overriding the customizeFactories method to use the new factory

```
protected void customizeFactories() {
  super.customizeFactories();
  // Change the default activity factory.
  chart.setActivityFactory(new CustomActivityFactory(chart));
  ...
```

Customizing activity rendering

Explains how to customize the visual representation of one or more activities.

In this section

The Activity Rendering API
Describes what an activity renderer is and explains the associations between the classes used for rendering.

Simple activity renderers
Describes the main activity renderers and the differences between `IlvActivityGraphic` and `IlvGraphic`.

Combining activity renderers
Explains how to combine default and customized activity renderers using the Custom Gantt Chart sample.

Rendering Activities with Multiple Dates
Explains how to render activities that include several intermediary dates.

Using Composite Graphics
Explains how activity rendering can be customized with the API.

Installing Custom Activity Renderers
Explains how to assign a renderer to all activities using factory classes.

The Activity Rendering API

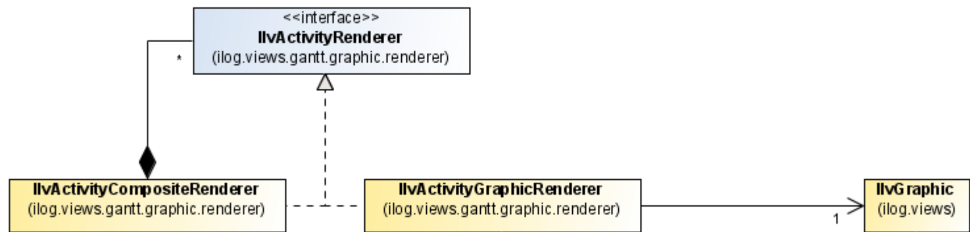
An activity renderer is an object that contributes to the visual representation of one or more activities. In the simplest case, an activity renderer is used to draw everything related to an activity. Renderers can also be combined. For example, one renderer is used to draw a bar, another draws a label, and two more draw the markers at the start and end of the activity.

You can apply activity renderers to many activities at once. An activity renderer can store references to the colors, fonts and the dimensions that apply to all corresponding activities. However, normally it does not store the size or label attributes that apply to a single activity. If it does, such information is highly transient and must be recomputed each time the activity renderer is accessed on behalf of the particular activity.

The two main activity rendering classes are:

- `IlvActivityGraphicRenderer`: draws a simple graphic,
- `IlvActivityCompositeRenderer`: combines other renderers.

The following figure shows the associations between rendering classes.



Simple activity renderers

Simple activity renderers draw a single, simple graphic representation. All simple activity renderers are subclasses of `IlvActivityGraphicRenderer`. An instance of `IlvActivityGraphicRenderer` uses a single `IlvGraphic` object as a delegate for drawing.

The differences between the `IlvActivityGraphic` and `IlvGraphic` are:

- An `IlvActivityGraphic` instance delegates rendering tasks to its renderer. The simple parts of a renderer delegate to their respective `IlvGraphic` instances.

- An `IlvActivityGraphic` instance is part of a Gantt sheet. Calls to `IlvActivityGraphic.getGraphicBag()` return an `IlvGanttRow` instance. In turn, calls to `IlvGanttRow.getGraphicBag()` return an `IlvGanttSheet` object. Calls to `getGraphicBag()` from the graphic delegate of a renderer, on the other hand, return `null`.

- An `IlvActivityGraphic` object always represents an entire activity. The graphic delegate of a renderer contained in an `IlvActivityCompositeRenderer` instance is responsible for a part of the drawing of the activity.

The following table shows the most important classes of simple activity renderers.

Class	Description
`IlvActivityBar`	Draws a bar. Optionally, a label is added to the bar. This label is always clipped to the bounds of the bar.
`IlvActivityLabel`	Draws a label. You use this class when you need an unclipped label.
`IlvActivitySymbol`	Draws a symbol at the start or end of an `IlvActivityBar`.

Combining activity renderers

You combine activity renderers with the `IlvActivityCompositeRenderer` class.

An `IlvActivityCompositeRenderer` instance holds a number of child renderers and delegates rendering tasks to them. Child renderers with a higher index are drawn after renderers with a lower index, and appear to be placed above them in the Gantt chart.

An `IlvActivityCompositeRenderer` instance has a fixed number of child renderers. While the renderer is in use, every child renderer is assigned a fixed index. You can remove a child renderer by storing `null` at its index. However, it is not possible to reduce the number of available indices.

For example, suppose an `IlvActivityCompositeRenderer` instance has the following child renderers:

- An `IlvActivityBar` instance that represents the main bar.
- An `IlvActivityBar` instance that represents the completion bar, This bar is drawn on top of the main bar.
- An `IlvActivityLabel` instance that represents a label drawn to the right of the main bar.
- An `IlvActivitySymbol` instance that represents the marker displayed at the start of the activity.
- An `IlvActivitySymbol` instance that represents the marker displayed at the end of the activity.

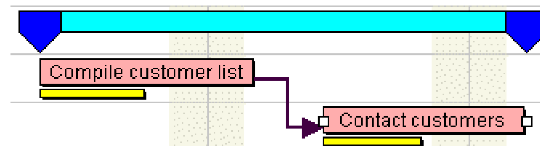
The following information is drawn from the Rendering a Custom Data Model sample located in:

<installdir>/jviews-gantt86/samples/customData

The Custom Gantt Chart example represents an `IlvGeneralActivity` instance in a data model with customized rendering:

- Parent activities are rendered in the default manner, that is, using a light blue bar with dark blue end markers. The renderer is an instance of `IlvActivitySummary`, a subclass of `IlvActivityCompositeRenderer`.
- Leaf activities are rendered as a composite of the default `IlvActivityBar` class. This object displays the name of the activity with a thin yellow activity bar. The following figure displays an activity bar that is proportional to the priority level of the activity.

The following figure shows the rendering of CustomActivity instances.



Some activities show a black completion bar below the main bar. The completion value is taken as a property from the activity. The **<installdir>/jviews-gantt86/samples/customData/src/customData/CompletionProperty.java** class provides centralized access to this property. The static methods `getCompletion` and `setCompletion` are used to access an activity while providing a range check and a default value:

```
public static Number getCompletion(IlvActivity activity) {
  if (!(activity instanceof IlvUserPropertyHolder))
    return null;
  Number completion =
 (Number)((IlvUserPropertyHolder) activity).getProperty(COMPLETION_PROPERTY);

  if (completion == null)
    completion = new Integer(0);
  return completion;
}

public static void setCompletion(IlvActivity activity, Number completion) {
  if (!(activity instanceof IlvUserPropertyHolder))
    throw new IllegalArgumentException("Completion cannot be set on " +
activity);
  if (completion == null)
    throw new IllegalArgumentException("Completion cannot be null");
  if (completion.doubleValue() < 0)
    completion = new Integer(0);
  ((IlvUserPropertyHolder)activity).setProperty(COMPLETION_PROPERTY, completion)
;
}
```

It is also useful to create a derived, `String` valued property class. The class **<installdir>/jviews-gantt86/samples/customData/src/customData/FormattedCompletionProperty.java** is a full implementation of the `IlvStringProperty` interface. In the Rendering a Custom Data Model sample, you use a `FormattedCompletionProperty` instance to display completion values as text in the table cells or as label on the activity renderers. `FormattedCompletionProperty` is a subclass of `IlvFormattedNumberProperty` and thus inherits its numerical formatting capabilities:

```
public class FormattedCompletionProperty extends IlvFormattedNumberProperty {

  ...

  // Returns the completion proportion of the specified activity.
  protected Object getValueImpl(Object activity) {
    return CompletionProperty.getCompletion((IlvActivity)activity);
  }

  // Sets the completion proportion of the specified activity.
  protected void setValueImpl(Object activity, Object completion) {
   CompletionProperty.setCompletion((IlvActivity)activity, (Number)completion)
;
  }

  ...
}
```

The black bar below the main bar is a **<installdir>/jviews-gantt86/samples/customData/src/customData/CompletionBar.java** object. `CompletionBar` is a subclass of `IlvActivityBar`. In the following code example, the displayed property is set to `null` so no text is rendered.

No Text Rendered

```
class CompletionBar extends IlvActivityBar {
  public CompletionBar() {
    setDisplayedProperty(null);
  }
}
```

The length of the `CompletionBar` object is rendered proportionally to the completion percentage and the duration of the activity. This is done by overriding the `getEndTime` method. The `getStartTime` method remains unchanged. The `isRelayoutNeeded` method is overridden so that each activity is redrawn automatically whenever its completion value changes. The following code sample shows these changes.

Rendering Length Proportionally and Redrawing Automatically

```
// Returns the time point that defines the end of this bar.
public Date getEndTime(IlvActivityGraphic ag) {
  Date startTime = getStartTime(ag);
  IlvActivity activity = ag.getActivity();
  Number completion;
  completion = CompletionProperty.getCompletion(activity);
  if (completion != null) {
    return IlvTimeUtil.subtract(super.getEndTime(ag), startTime)
        .multiply(completion.doubleValue())
        .add(startTime);
  } else {
    return startTime;
  }
}

// Determines if the bounding box may have changed.
public boolean isRelayoutNeeded(ActivityEvent evt) {
  return super.isRelayoutNeeded(evt) || CompletionProperty.getInstance().
isPropertyChangedEvent((EventObject)evt);
}
```

The `CompletionBar.getToolTipText` method has been overridden so that when the mouse hovers over the completion bar, the completion value of the activity is displayed.

Displaying Tool Tips

```
public String getToolTipText(IlvActivityGraphic ag,
                             IlvPoint p,
                             IlvTransformer t) {
  IlvActivity activity = ag.getActivity();
  Number completion = CompletionProperty.getCompletion(activity);
  if (completion != null)
    return MessageFormat.format("{0}% completed",
                    Math.round(completion.doubleValue()*100));
  else
    return null;
}
```

When any type of graphic is selected in a Gantt sheet, an `IlvSelection` instance is created to visually represent the selected state. When an `IlvActivityGraphicObject` is selected, it asks its renderer to create an `IlvSelection` instance by calling `IlvActivityRenderer.makeSelection`. All the `IlvActivityRenderer` implementations provided by JViews Gantt return an `IlvActivityGraphicSelection` instance. `IlvActivityGraphicSelection` is an extension `IlvSelection` that visually represents the selected state of an activity by displaying two squares at either end of the activity graphic. The squares are centered in the vertical direction.

In the Rendering a Custom Data Model sample, the default `CustomActivityRenderer` instance uses a standard `IlvActivityGraphicSelection` instance that displays the two squares centered vertically in the blank space between two horizontal bars. To display the selection squares centered vertically along the upper bar, override `CustomActivityRenderer.makeSelection`:

Controlling How Selection Squares Are Displayed

```
public IlvSelection makeSelection(IlvActivityGraphic g) {
  return new IlvActivityGraphicSelection(g) {
    public IlvPoint getHandle(int i, IlvTransformer t) {
      IlvActivityGraphic ag = (IlvActivityGraphic)getObject();
      IlvActivityCompositeRenderer renderer =
                    (IlvActivityCompositeRenderer)ag.getActivityRenderer();
      // The definition rectangle of the activity graphic gives us the
      // x position of the start and end times.
      IlvRect definitionBox = ag.getDefinitionRect(t);
      // The bounding box of the upper bar gives us its y position.
      IlvRect boundingBox = renderer.getRendererAt(0).getBounds(ag, t);
      IlvPoint p = new IlvPoint();
      float y = boundingBox.y + boundingBox.height/2;
      switch(i) {
        case 0:
          p.move(definitionBox.x, y);
          break;
        case 1:
          p.move(definitionBox.x + definitionBox.width, y);
          break;
      }
      return p;
    }
  };
}
```

In the Rendering a Custom Data Model sample, the background color of a completion bar in a composite renderer is set to black. Vertical margins are added to avoid overlapping with the other bars. The following code sample shows the corresponding part of the `CustomActivityRenderer` class.

Custom Activity Renderer Class

```
private void initializeCompletionBar() {
  ...
  completionBar = new CompletionBar();
  completionBar.setTopMargin(0.5f);
  completionBar.setBottomMargin(0.25f);
  completionBar.setBackground(Color.black);
```

```
  addRenderer(completionBar);
}
```

Rendering Activities with Multiple Dates

Activities which implement the `IlvPropertyHolderActivity` interface, including the `IlvGeneralActivity` and the `IlvTableActivity` classes contain several intermediary dates, rather than only the start and end dates.

This can be used for the following purposes:

- Represent the actual and planned extent of activities in the same Gantt chart, using a single model object for each activity.
- Split activities and breaks. These activities are rendered as several bars with time intervals between two successive bars.
- Display activities with embedded milestones, such as progress validations. No separate milestone objects are needed in the model.
- Display lag time, that is, preparation work prior to the start, and cool down phases for the activity or termination work after the activity is completed can be displayed in the Gantt chart. No separate model objects are needed to display lag time.

The start and end of an activity are computed as the minimum and maximum of the date properties returned by calling `IlvPropertyHolderActivity.getTimeProperties()`. Date properties other than those returned by this method can also be used in the rendering, for example, the starting point of the lag time in the use case above.

Activities with multiple dates are rendered using an `IlvActivityCompositeRenderer`, see *Combining activity renderers* for more information. The child renderers of this `IlvActivityCompositeRenderer` instance are simple renderers on which the start and end time properties have been set.

To represent two bars, one going from the START1 to the END1 dates, and the other from START2 to the END2 date, do the following:

1. Instantiate two `IlvActivityBar` instances.
2. Set the `startTimeProperty` and `endTimeProperty` of the first `IlvActivityBar` to START1 and END1
3. Set the same properties on the second `IlvActivityBar` to START2 and END2.

Note: To display an `IlvActivitySymbol` for a particular date, set the `startTimeProperty` and `endTimeProperty` to the same date, for example END1.

The Custom Gantt Chart example demonstrates how to split activities into bars according to a holiday schedule. For the full code, see:

`<installdir>/jviews-gantt86/samples/customData/src/customData/ComplexEngineeringProject.java`

As part of the model creation, properties indicating the start and end time of each bar are set in the activities. For simplicity, the holiday schedule is assumed to consist only of weekends. As there are a lot of weekends that can interrupting an activity, many properties

are needed. These properties are defined on the fly by calling `CustomActivityRendererFactory.getSplitTimeProperty`:

```
/**
 * This property indicates the number of bars in a split activity.
*/
public static final String SPLIT_COUNT_PROPERTY = "Split";

/**
 * Returns the name of the property that contains the start/end time of the
 * i-th part of a split activity.
 * @param i A nonnegative integer.
 * @param endp true for the end time, false for the start time
*/
public static String getSplitTimeProperty(int i, boolean endp) {
  return ((endp ? "E" : "S") + i).intern();
}
```

The following code example shows how to split the activity time interval. It is taken from:

<installdir>/jviews-gantt86/samples/customData/src/customData/ComplexEngineeringProject.java

```
ArrayList <String> = new ArrayList<String>();
int interval;
for (interval = 0; ; interval++) {
  ...
  String prop1 = CustomActivityRendererFactory.getSplitTimeProperty(interval,
 false);
  String prop2 = CustomActivityRendererFactory.getSplitTimeProperty(interval,
 true);
  activity.setProperty(prop1, t1);
  activity.setProperty(prop2, t2);
  timeProperties.add(prop1);
  timeProperties.add(prop2);
  ...
}
if (interval > 1) {
  activity.setTimeProperties(timeProperties.toArray(new String[timeProperties.
size()]));
  activity.setProperty(CustomActivityRendererFactory.SPLIT_COUNT_PROPERTY, new
 Integer(interval));
}
```

As shown in:

<installdir>/jviews-gantt86/samples/customData/src/customData/CustomActivityRenderer.java

the renderer takes the pairs of time property values and creates a bar renderer for each pair:

```
public CustomActivityRenderer(IlvActivity activity) {
  Integer splitCount =
    (Integer) ((IlvGeneralActivity)activity).getProperty(
                 CustomActivityRendererFactory.SPLIT_COUNT_PROPERTY);
```

```
  int n = splitCount.intValue();
  for (int i = 0; i < n; i++) {
    addRenderer(CustomActivityRendererFactory.getSplitRenderer(i));
  }
  initializeExtraChildRenderers();
}
```

The `CustomActivityRendererFactory.getSplitRenderer` method sets the start and end time properties of the bar renderer for a specific part of a split activity. Without these calls to `setStartTimeProperty` and `setEndTimeProperty`, all bars would extend from the start to the end of the entire activity and overlap.

```
IlvActivityGraphicRenderer barRenderer = (j == 0 ? new IlvActivityBar() : new
 IlvBasicActivtyBar());
barRenderer.setStartTimeProperty(getSplitTimeProperty(j, false));
barRenderer.setEndTimeProperty(getSplitTimeProperty(j, true));
```

Using Composite Graphics

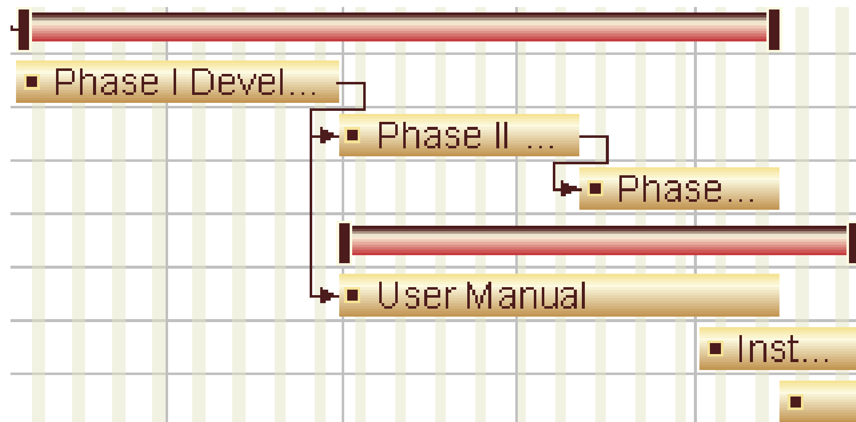
The preferred way of using composite graphics is through the Designer. Composite graphics are not designed to be used programmatically, but the Rendering activities with composite graphics example shows how activity rendering can be customized using the API. The example code can be found at:

<installdir>/jviews-gantt86/codefragments/application/compositeGraphic/src/CGORendererExample.java

The following figure shows that in this sample:

- Parent activities are rendered with composite graphics composed of bars with a color gradient and rectangles as end markers.
- Leaf activities are rendered with composite graphics to display a bar with a color gradient, the name of the activity as text, and a square as decoration positioned near the left end of the bar.

The following figure shows the composite graphics for parent and leaf activity bars.



Composite Graphics

In the Rendering activities with composite graphics example found at **<installdir>/jviews-gantt86/codefragments/application/compositeGraphic/src/CGORendererExample.java**, the class `CustomParentActivityCompositeGraphic` defines a customized graphic for parent activities.

These objects are composed of:

- A rectangle with a color gradient.
- Rectangles as end markers.

The method `IlvGraphic.moveResize` is overridden to change the graphic dynamically depending on its size.

The following code sample shows the definition of the parent-activity composite graphic.

Composite Graphic for a Parent Activity

```
class CustomParentActivityCompositeGraphic extends IlvCompositeGraphic
{
  // The rectangle.
  private IlvGeneralPath base;

  // Decoration 1.
  private IlvRectangle sRect;

  // Decoration 2.
  private IlvRectangle eRect;

  ...

  // Overrides to update graphics depending on the composite's size changes.
  public void moveResize(IlvRect size) {
    super.moveResize(size);
    final int decorationWidth = 5;
    int height = (int) (size.height - 5);
    rect.resize(size.width, height);
    sRect.resize(decorationWidth, size.height);
    eRect.resize(decorationWidth, size.height);
  }
}
```

The class `CustomLeafActivityCompositeGraphic` implementation found in **`<installdir>/jviews-gantt86/codefragments/application/compositeGraphic/src/CGORendererExample.java`** defines a customized graphic for child activities.

These objects are composed of:

- A rectangle with a color gradient that represents the activity bar.
- An `IlvText` object that displays the name of the activity.
- A square decoration.

The method `CustomLeafActivityCompositeGraphic.setLabel` changes the label display, externally through the renderer.

The method `IlvGraphic.moveResize` is overridden to change the graphic dynamically depending on its size.

The following code sample shows the definition of the leaf-activity composite graphic.

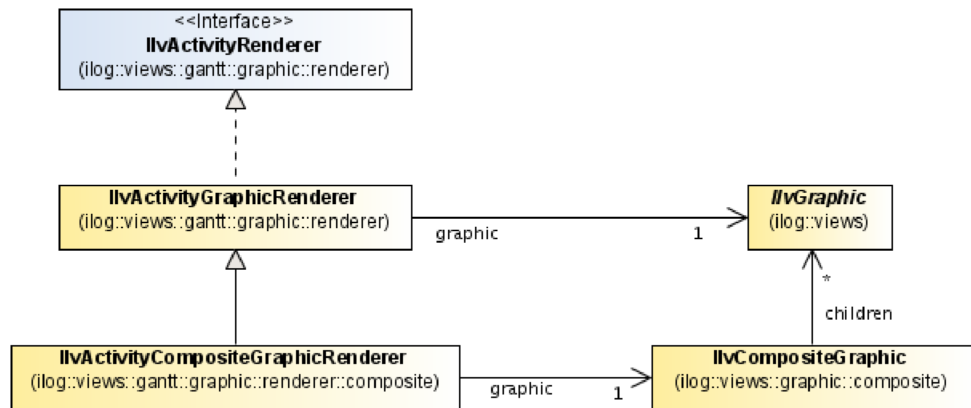
Composite Graphic for a Leaf Activity

```
class CustomLeafActivityCompositeGraphic extends IlvCompositeGraphic {
  // The text graphic used to display the label.
  private IlvText text;
  // A decoration.
  private IlvRectangle rect;

  ...

  // Set a label to this graphic.
```

```
  // @param label The label to display.
  public void setLabel(String label) {
    text.setLabel(label);
  }

  // Overrides to update graphics depending on the composite's size changes.
  public void moveResize(IlvRect size) {
    if (size.width > 0) {
      super.moveResize(size);
      ...
    }
  }
}
```

Composite Graphic Renderers

The following figure shows the classes used to render composite graphics.



In the Rendering activities with composite graphics example found in **`<installdir>/jviews-gantt86/codefragments/application/compositeGraphic/src/CGORendererExample.java`**, the renderer classes `CustomParentActivityCompositeGraphicRenderer` and `CustomLeafActivityCompositeGraphicRenderer` use the definitions of the composite graphics to render parent and leaf activity graphics. They are subclasses of `IlvActivityCompositeGraphicRenderer`. These classes are used to customize the composite graphic with which they are associated.

Note: By default, an associated empty composite graphic is created.

The following code sample shows the definition of `CustomParentActivityCompositeGraphicRenderer`.

Definition of CustomParentActivityCompositeGraphicRenderer

```
class CustomParentActivityCompositeGraphicRenderer extends
  IlvActivityCompositeGraphicRenderer {

  // Builds a CustomParentActivityCompositeGraphicRenderer
  public CustomParentActivityCompositeGraphicRenderer() {
    this.setGraphic(new CustomParentActivityCompositeGraphic());
  }
  ...
}
```

The class `CustomLeafActivityCompositeGraphicRenderer` overrides the method `IlvActivityGraphicRenderer.prepareGraphic` to update the label that is displayed and that depends on the activity name. The class `CustomLeafActivityCompositeGraphicRenderer` also overrides the method `IlvActivityGraphicRenderer.isRedrawNeeded` to trigger graphical updates on changes in activity name.

The following code sample shows the definition of `CustomLeafActivityCompositeGraphicRenderer`.

Definition of CustomLeafActivityCompositeGraphicRenderer

```
class CustomLeafActivityCompositeGraphicRenderer extends
    IlvActivityCompositeGraphicRenderer {

  // Builds a CustomLeafActivityCompositeGraphicRenderer
  public CustomLeafActivityCompositeGraphicRenderer() {
    this.setGraphic(new CustomLeafActivityCompositeGraphic());
  }

  // Overrides to update the label depending on the activity name.
  // @param ag The activity graphic.
  // @param t The current transformer.
  protected IlvGraphic prepareGraphic(IlvActivityGraphic ag, IlvTransformer
t)
  {
    CustomLeafActivityCompositeGraphic graphic =
      (CustomLeafActivityCompositeGraphic) super.prepareGraphic(ag, t);
    graphic.setLabel(ag.getActivity().getName());
    return graphic;
  }

  // Overrides to trigger a redraw when the name is updated.
  public boolean isRedrawNeeded(ActivityEvent evt) {
    if (super.isRedrawNeeded(evt)) {
      return true;
    }
    if (evt instanceof ActivityNameEvent) {
      return true;
    }
    return false;
  }

  ...
}
```

Renderer Factory for Composite Graphics

To customize activity rendering, you must create a customized activity renderer factory that creates the correct renderer on request. For more information, see *Combining activity renderers*. The factory that creates activity renderers in the Composite Graphics code example is the class `CustomActivityCompositeGraphicRendererFactory`.

The following code sample shows the skeleton of this factory.

Factory for Customizing Activity Renderers

```
public class CustomActivityCompositeGraphicRendererFactory extends
  IlvDefaultActivityRendererFactory {

  // Creates a customized activity renderer factory.
  public CustomActivityCompositeGraphicRendererFactory(IlvHierarchyChart chart)
 {
    super(chart);
    // Customizes the leaf activity rendering.
    setLeafActivityRenderer(new CustomLeafActivityCompositeGraphicRenderer())
;
    // Customizes the parent activity rendering.
   setParentActivityRenderer(new CustomParentActivityCompositeGraphicRenderer
());
  }
}
```

Installing Custom Activity Renderers

Activity renderers are installed by an instance of `IlvActivityRendererFactory`. An `IlvActivityRendererFactory` instance assigns a renderer to every activity; the same renderer instance is reused for many activities while treating special kinds activities simultaneously as needed.

The `IlvDefaultActivityRendererFactory` distinguishes activities where the start date and the end date coincide, that is, leaf activities, parent activities, and milestones. An `IlvDefaultActivityRendererFactory` instance uses a single renderer for each of the three categories.

The Rendering a Custom Data Model sample found in located in **<installdir>/ jviews-gantt86/samples/customData** uses the new activity renderer factory by calling its `setActivityRendererFactory` method.

The factory that creates activity renderers for the Rendering a Custom Data Model sample is an inner class of `CustomGanttExample` called `CustomActivityRendererFactory`. This class is an extension of the `IlvDefaultActivityRendererFactory` class. As such, it inherits a default renderer for parent activities.

The following code sample shows the skeleton of the factory.

Factory for Creating Custom Activity Renderers

```
public class CustomGanttExample extends GanttExample {
  ...
  class CustomActivityRendererFactory
  extends IlvDefaultActivityRendererFactory {

    // Creates a customized activity renderer factory.
    public CustomActivityRendererFactory(IlvHierarchyChart chart) {
      super(chart);
      // The leaf renderer is a composite renderer that will contain the 2
      //bars.
      setLeafActivityRenderer(new CustomActivityRenderer());
    }

  }
}
```

The leaf activity rendering is overridden by the call to `IlvDefaultActivityRendererFactory. setLeafActivityRenderer` made in the constructor.

You need to inform the Custom Gantt Chart example to use the new factory before any activities are initially rendered. This is done in the `customizeFactories` method because the chart has been created, but the data model has not yet been populated with activities:

Use the New Factory Before Rendering Activities

```
protected void customizeFactories() {
 super.customizeFactories();
 ...
 // Change the default activity renderer factory.
 gantt.setActivityRendererFactory(new CustomActivityRendererFactory());
}
```

Customizing table columns

Explains how to customize an existing column in the table portion of the Gantt chart or Schedule chart and also how to define a new type of column and add it to the table.

In this section

Running the example
Describes how to run the customData sample.

Tree column icons
Describes how to customize the expanded and collapsed icons for parent activities in a column.

The PriorityColumn class
Describes how the custom table-column class is implemented.

Adding the column to the table
Explains how to add a PriorityColumn object to the table.

Dynamic columns
Explains the custom code implemented in the Dynamic Columns code example.

Running the example

This section refers to the **Rendering a Custom Data Model** sample. You can find the corresponding source code in:

<installdir>/jviews-gantt86/samples/customData/src/customData/CustomGanttExample.java

To run the example:

1. Make sure that the Ant utility is properly configured. If not, read Starting the samples for instructions on how to configure `Ant` for JViews Gantt:
2. Go to the directory where the example is installed and type:

   ```
   ant run
   ```

   to run the example as an application.

Tree column icons

The first column in the table is an instance of the `IlvTreeColumn` class. This type of column uses a renderer that implements the standard Swing `TreeCellRenderer` interface to display its contents. As in the Swing `JTree` component, the default renderer that the column uses is a Swing `DefaultTreeCellRenderer` object. The following code shows how to customize the expanded and collapsed icons for parent activities in the column:

```
IlvTreeColumn treeColumn = gantt.getTable().getTreeColumn("Name");
DefaultTreeCellRenderer renderer =
    (DefaultTreeCellRenderer)treeColumn.getRenderer();
renderer.setOpenIcon(new ImageIcon(...));
renderer.setClosedIcon(new ImageIcon(...));
```

The PriorityColumn class

New custom columns can be added to the table by creating an implementation of the `IlvJTableColumn` interface and adding it to the chart table. The custom table-column implementation is located in the file:

`<installdir>/jviews-gantt86/samples/customData/src/customData/PriorityColumn.java`

This class is an extension of `IlvAbstractJTableColumn`: and thereby implements the `IlvJTableColumn` interface. Start out by duplicating each of the superclass constructors and adding an `init` method, which is where you will customize various aspects of the column.

The basic skeleton of the class looks like this:

```
public class PriorityColumn extends IlvAbstractJTableColumn {

  public PriorityColumn(Object headerValue) {
    super(headerValue);
  }

  public PriorityColumn(Object headerValue, int width) {
    super(headerValue, width);
  }

  ...
}
```

An `IlvJTableColumn` object is a wrapper around a standard Swing `javax.swing.table.TableColumn` object. The underlying `TableColumn` object is responsible for rendering and editing each cell within the column. However, because the standard `TableColumn` object considers row indices, the `IlvJTableColumn` wrapper maps the column to work in terms of `IlvHierarchyNode` data nodes (that is, activities and resources) instead. The `TableColumn` object also provides hooks so that the column can refresh automatically in response to data model events.

The custom `PriorityColumn` has two purposes:

- Rendering the priority of each custom activity as a numeric string. The color of the text should change depending on the value.
- Permitting the application user to edit the priority values using a standard `JSlider` component.

When an instance of `IlvAbstractJTableColumn` (and hence of `PriorityColumn)` is initially constructed, it creates an underlying Swing `TableColumn` object that has no cell renderer or cell editor set. This means that the column cells will be rendered and edited using the class-based default settings of the table. Typically, this means that a `JLabel` object will be used for rendering text and a `JTextField` object will be used for editing. To obtain the customized rendering and editing behavior you want instead, create a `TableCellRenderer` and `TableCellEditor` object for the column explicitly.

Support methods

Before the review of the column renderer and editor, implement the following `IlvJTableColumn` methods in the class `PriorityColumn`.

The methods to be implemented are:

1. The method `getValue(ilog.views.gantt.IlvHierarchyNode)` returns the priority for an activity. Because priorities are stored as primitive `int` values, wrap the priority in an `Integer` object.

```
public Object getValue(IlvHierarchyNode activity) {
  return PriorityProperty.getPriority((IlvActivity)activity);
}
```

2. The method `setValue(ilog.views.gantt.IlvHierarchyNode, java.lang.Object)` sets the priority for an activity. The priority will be passed in as an `Integer` because of the way `getValue` is coded.

```
public void setValue(IlvHierarchyNode activity, Object value) {
  if (!(activity instanceof IlvUserPropertyHolder && value instanceof
Number))
    return;
  PriorityProperty.setPriority((IlvActivity)activity, (Number)value);
}
```

3. The method `isEditable(ilog.views.gantt.IlvHierarchyNode)` is overridden to return `true` so that the priority values can be edited.

```
public boolean isEditable(IlvHierarchyNode activity) {
  Number priority = PriorityProperty.getPriority((IlvActivity)activity);

  return priority != null;
}
```

The cell renderer

The renderer of the priority column is created as an extension of the Swing `DefaultTableCellRenderer` class.

To create the priority column renderer:

1. Overriding the `createRenderer()` method that the class `PriorityColumn` inherits from `IlvAbstractJTableColumn`.

   Only two methods are of the class `DefaultTableCellRenderer` are overridden:

   - Override the `setValue(ilog.views.gantt.IlvHierarchyNode, java.lang.Object)` method to use a `NumberFormat` to format the priority value into a text string for display.
   - Override the `getTableCellRendererComponent` method to center the text in the column and to set the text color based upon the priority value.

   The new additions to `PriorityColumn` look like this:

Additions to the PriorityColumn

```
private NumberFormat formatter = NumberFormat.getInstance();

protected TableCellRenderer createRenderer() {
  DefaultTableCellRenderer renderer = new DefaultTableCellRenderer() {

    private Color darkGreen = Color.green.darker();

    protected void setValue (Object value) {
      if (!(value instanceof Integer))
        setText("");
      else
        setText(formatter.format(value));
    }

    public Component getTableCellRendererComponent
                                       (JTable table,
                                       Object value,
                                       boolean isSelected,
                                       boolean hasFocus,
                                       int row,
                                       int column) {
      Component comp = super.getTableCellRendererComponent
                                           (table,
                                           value,
                                           isSelected,
                                           hasFocus,
                                           row,
                                           column);
      if (value instanceof Integer) {
        int intVal = ((Integer) value).intValue();
        Color color;
        if (intVal <= 2)
          color = Color.red;
        else if (intVal <= 4)
          color = Color.orange;
        else if (intVal <= 7)
          color = darkGreen;
        else
          color = Color.blue;
        comp.setForeground(color);
      }
    return comp;
    }
  };
  renderer.setHorizontalAlignment(JLabel.CENTER);
  return renderer;
}
```

2. Set the `PriorityColumn` to automatically refresh any activities whose value has changed.

   This change will be signaled from the Gantt data model by an instance of `ActivityUserPropertyEvent`. When the column is added to the table, the method `setGanttConfiguration(ilog.views.gantt.IlvGanttConfiguration)` will be called.

Use this opportunity to register the column to receive the desired `ActivityUserPropertyEvent` notifications. To act as an event listener, the column must implement the `GenericEventListener` interface and its `inform` method. In the `inform(java.util.EventObject)` method, the column calls the `cellUpdated(ilog.views.gantt.IlvHierarchyNode)` method of its superclass to refresh the activity whose priority has changed. The relevant code looks like this:

Set the PriorityColumn.

```
public class PriorityColumn extends IlvAbstractJTableColumn
  implements GenericEventListener {

  private IlvGanttConfiguration ganttConfig;

  public void setGanttConfiguration(IlvGanttConfiguration ganttConfig) {

    // When the column is removed from the table, unregister from
    // all notifications.
    if (this.ganttConfig != null)
      this.ganttConfig.removeListener(this);
    this.ganttConfig = ganttConfig;
    // When the column is added to the table, register for
    // user-defined property events.
    if (this.ganttConfig != null)
     this.ganttConfig.addListener(this, ActivityUserPropertyEvent.class)
;
  }

  // GenericEventListener implementation
  public void inform(EventObject event) {
    if (!(event instanceof ActivityUserPropertyEvent))
    // This should never happen, but we will verify anyway.
      return;
   ActivityUserPropertyEvent pEvent = (ActivityUserPropertyEvent) event;

    // Make sure that the event is not an about-to-change event. This
would do
    // no harm, but it would be an unnecessary repaint.
    if (PriorityProperty.PRIORITY_PROPERTY.equals(pEvent.getPropertyName
())
        && pEvent.isChangedEvent())
      cellUpdated((IlvHierarchyNode) event.getSource());
    }
}
```

The cell editor

To be able to use a Swing `JSlider` as the editor for the priority column, there is a subclass of `JSlider` that implements the `TableCellEditor` interface. This class is named `SliderEditor` and is a nested inner class of `PriorityColumn`. All methods of the class `SliderEditor` are basic implementations of the `TableCellEditor` interface.

To create the editor:

♦ Override the `createEditor()` method of the class `PriorityColumn` that it inherits from `IlvAbstractJTableColumn`:

```
protected TableCellEditor createEditor() {
  return new SliderEditor();
}
```

Adding the column to the table

Now that you have designed the `PriorityColumn` class, you need to add an instance of it to the table.

To add a `PriorityColumn` object to the table:

- ♦ Add the column to the `IlvJTable` instance of the chart from its `getTable()` method.

  In the Custom Gantt chart example, this is implemented in the `addCustomTableColumns` method:

```
protected void addCustomTableColumns() {
  IlvJTable table = gantt.getTable();
  table.addColumn(new PriorityColumn("Pri"));
}
```

Dynamic columns

When your Gantt data model implements the in-memory default data model that defines `IlvGeneralActivity` or `IlvGeneralResource` objects, you can add user-defined properties for:

♦ Activities in an `IlvGanttChart`

♦ Resources in an `IlvScheduleChart`

This explanation is based on the Dynamic Columns code example found in:

<installdir>/jviews-gantt86/codefragments/table/dynamicColumns. You can find the corresponding source code in:

<installdir>/jviews-gantt86/codefragments/table/dynamicColumns/src/DynamicColumnSample.java

Starting the sample

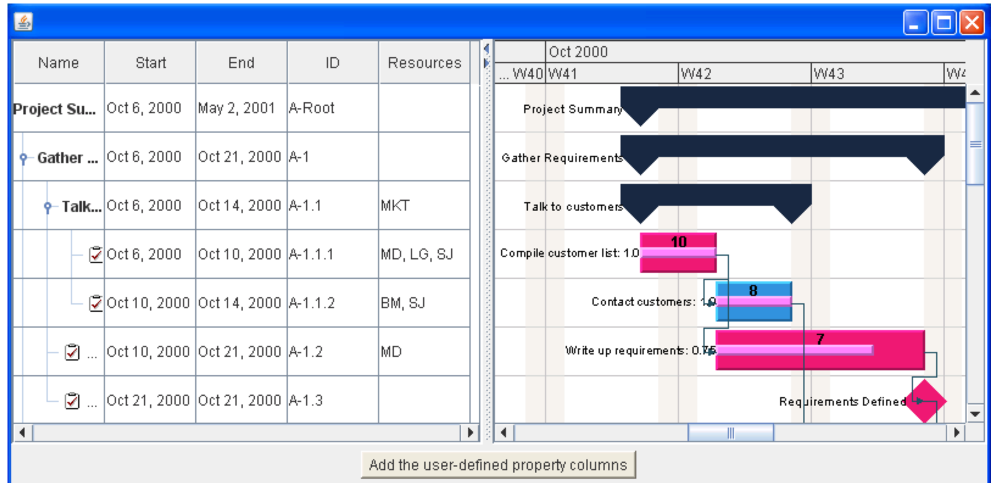
Explains how to start an activity-based Gantt chart with the mandatory properties of each activity and the resources assigned to some activities.

To start the Dynamic columns example:

- ♦ Do one of the following:
  - ♦ double-clicking the executable JAR file

    or
  - ♦ Use Ant as explained in Starting the samples .

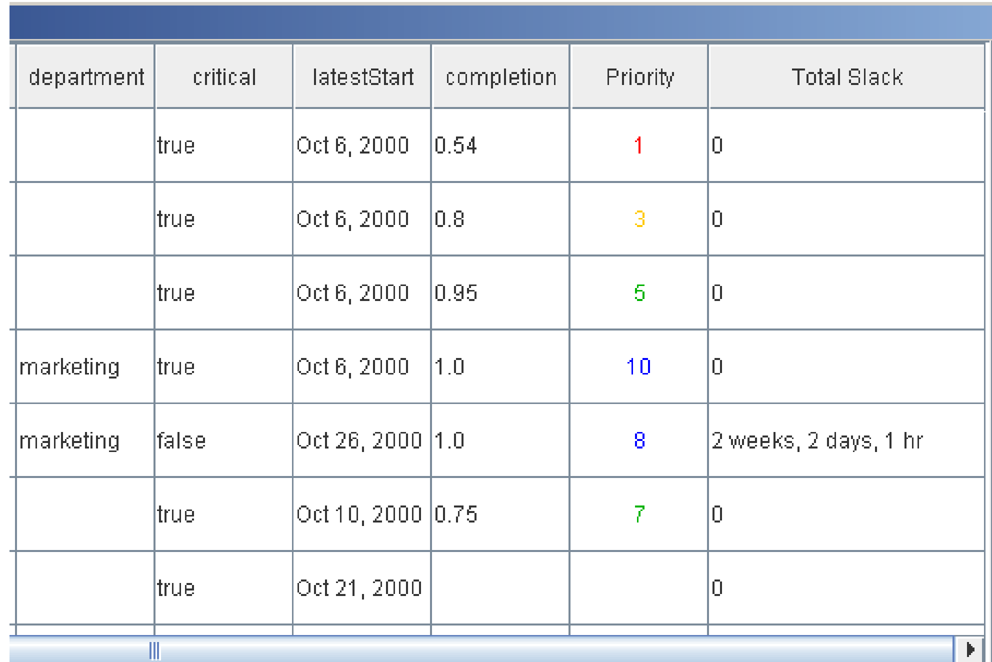
The following figure shows the running the dynamic columns example.



The example shows an activity-based Gantt chart with the mandatory properties of each activity and the resources assigned to some activities.

When you click **Add the user-defined property columns**, the example displays columns with the user-defined properties of the activities as shown in the next figure. Before clicking to display the columns for user-defined properties, you might want to adjust the width of the sample window, so that you can see the additional columns.

The following figure shows the columns for user-defined properties.



department	critical	latestStart	completion	Priority	Total Slack
	true	Oct 6, 2000	0.54	1	0
	true	Oct 6, 2000	0.8	3	0
	true	Oct 6, 2000	0.95	5	0
marketing	true	Oct 6, 2000	1.0	10	0
marketing	false	Oct 26, 2000	1.0	8	2 weeks, 2 days, 1 hr
	true	Oct 10, 2000	0.75	7	0
	true	Oct 21, 2000			0

Adding customized table columns

To add customized table columns that map user-defined properties, you need to:

1. Create the user-defined property adapter.

   This adapter can be accessed by the generic `IlvStringProperty` interface.

   To create the adapter, instantiate an `IlvAbstractUserDefinedProperty` subclass. See *Creating the user-defined property adapter*.

2. Customize the formatting of the property adapter.

   The customization is based on the property class. See *Customizing format*.

3. Create the configurable table column for the property adapter.

   To create the configurable column, instantiate an `IlvConfigurableTableColumn` object. See *Creating a configurable table column*.

4. Add the table column to the table. See *Adding the column to the table*.

Creating the user-defined property adapter

The class `IlvAbstractUserDefinedProperty` provides a common framework for accessing the user-defined properties of an `IlvUserPropertyHolder` through the generic `IlvStringProperty` interface.

The following subclasses of `IlvAbstractUserDefinedProperty` are provided:

- `IlvActivityUserDefinedProperty` for the user-defined properties of an `IlvGeneralActivity` object.

- `IlvResourceUserDefinedProperty` for the user-defined properties of an `IlvGeneralResource` object.

To make to create an adapter for the user-defined property `property` of class `propertyClass` of an `IlvGeneralActivity` object:

- Add the following code to your application:

Creating an adapter of user-defined properties

```
// Create the user-defined property adapter that can be accessed
// through the generic IlvStringProperty interface.
IlvActivityUserDefinedProperty userDefinedProperty =
     new IlvActivityUserDefinedProperty(property, propertyClass);
```

If the property class is not provided, the default is `String`.

Customizing format

A `Format` object can be supplied to convert properties of types other than `String` to or from a string.

The class of the user-defined property is also used to try to convert property values of types other than `String` to or from a string. The conversion that uses the user-defined property class is done by using `convert(java.lang.Object, java.lang.Class)`. This conversion is performed when no format is specified and the property class is not `String`.

To set the format for formatting a `Date` or an `IlvDuration`:

- Add the following code to your application:

Setting the formatting of a date or duration

```
// Customize formatting based on the property class.
if (propertyClass.isAssignableFrom(Date.class)) {
    DateFormat dateFormat =
        IlvDateFormatFactory.getDateInstance(DateFormat.DEFAULT,
                                                               getLocale());
     userDefinedProperty.setFormat(dateFormat);
} else if (propertyClass.isAssignableFrom(IlvDuration.class)) {
    IlvDurationFormat durationFormat =
        new IlvDurationFormat(IlvDurationFormat.TIME_UNIT_MEDIUM);
    durationFormat.setLenientParseMode(true);
    userDefinedProperty.setFormat(durationFormat);
}
```

Examples of the result of applying this code can be seen in the columns **latestStart** and **Total Slack** in *Starting the sample*.

Creating a configurable table column

The class `IlvConfigurableTableColumn` defines a column that can be customized for rendering and editing a property of an `IlvHierarchyNode`, that is, an activity or a resource in an `IlvJTable`. This class is an extension of `IlvStringColumn`, through which it implements the `IlvJTableColumn` interface.

The property that is rendered is defined by an `IlvStringProperty`. Property editing can be enabled separately for rows of parent activities or resources or of leaf activities. or resources.

If editing is not customized, the property will be edited in an `IlvTextFieldTableEditor`. Editing can be customized at instantiation time or through a call to the method `setTableCellEditor(javax.swing.table.TableCellEditor)`.

If the rendering is not customized, `IlvDefaultTableCellRenderer` will be used. Rendering can be customized at instantiation time or through a call to the method `setTableCellRenderer(javax.swing.table.TableCellRenderer)`.

Instantiating a column with specific table cell editor and renderer shows how to instantiate a column with a specific table cell editor and a specific table cell renderer. You specify the cell editor as a slider and render the cell in different colors according to the priority of the activity or resource.

To instantiate a column with a specific table cell editor:

- Add the following code to your application:

Instantiating a column with specific table cell editor and renderer

```
// For the "priority" property use a slider as cell editor and
// render the value with different colors.
IlvConfigurableTableColumn propertyColumn =
    new IlvConfigurableTableColumn (headerValue,
                                    userDefinedProperty,
                                    property,
                                    new SlideEditor(0,10,0),
                                    new PriorityRenderer());
```

The effect of attributing different colors according to priority is shown in the **Priority** column in *Starting the sample*.

Adding the column to the table

To add the configurable column `propertyColumn` to the table, add the column to the `IlvJTable` instance of the Gantt chart.

To do this:

- Use the following code in your application.

```
table.addColumn(propertyColumn);
```

The column is added as the last column in the table. You can then move the position of the column by using *http://java.sun.com/javase/6/docs/api/javax/swing/JTable.html#moveColumn(int, int)*.

Interacting with the Gantt charts

Describes the association between classes and interactors, explains how predefined interactors work and how to use them.

In this section

Class associations for interactors
Illustrates the associations between classes for interactors.

Selecting activities and constraints
Describes how to select activities and constraints and explains the API calls triggered by these actions.

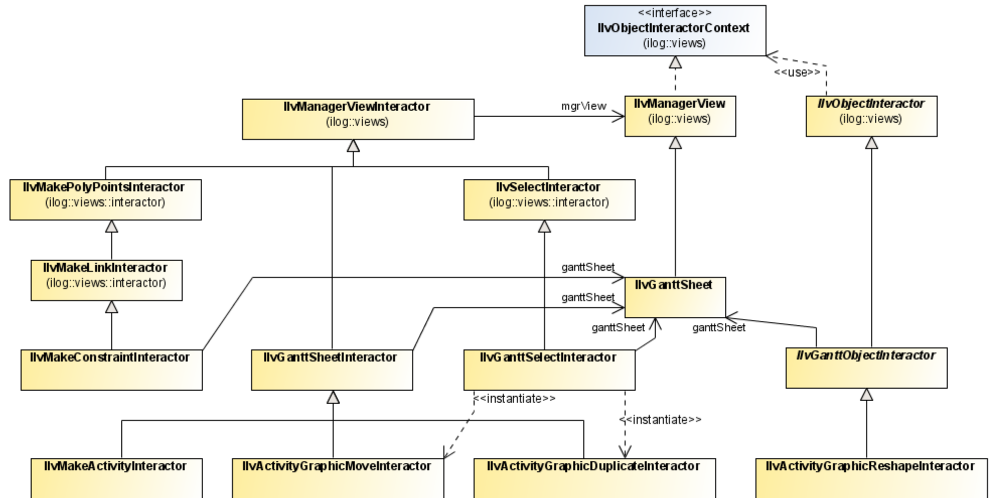
Moving activity and reservation graphics
Explains what happens when you move graphics and the limitations involved in applying these actions.

Duplicating reservation graphics
Explains how to duplicate reservation graphics.

Resizing activity and reservation graphics
Explains how to resize a specific graphic using the GUI.

Class associations for interactors

The following figure shows the associations between classes for interactors.



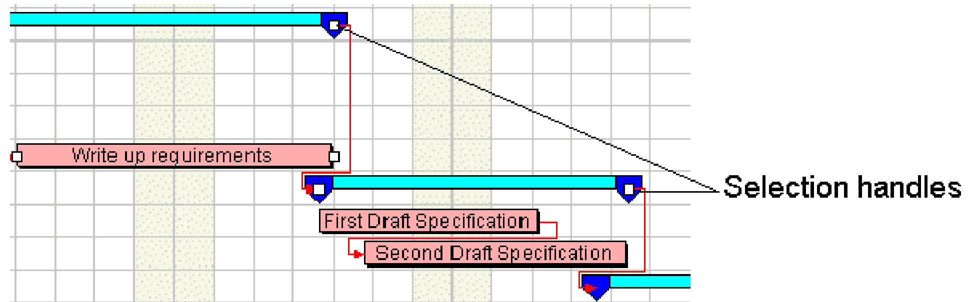
Several predefined interactors are implemented in the Gantt sheet for the following purposes:

- *Selecting activities and constraints*
- *Moving activity and reservation graphics*
- *Duplicating reservation graphics*
- *Resizing activity and reservation graphics*
- *Interacting with the Gantt sheet using the mouse*

Selecting activities and constraints

Before you can manipulate a graphic object, you need to select it. The following figure shows a selected activity graphic and a selected constraint graphic.

The following figure shows the selected activity graphic and constraint graphic.



Installing the selection interactor

A predefined interactor, `IlvGanttSelectInteractor`, handles the graphic selection. To install this interactor, call the `pushInteractor(ilog.views.IlvManagerViewInteractor, java.awt.AWTEvent)` method of the Gantt sheet. When a new Gantt sheet is created the selection interactor is already pre-installed, so the end user does not need to install it explicitly.

Selecting graphics

Once the selection interactor is installed in the Gantt sheet, there are three ways to select activity, reservation, or constraint graphics.

- To select a single graphic object, click it with the left mouse button. Any other previously selected object will be deselected.
- To select several graphic objects, you can also drag a selection rectangle around them using the left mouse button. Be careful not to click a graphic when you start dragging the rectangle. All the graphic objects inside the rectangle will be selected.
- To extend the selection when one object is already selected (multiple selection), Shift-click the next object you want to select. As long as you hold down the Shift key, each click a graphic switches it between selected and deselected.

To access selected graphics, call the method `getSelectedGraphics()` of the class `IlvHierarchyChart`.

Moving activity and reservation graphics

You can move selected activity graphics or reservation graphics by dragging the mouse.

Activity graphic move interactor

When you begin dragging a selected graphic, the Gantt selection interactor calls the method `getMoveSelectionInteractor()` of the class `IlvGanttSelectInteractor` to create a move interactor as an instance of the class `IlvActivityGraphicMoveInteractor`. The new move interactor is then attached to the Gantt sheet and becomes active. Note that the shape of the move cursor changes.

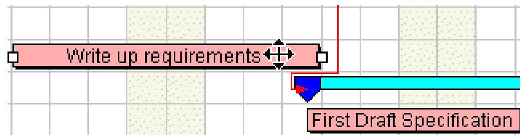
The move interactor will be detached from the Gantt sheet when you release the mouse button.

Note: Constraint graphics cannot be moved in any direction.

Moving activity graphics

In a Gantt chart, activity graphics can only be moved horizontally. This means that you cannot move an activity graphic from one row to another.

The following figure shows how to move a selected activity graphic.



Moving reservation graphics

In a Schedule chart, reservation graphics can be moved horizontally or vertically.

- Moving a reservation graphic horizontally changes the start time and end time of the associated activity.
- Moving a reservation graphic vertically—that is, from one row to another—means dissociating the selected reservation from its current resource and assigning it to a new resource.

Duplicating reservation graphics

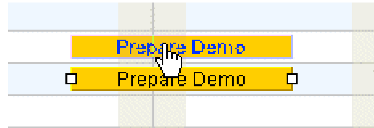
In a Schedule chart, you can duplicate reservation graphics.

To duplicate reservation graphics:

- Press **ALT** and drag the selected reservation graphic.

  The mouse cursor turns into a hand and a copy of the selected reservation graphic is created when you release the mouse button.

  The following figure shows the duplication of a reservation graphic.



Tip: To abort duplication while you are dragging the mouse, press **ESCAPE**.

Resizing activity and reservation graphics

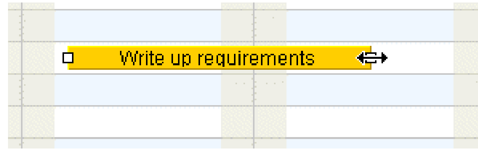
A selected activity or reservation graphic is marked by two handles.

To resize a graphic:

1. Selected the graphic to resize.
2. Dragging any of the handles to the left or to the right to make the bar longer or shorter.

   In doing so, you change the start time and/or end time of the associated activity.

   The following figure show a selected activity graphic being resized.

Interacting with the Gantt sheet using the mouse

Explains how to use the mouse to create activity and reservation graphics in a Schedule chart or constraints in a Gantt chart.

In this section

Creating activities and reservations
Explains how to install the appropriate interactor to the Gantt sheet

Creating constraints
Explains how to install the appropriate interactor to a Gantt sheet.

Popup menus
Explains how to enable and share popup menus in a Gantt sheet.

Creating activities and reservations

To create activity and reservation graphics by using the mouse in a Schedule chart, you must install the appropriate interactor to the Gantt sheet.

To install the appropriate interactor:

1. Create an instance of the class `IlvMakeActivityInteractor`.
2. Attach this interactor to the Gantt sheet by calling the `pushInteractor(ilog.views.IlvManagerViewInteractor, java.awt.AWTEvent)` method of the Gantt sheet.

Once the interactor is installed, you can create an activity or a reservation by drawing a rectangle in the Gantt sheet. The interactor first creates a new instance of `IlvActivity` and then assigns the new activity to the resource where you clicked by creating a new instance of `IlvReservation`.

To create the new activity or reservation, the interactor uses the activity factory or the reservation factory registered with the Gantt sheet. See the methods `getActivityFactory()` and the `getReservationFactory()` of the `IlvGanttSheet` class.

Creating constraints

In a Gantt chart, you can create constraints using the mouse. To do so, you must install the appropriate interactor to the Gantt sheet:

1. Create an instance of the class `IlvMakeConstraintInteractor`.

2. Attach the interactor to the Gantt sheet by calling the `pushInteractor(ilog.views.IlvManagerViewInteractor, java.awt.AWTEvent)` method of the Gantt sheet.

   Once the interactor is installed, you can use it to create `IlvConstraint` objects.

3. Click the source activity graphic (also called From activity).

   Click the left end or right end depending on whether you want to constrain the start time or the end time of the source activity. When you move the mouse, a “ghost” line follows the pointer.

4. Click the target activity graphic (also called To activity).

   Click the left end or right end depending on whether you want to link the source activity to the start time or the end time of the target activity. As soon as you release the mouse button, the arrowed polyline link representing the constraint appears between the two activities.

To create constraints, the interactor uses the constraint factory registered with the Gantt sheet. See the `getConstraintFactory()` method of the class Gantt sheet.

Popup menus

Popup menu support in a Gantt sheet is based on the popup menu support in IBM® ILOG® JViews. For information in popup menu support, see *Tooltips and popup menus on graphic objects* in *The Essential IBM® ILOG® JViews Framework*.

To set up popup support in your application:

1. Call the following code to enable popup menus in a Gantt sheet:

```
IlvGanttSheet.setPopupMenusEnabled(true) :
```

2. Create a `JPopupMenu` object and link it to the graphic in order to associate a specific popup menu with an activity graphic or a constraint.

```
activityGraphic.setPopupMenu(activityMenu);
constraintGraphic.setPopupMenu(constraintMenu);
```

After the popup menu is registered, whenever a user right-clicks the graphic, its popup menu appears.

3. Share popup menus among multiple graphic objects.

Popup menus use a lot of memory. To avoid wasting memory, instead of registering a popup menu with an individual graphic using `graphic.setPopupMenu(...)`, register the popup menu directly with the popup menu manager by calling:

```
IlvPopupMenuManager.registerMenu("ActivityPopupMenu ", activityMenu);
IlvPopupMenuManager.registerMenu("ConstraintPopupMenu ", constraintMenu)
;
```

4. Assign this popup menu to the graphic renderer:

```
graphicRenderer1.setPopupMenuName("ActivityPopupMenu ");
graphicRenderer2.setPopupMenuName("ActivityPopupMenu ");
constraintGraphic.setPopupMenuName("ConstraintPopupMenu ");
```

5. Associate an action listener with popup menu items know which graphic object triggered the event.

The listener retrieve the context of the popup menu using an `IlvPopupMenuActivityContext` for activities or a `IlvPopupMenuConstraintContext` for constraints.

```
public void actionPerformed(ActionEvent e) {
    // retrieve the selected menu item
    JMenuItem m = (JMenuItem) e.getSource();

    // retrieve the graphic that has this popup menu
   IlvPopupMenuContext context = IlvPopupMenuManager.getPopupMenuContext
(m);
    if (context == null
        || !(context instanceof IlvPopupMenuActivityContext)) {
       return;
    }
```

```
    IlvPopupMenuActivityContext activityContext =
(IlvPopupMenuActivityContext) context;
    // retrieve the activity of the graphic
    IlvGeneralActivity activity = (IlvGeneralActivity) activityContext.
getActivity();
    //Do the action on this activity for this view.
  }
```

For information on popup menus, see `IlvPopupMenuContext` and `IlvPopupMenuManager` in the Java API Reference Manual.

`IlvSimplePopupMenu` is a subclass of `JPopupMenu` that allows you to configure popup menus easily. For information on configuring popup menus, see `IlvSimplePopupMenu`.

A sample that illustrates how to use popup menus with names defined in a CSS file can be found at:

<installdir>/jviews-gantt86/samples/extension

Resource Data charts

Explains how to handle rendering and interaction in the Resource Data chart.

In this section

The architecture of the Resource Data chart
Illustrates the main classes for handling the Resource Data chart.

The Resource Data chart bean
Describes the properties of the Resource Data chart bean and shows how to incorporate a Resource Data chart into your application. .

Comparing the Resource Data chart with IBM® ILOG® JViews Charts
Discusses the different methods used in `IlvChart` and `IlvResourceDataChart` to achieve the same tasks.

Computing and displaying resource data
Explains how to instantiate a Gantt data model implementation and bind it to a chart.

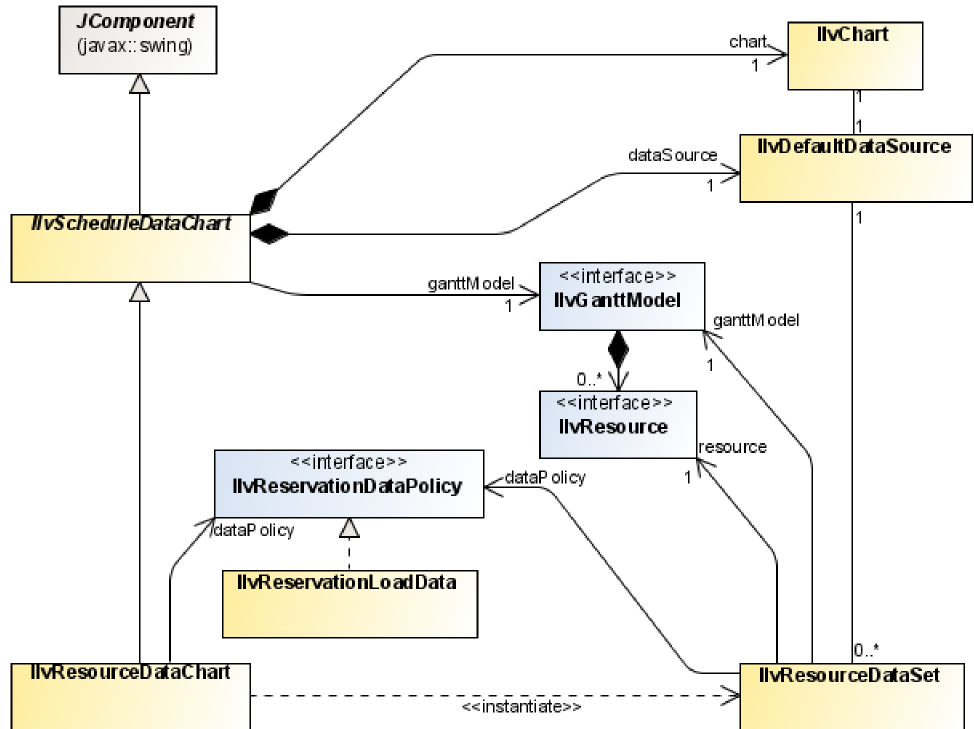
Synchronizing Schedule charts and Resource Data charts
Describes resource selection and display modes, internal chart data rendering and the relationship between time scales, x-grids and time scrolling.

© Copyright IBM Corp. 1987, 2009

The architecture of the Resource Data chart

The following figure shows the main classes for handling the Resource Data chart in the JViews Gantt API. The type of Resource Data chart shown is a load chart. This chart combines classes from the JViews Gantt and JViews Charts libraries. These classes are encapsulated by the high level Bean described in *The Resource Data chart bean*.

The following figure shows the classes for the Resource Data chart.



The Resource Data chart bean

Describes the properties of the Resource Data chart bean and shows how to incorporate a Resource Data chart into your application. .

In this section

Basic architecture
Describes the properties and architecture of the `IlvResourceDataChart` class.

Basic steps in using the Resource Data chart bean - details
Explains how to incorporate a Resource Data chart into the code of your application.

Basic architecture

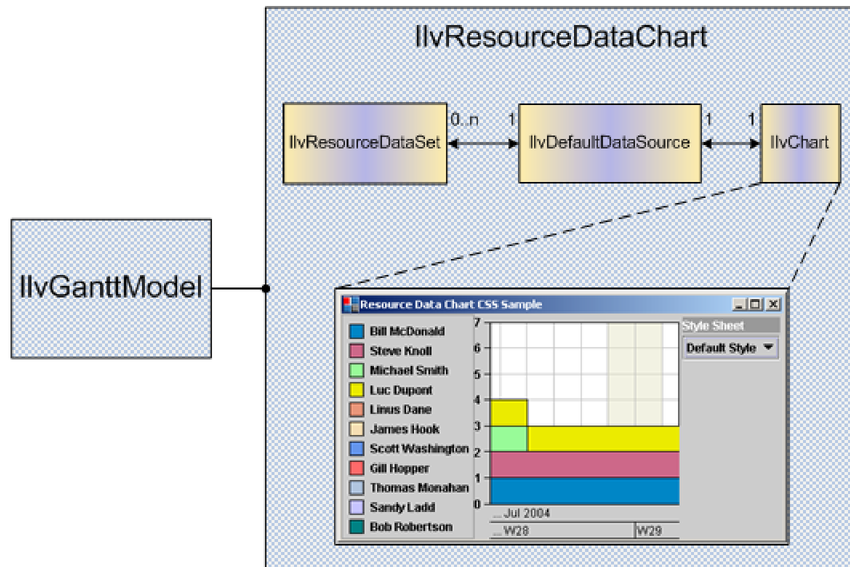
JViews Gantt features a high-level Bean, called Resource Data chart bean. Its API is based on the `IlvResourceDataChart` class, which is a subclass of `IlvScheduleDataChart`. The Bean encapsulates the Gantt and Charts libraries. Although the libraries can be used without the Bean, you will find it easier to rely on the Bean. Together with the `IlvGanttModel` interface, the Bean is the main class for handling Resource Data charts in the JViews Gantt API.

The Resource Data chart displays numerical information derived from the resources in a Gantt data model. The default data displayed by the chart is the number of activities reserved by a resource at each point in time. This data is called *resource loading*. The Resource Data chart displays the numerical information as a standard Cartesian chart, where the x-axis represents time. In this regard, the Resource Data chart provides an alternate view of the data contained in a Gantt data model and complements the displays provided by the Gantt chart and Schedule chart beans.

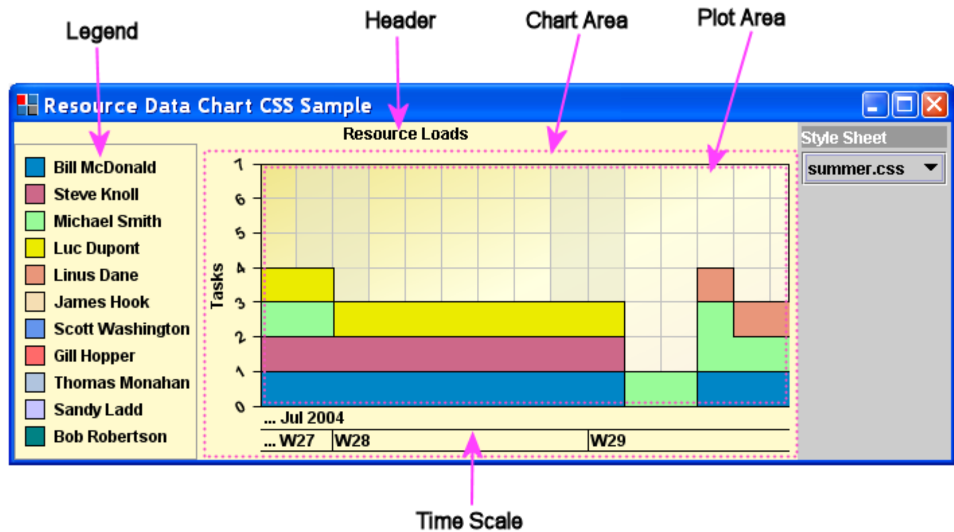
The Resource Data Chart utilizes the rendering capabilities of the `IlvChart` class from the Charts library. The `IlvChart` class is encapsulated by `IlvResourceDataChart`, which exposes a relevant subset of the complete API of `IlvChart`. Therefore, you may find it useful to review IBM® ILOG® JViews Charts Developing with the JViews Charts SDK, especially Introducing the Main Classes, for more detailed explanations of the JViews Charts architecture, the `IlvChart` class itself, and its related classes. This section assumes that you are familiar with the basic concepts and architecture of the JViews Charts library.

The basic architecture of the `IlvResourceDataChart` Bean is shown in the following figure with `IlvChart` expanded out to show a graphical representation of the chart.

The following figure shows the architecture of the Resource Data chart bean.



The following figure shows the chart in previous figure further expanded to show the subcomponents of the IlvChart.



As just mentioned, the Resource Data Chart encapsulates an IlvChart instance that provides the Cartesian chart rendering. The IlvChart is permanently bound to a single data source that is also encapsulated and is not accessible via the IlvResourceDataChart public API. The chart has a single chart renderer for its data source. The Resource Data Chart automatically creates an IlvDataSet implementation for each resource in the Gantt data model that is displayed. The datasets are instances of the IlvResourceDataSet class.

Basic steps in using the Resource Data chart bean - details

Explains what you need to do to incorporate a Resource Data chart into the code of your application.

To incorporate a Resource Data chart into your code:

1. To instantiate a Resource Data Chart and bind it to a Gantt data model, import the following packages as a minimum:

```
import ilog.views.gantt.*;
import ilog.views.schedule.*;
```

2. Create a Gantt data model that implements the `IlvGanttModel` interface.

   The data model should contain resources and reservations that will be displayed by the Resource Data chart. Refer to *Connecting to data* for detailed information on how to instantiate different Gantt data model implementations and connect to your business data:

```
IlvGanttModel model = ...
```

3. Create the Resource Data chart bean instance with the following line of code:

```
IlvResourceDataChart chart = new IlvResourceDataChart();
```

4. Bind the Resource Data chart bean to the data model to enable the chart to display the resource and reservation information of the data model.

```
chart.setGanttModel(model);
```

5. Customize the appearance and behavior of the chart by using the API of the `IlvResourceDataChart` class and its constituent components. For example, to set the time interval displayed by the x-axis to one week and add a header at the top of the chart, use the following code:

```
chart.setVisibleDuration(IlvDuration.ONE_WEEK);
chart.setHeaderText("My Chart");
```

6. Add the Resource Data chart bean to the user interface of your application in the same way as any other Swing component.

   For example, if your application uses standard *http://java.sun.com/javase/6/docs/api/javax/swing/JFrame.html* with a *http://java.sun.com/javase/6/docs/api/java/awt/BorderLayout.html*, you would add the chart to the center of the window like this:

```
JFrame appWindow = ...
appWindow.getContentPane().add(chart, BorderLayout.CENTER);
```

Comparing the Resource Data chart with IBM® ILOG® JViews Charts

Because the Resource Data chart encapsulates an IlvChart instance, much of the IlvResourceDataChart API and behavior is the same as that of IlvChart. Therefore, a basic understanding of the IBM® ILOG® JViews Charts SDK is necessary for understanding the Resource Data chart, and is assumed for this chapter. The primary areas of difference are in how data is bound to the charts for display and how time is displayed along the x-axis.

The following table compares the differences between the Resource Data chart and IBM® ILOG® JViews Charts.

Category	IlvChart	IlvResourceDataChart
Connecting to Data	Connect an IlvDataSource to an IlvChartRenderer Implementation. Then add the chart renderer to the chart.	Connect an IlvGanttModel implementation to the chart. Optionally, you can change the resources that are displayed by calling the setResourceDisplayMode and displayResource methods
Chart Types	Cartesian, polar, radar, or pie	Cartesian only
Renderers	Multiple, one for each data source.	One, connected to the single internal data source that represents the displayed resources.
X-Axis and Scale	IlvAxis and IlvScale	IlvTimeScale, default is IlvGanttTimeScale. IlvResourceDataChart implements the IlvTimeScrollable interface.
X Grid	IlvGrid	IlvGanttGridRenderer, default is IlvWeekendGrid.

Computing and displaying resource data

An `IlvResourceDataChart` displays numerical information derived from the resources and reservations contained in a Gantt data model.

Instantiating a Gantt data model and connecting to your business data

Refer to *Connecting to data* for detailed information on how to instantiate different Gantt data model implementations and how to connect to your business data.

Binding implementations to the chart

Once you have instantiated an `IlvGanttModel` implementation, you can bind it to the chart by using the following APIs:

The following table shows the APIs to bind an IlvGanttModel implementation to a chart.

Property	Methods
Data Model	`IlvGanttModel getGanttModel()`
	`void setGanttModel(IlvGanttModel model)`

Synchronizing Schedule charts and Resource Data charts

Describes resource selection and display modes, internal chart data rendering and the relationship between time scales, x-grids and time scrolling.

In this section

Overview
Explains where to find an example that illustrates Gantt data model synchronization.

Selecting resources for display
Explains resource selection and display modes.

Computing resource data
Explains how to compute multiple `IlvDataValue` instances using a `IlvReservationLoadData` object which is then assigned to your Resource Data chart.

Rendering resource data
Describes how internal chart data is rendered.

The x-axis
Explains how time scales, x-grids and time scrolling are related.

Overview

If your application also includes a Schedule chart, you have the option to synchronize the Gantt data models of both charts. Once the Schedule chart and the Resource Data chart are synchronized, changing the Gantt data model set on one chart will change the Gantt data model set on the other chart. For an example of this, you can look at the Resource Load chart sample, located in **<installdir>/jviews-gantt86/samples/resourceLoadChart**.

The following table shows the APIs to synchronize the data model of a Resource Data chart to that of a Schedule chart.

Property	Methods
Synchronizing the Data Model to a Schedule chart	void syncGanttModel(IlvScheduleChart scheduleChart)
	void syncGanttModel(IlvScheduleChart scheduleChart, int resourceDisplayMode)
	void unsyncGanttModel()

Selecting resources for display

By default, when you bind a data model to `IlvResourceDataChart`, the data for all leaf resources in the model are displayed. This binding is dynamic. So, if a leaf resource is added to the data model or if a parent resource becomes a leaf because all of its children are deleted, the resource will be added to the chart display. There are also several alternate modes that can be used to determine which resources from the Gantt data model are displayed in the chart.

The following table shows the APIs to control resources displayed in a chart.

Property	Methods
Resource Display Mode	`int getResourceDisplayMode()`
	`void setResourceDisplayMode(int mode)`
Manual Resource Display	`void clearAllDisplayedResources()`
	`void displayResource(IlvResource resource, boolean displayed)`
Accessing the Displayed Resources	`Iterator displayedResourcesIterator()`
	`boolean isDisplayed(IlvResource resource)`

Resource display modes - Schedule chart not synchronized

In the default case, when the Resource Data chart is not synchronized to a Schedule chart, you have a choice of three resource display modes:

- `IlvResourceDataChart.AUTO_RESOURCE_DISPLAY_DISABLED`: In this mode, resources in the data model are not automatically displayed by the Resource Data chart. In this mode, you must manually select resources to display by calling the `displayResource` method.
- `IlvResourceDataChart.DISPLAY_ALL_RESOURCES`: In this mode, all resources in the data model are automatically displayed by the Resource Data chart.
- `IlvResourceDataChart.DISPLAY_ALL_LEAVES`: This is the default resource display mode. In this mode, all leaf resources in the data model are automatically displayed by the Resource Data chart.

Resource display modes - Schedule chart synchronized

If you have synchronized the data model of the Resource Data chart to a Schedule chart by calling the `syncGanttModel` method, then you have a choice of three additional resource display modes.

These additional modes allow you to automatically display data in the Resource Data chart for resources that are selected in the Schedule chart:

- `IlvResourceDataChart.DISPLAY_SELECTED_RESOURCES`: In this mode, all resources that are selected in the Schedule chart are displayed in the Resource Data chart.

- `IlvResourceDataChart.DISPLAY_SELECTED_SUBTREES`: In this mode, the same as `DISPLAY_SELECTED_RESOURCES` mode, all resources that are selected in the Schedule chart are displayed in the Resource Data chart. In addition, when a parent resource is selected in the Schedule chart, all of its descendant resources are also displayed in the Resource Data chart.

- `IlvResourceDataChart.DISPLAY_SELECTED_LEAVES`: In this mode, all leaf resource that are selected in the Schedule chart are displayed in the Resource Data chart. For parent resources selected in the Schedule chart, their descendant leaf resources are displayed in the Resource Data chart. The selected parent resource itself is not displayed.

For an example of automatically displaying the resources selected in a Schedule chart, you can look at the Resource Load chart sample, located in:

<installdir>/jviews-gantt86/samples/resourceLoadChart

Computing resource data

Each resource that is selected for display is represented by a data series of numerical values along the y-axis, plotted against time along the x-axis.

The data series is computed by:

1. A list of `IlvDataValue`s, computed for each reservation assigned to the resource.
2. The lists of data values for the reservations of the resource, which are summed along the y-axis and merged along the x-axis to form the data series.

An instance of `IlvDataValue` represents a single numerical value at a specific date and time. Implementations of the `IlvReservationDataPolicy` interface are responsible for computing the list of data values for a single reservation. The default data policy is an instance of the `IlvReservationLoadData` class. This class performs a simple computation of the number of activities assigned to a resource versus time for a single reservation. Because a reservation represents the assignment of a single activity, `IlvReservationLoadData` computes a list of only two data values. The first is a value of 1 at the activity's start time, and the second is a value of 0 at the activity's end time. You can extend or create your own `IlvReservationDataPolicy` implementation that computes more complex data values from the reservations assigned to a resource.

The following table shows the APIs to set the policy on a Resource Data chart once you have created an `IlvReservationDataPolicy` object.

Property	Methods
Reservation Data Policy	`IlvReservationDataPolicy getReservationDataPolicy()`
	`void setReservationDataPolicy(IlvReservationDataPolicy dataPolicy)`

The Resource Data chart internally creates instances of `IlvResourceDataSet` that represent the data series of each resource. This class is responsible for summing the y-values of the data value lists computed by the reservation data policy and for merging the time values along the x-axis. The chart adds the resource data sets to its internal data source, thereby displaying the resource data.

Rendering resource data

The Resource Data chart uses a single `IlvChartRenderer` implementation to render the internal data source of the chart. This internal data source is automatically populated with a data set for each resource that is displayed by the chart. Therefore, the chart renderer should be a simple composite renderer, a subclass of `IlvSimpleCompositeChartRenderer`. Simple composite renderers handle a one-to-one relation between their child renderers and their data sets (that is, one child renderer per data set). The default renderer of the Resource Data chart is an instance of the `IlvStairChartRenderer` class.

The following table shows the APIs to set the renderer of an `IlvResourceDataChart`.

Property	Methods
Chart Renderer	`IlvChartRenderer getRenderer()`
	`void setRenderer(IlvChartRenderer renderer)`

See Handling Chart Renderers in Developing with the JViews Charts SDK of JViews Charts for details on the supplied chart renderer implementations and their graphical presentation.

The x-axis

The time scale

The Resource Data chart uses an `IlvTimeScale` to render the x-axis. The default time scale is an instance of `IlvGanttTimeScale`. You can follow the instructions in *Using the time scale* to customize or extend the time scale.

The following table shows the APIs to change the time scale of a Resource Data chart.

Property	Methods
Time Scale	`IlvTimeScale getTimeScale()`
	`void setTimeScale(IlvTimeScale timeScale)`

Note: The time scale is used as a renderer by the Resource Data chart. It is not added as a true subcomponent of the chart, as in the Gantt and Schedule charts. Therefore, the standard IlvDefaultScaleMouseInteractor does not work in the context of the Resource Data chart.

The x-grid

Like the Gantt and Schedule charts, the Resource Data chart uses an `IlvGanttGridRenderer` object to render the vertical grid of the x-axis. The default x-axis grid is an instance of `IlvWeekendGrid`.

The following table shows the APIs to Change the X-axis Grid of a Resource Data Chart.

Property	Methods
X Grid	`IlvGanttGridRenderer getXGrid()`
	`void setXGrid(IlvGanttGridRenderer grid)`

Time scrolling

Like the Gantt and Schedule charts, `IlvResourceDataChart` implements the `IlvTimeScrollable` interface.

The following table shows the methods to modify the time interval displayed by the chart.

Property	Methods
Time Scrolling	`Date getVisibleTime()`
	`void setVisibleTime(Date time)`
	`IlvDuration getVisibleDuration()`
	`void setVisibleDuration(IlvDuration duration)`
	`IlvTimeInterval getVisibleInterval()`
	`void setVisibleInterval(Date time, IlvDuration duration)`
	`Date getMinVisibleTime()`
	`void setMinVisibleTime(Date min)`
	`Date getMaxVisibleTime()`
	`void setMaxVisibleTime(Date max)`

For example, you can scroll a chart horizontally so that it displays from one week before the start of an activity until one week after the activity ends:

```
IlvActivity activity = ...
IlvTimeInterval interval = activity.getTimeInterval();
Date start = IlvTimeUtil.subtract(interval.getStart(),
                                  IlvDuration.ONE_WEEK))
IlvDuration duration = interval.getDuration();
Duration = duration.add(IlvDuration.ONE_WEEK.multiply(2));
myChart.setVisibleInterval(start, duration);
```

Calendar view components

Describes how to display activities from a Gantt data model on a monthly calendar or daily planner grid.

In this section

Calendar view beans

Describes the steps necessary to create calendar view beans.

Running the Calendar View sample

Explains how to run the Calendar View sample.

Basic architecture

Describes the properties of the Monthly and Daily Calendar View JavaBeans

© Copyright IBM Corp. 1987, 2009

Calendar view beans

JViews Gantt features two high-level JavaBeans, the Monthly Calendar View JavaBean and Daily Calendar View JavaBean. Their API is based on the IlvMonthView and IlvDayView classes. These JavaBeans encapsulate the JViews Gantt library. Although the library can be used without the JavaBeans, you will find it easier to rely on them. Together with the IlvGanttModel interface, the two JavaBeans make up the main classes for handling calendar views in the JViews Gantt API.

Calendar View JavaBeans display activities from a Gantt data model overlaid on a monthly calendar or daily planner grid. Monthly and Daily Calendar Views provide an alternative view of the data contained in a Gantt data model and complement the displays provided by the Gantt chart bean and Schedule chart bean.

A basic sample Java™ application, is provided to illustrate the basic steps needed to incorporate either a Monthly Calendar View or a Daily Calendar View into the code of your application. The source code of this sample can be found in:

<installdir>/jviews-gantt86/samples/calendarView/src/calendarView/CalendarViewExample.java

To incorporate either a Monthly Calendar View or a Daily Calendar View into the code of your application:

1. To instantiate a Monthly or Daily Calendar View and bind it to a Gantt data model, you need to import a minimum of the following packages:

```
import ilog.views.gantt.*;
import ilog.views.gantt.swing.calendarview.*;
```

2. Create a Gantt data model that implements the IlvGanttModel interface.

   The data model should contain activities to be displayed by the Calendar View. Refer to *Connecting to data* for detailed information on how to instantiate different Gantt data model implementations and connect to your business data:

```
IlvGanttModel model = ...
```

3. Create the Monthly and Daily Calendar View JavaBean instances with the following lines of code:

```
IlvMonthView monthView = new IlvMonthView();
IlvDayView dayView = new IlvDayView();
```

4. Bind the Calendar View JavaBeans to the data model to allow them to display the activity information of the data model:

```
monthView.setGanttModel(model);
dayView.setGanttModel(model);
```

5. Customize the appearance and behavior of the chart using the IlvMonthView and IlvDayView classes API. For example:

```
Date startTime = model.getRootActivity().getStartTime();
monthView.setDate(startTime);
```

```
dayView.setDate(startTime);
final IlvMonthPanel monthPanel = monthView.getMonthPanel();
monthPanel().addMouseListener(new MouseAdapter() {
  public void mousePressed(MouseEvent e) {
    Point point = e.getPoint();
    Calendar calendar = monthPanel.getCellDate(point);
    if (calendar != null) {
      ganttChart.setVisibleTime(IlvTimeUtil.subtract(calendar.getTime(),
 ganttChart.getVisibleDuration().divide(2)));
    }
  }
}
```

The code example above creates an interaction; when the mouse is pressed inside a Monthly Calendar View day cell, the Gantt chart is scrolled to center on that date.

6. Add the Calendar View beans to the user interface

The Monthly and Daily Calendar View Beans are standard Swing components that can be added to your application user interface in the same way as any other Swing component. For example, if your application uses a standard JFrame with a BorderLayout, you would add the chart to the center of the window like this:

```
JFrame appWindow = ...
appWindow.getContentPane().add(monthView, BorderLayout.CENTER);
```

Running the Calendar View sample

To find source code for using the capabilities of the Calendar View Beans in the Calendar View sample:

1. Open the file:

   ```
   <installdir>/jviews-gantt86/samples/calendarView
   ```

2. Follow the instructions to run the Calendar View sample.

   The source code of this sample can be found in:

   ```
   <installdir>/jviews-gantt86/samples/calendarView/src/calendarView/
   CalendarViewExample.java
   ```

Here you can find the information you need to incorporate the Calendar View Beans into the code of your application.

Basic architecture

Describes the properties of the Monthly and Daily Calendar View JavaBeans

In this section

Overview
Describes the properties of the Monthly and Daily Calendar View JavaBeans

Calendar View models
Explains how to synchronize a date displayed in multiple views by sharing calendar models or listening for calendar model events.

Calendar View renderers
Describes the API used to set renderers.

Leaf activity and holiday renderers
Describes the properties of the activity calendar renderer classes.

Milestone renderers
Describes the rendering of milestones in a Calendar Views component and explains how to customize them.

Overview

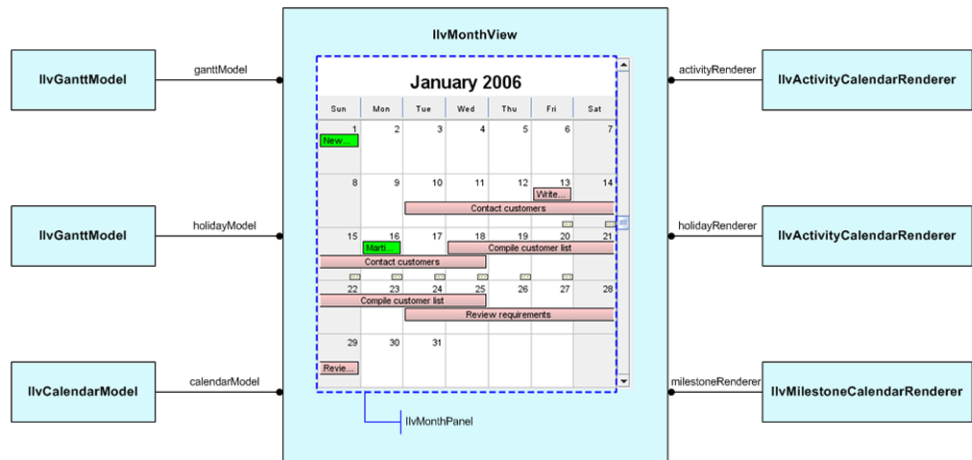
The Monthly and Daily Calendar View JavaBeans are Swing components used to display activities from a Gantt data model.

Both Calendar Views have a similar architecture:

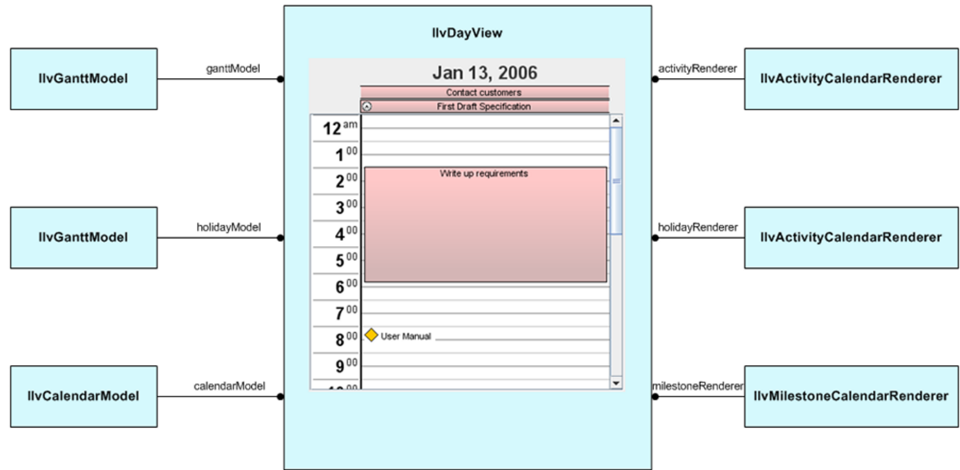
- ♦ They are both composed of basic Swing components.
- ♦ They both display activities and milestones from the Gantt data model to which they are bound.
- ♦ Both views can display holidays from optional Gantt data to which they are bound.
- ♦ The current displayed date is controlled by a calendar data model to which the views are bound.
- ♦ Easily customized renderer implementations determine the appearance of activities, milestones, and holidays.

The basic architecture of the Monthly and Daily Calendar View JavaBean is shown in this section, including the view subcomponents, model and renderer implementations that can be bound to a view.

The following figure shows a Calender JavaBean displaying a month view.



The following figure shows a Calender JavaBean displaying a day view.


IlvDayView
IlvGanttModel
ganttModel
Jan 13, 2006
Contact customers
First Draft Specification
activityRenderer
IlvActivityCalendarRenderer
Write up requirements
IlvGanttModel
holidayModel
holidayRenderer
IlvActivityCalendarRenderer
User Manual
IlvCalendarModel
calendarModel
milestoneRenderer
IlvMilestoneCalendarRenderer

Calendar View models

Daily and Monthly Calendar Views display the leaf activities and milestones of the Gantt data model to which they are bound. Parent or summary activities are not displayed by the views. Both views are automatically updated when activities in the data model are modified. The views also accept optional Gantt data model holiday activities that are rendered in green by default. Calendar Views are not automatically updated when holidays are modified. If a holiday activity has been modified, you must rebind the holiday model to the view. Calendar Views are bound to a calendar model that controls the currently displayed date. By sharing calendar models or listening for calendar model events, it is possible to synchronize the date displayed by multiple views.

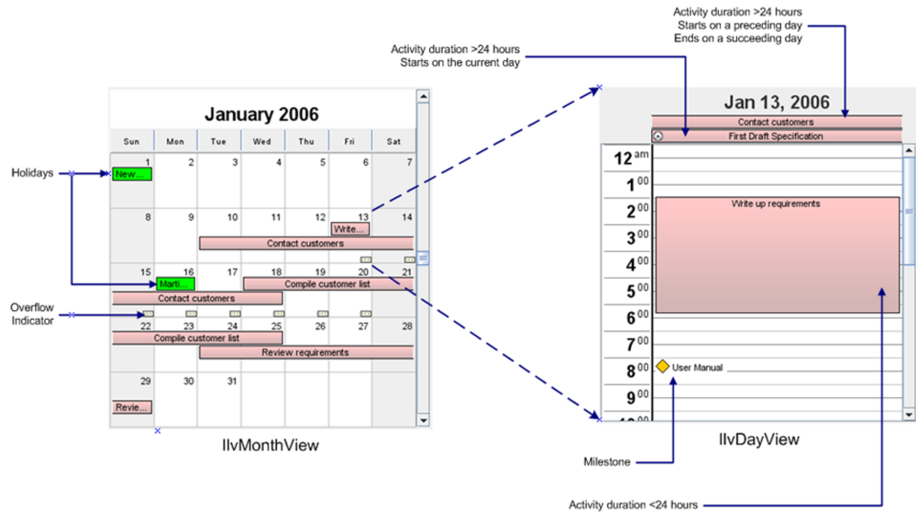
The following table shows the APIs to set the models used by `IlvMonthView` or `IlvDayView` objects.

Model	Methods
Gantt Data	`IlvGanttModel getGanttModel()`
	`void setGanttModel(IlvGanttModel ganttModel)`
Holidays	`IlvGanttModel getHolidayModel()`
	`void setHolidayModel(IlvGanttModel holidayModel)`
Calendar	`IlvCalendarModel getCalendarModel()`
	`void setCalendarModel(IlvCalendarModel calendarModel)`
	`Date getDate()`
	`void setDate(Date date)`

Calendar View renderers

The graphical representation of leaf activities and holidays for Calendar Views components is determined by implementations of the `IlvActivityCalendarRenderer` interface. The default activity and holiday renderer is the `IlvDefaultActivityCalendarRenderer` class. The graphical representation of milestones is determined by an implementation of the `IlvMilestoneCalendarRenderer` interface. The default milestone renderer is the `IlvDefaultMilestoneCalendarRenderer` class.

The following figure shows the appearance of the default renderers in the Calendar View Beans.



The following table shows the APIs to set the renderers for an `IlvMonthView` or `IlvDayView` object.

Renderer	Methods
Leaf Activities	`IlvActivityCalendarRenderer getActivityRenderer ()`
	`void setActivityRenderer (IlvActivityCalendarRenderer activityRenderer)`
Holidays	`IlvActivityCalendarRenderer getHolidayRenderer()`
	`void setHolidayRenderer (IlvActivityCalendarRenderer holidayRenderer)`
Milestones	`IlvMilestoneCalendarRenderer getMilestoneRenderer ()`
	`void setMilestoneRenderer (IlvMilestoneCalendarRenderer milestoneRenderer)`

Leaf activity and holiday renderers

The default rendering for leaf activities and holidays in Calendar Views components is performed by the IlvDefaultActivityCalendarRenderer class. This renderer draws a single rectangular bar representing an activity or holiday. For a Monthly Calendar View, when an activity spans multiple weeks, the renderer draws each weekly bar segment separately.

When a Calendar View draws a leaf activity or a holiday, it invokes the renderer's draw method for each rectangular region by calling draw:

```
void draw(Graphics g,
          IlvActivity activity,
          boolean isSelected,
          Rectangle rect,
          ComponentOrientation orientation,
          int startStyle,
          int endStyle)
```

The startStyle and endStyle parameters can be one of three possible values. Each value indicates the way in which the vertical ends of the rectangular region will be rendered. In a left-to-right component orientation, the startStyle represents the rectangle's left side rendering, the endStyle represents the rectangle's right side rendering.

Possible values for the style parameters are:

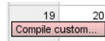
- IlvActivityCalendarRenderer.END_STYLE_CLOSED
- IlvActivityCalendarRenderer.END_STYLE_OPEN
- IlvActivityCalendarRenderer.END_STYLE_INTRA_DAY

The following sections explain the styles and their meanings and manipulation in greater depth.

IlvActivityCalendarRenderer.END_STYLE_CLOSED

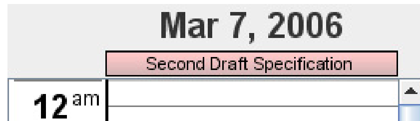
This style indicates that when an activity terminates on a day boundary, that the side of the rectangle will be drawn to indicate that it is “closed”. For example, in a Monthly Calendar View, an activity rendered with both vertical sides set to END_STYLE_CLOSED will look like the following figure.

The following figure show a closed multiple day activity in a Monthly Calendar View.



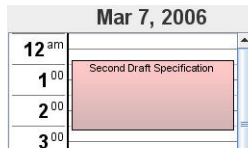
In a Daily Calendar View, an activity with a duration greater than or equal to 24 hours is displayed at the top of the view. When the activity starts or finishes on a day boundary, the vertical sides of the rectangle are shown to be closed.

The following figure shows a closed single day activity in a Daily Calendar View.



An activity that has a lasts less than 24 hours is displayed in an hourly grid in the Daily Calendar View. The vertical sides of such an activity are always closed.

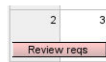
The following figure shows a closed activity lasting less than one day.



IlvActivityCalendarRenderer.END_STYLE_OPEN

This style indicate that the activity extends beyond the point denoted by that side of the activity rectangle. The side of the rectangle is drawn to indicate that it is "open". For example, in a Monthly Calendar View, an activity that starts before the current week will be rendered with an open `startStyle`.

The following figure shows the open startStyle for a Monthly Calendar View.



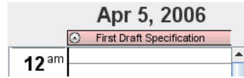
In a Daily Calendar View, an activity that has a duration greater than or equal to 24 hours is displayed at the top of the view. If the activity starts before the current day or terminates after the current day, the corresponding vertical sides of the rectangle are shown to be open.

The following figure shows the open startStyle and endStyle for a Daily Calendar View.

IlvActivityCalendarRenderer.END_STYLE_INTRA_DAY

This style indicates that the activity terminates within the bounds of a day. The side of the rectangle is drawn using the "closed" style. This also indicates that the activity does not lie on a day boundary. The default activity renderer uses a small clock image to indicate the intra-day style. In a Daily Calendar View, an activity that has a duration greater than or equal to 24 hours is displayed at the top of the view. If the activity starts within the current day, but not on the day boundary, it is rendered using the intra-day `startStyle`.

The following figure shows the intra-day startStyle for a Daily Calendar View.



APIs for customizing the appearance of the default activity renderer

The following table shows the APIs are used to customize the appearance of the default activity renderer.

Property	Methods
Stroke	`Stroke getStroke()`
	`void setStroke(Stroke stroke)`
	`Paint getStrokePaint()`
	`void setStrokePaint(Paint paint)`
Fill	`Paint getFillPaint()`
	`void setFillPaint(Paint paint)`
Label	`Font getLabelFont()`
	`void setLabelFont(Font font)`
	`Paint getLabelPaint()`
	`void setLabelPaint(Paint paint)`
	`int getLabelMargin()`
	`void setLabelMargin(int margin)`
Selection	`Paint getSelectionStrokePaint()`
	`void setSelectionStrokePaint(Paint paint)`
	`Paint getSelectionFillPaint()`
	`void setSelectionFillPaint(Paint paint)`
Clock Image	`Image getClockImage()`
	`void setClockImage(Image image)`

Milestone renderers

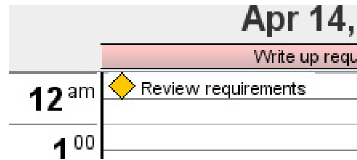
The default rendering of milestones in a Calendar Views component is performed by the `IlvDefaultMilestoneCalendarRenderer` class. The renderer is responsible for drawing a labeled symbol representing the milestone. When a calendar view wants to draw a milestone, the renderer's draw method is invoked:

```
void draw(Graphics g,
          IlvActivity milestone,
          boolean isSelected,
          Rectangle rect,
          Color background,
          ComponentOrientation orientation)
```

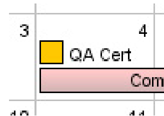
The following figures show that the default milestone renderer supports four predefined symbol shapes:

- `MILESTONE_CIRCLE`

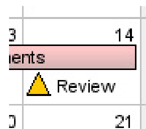
- `MILESTONE_DIAMOND`



- `MILESTONE_SQUARE`



- `MILESTONE_TRIANGLE`



The following table shows the APIs that can be used to customize the appearance of the default milestone renderer.

Property	Methods
Symbol	int getSymbol()
	void setSymbol(int symbol)
Stroke	Stroke getStroke()
	void setStroke(Stroke stroke)
	Paint getStrokePaint()
	void setStrokePaint(Paint paint)
Fill	Paint getFillPaint()
	void setFillPaint(Paint paint)
Label	Font getLabelFont()
	void setLabelFont(Font font)
	Paint getLabelPaint()
	void setLabelPaint(Paint paint)
	int getLabelMargin()
	void setLabelMargin(int margin)
Selection	Paint getSelectionStrokePaint()
	void setSelectionStrokePaint(Paint paint)
	Paint getSelectionFillPaint()
	void setSelectionFillPaint(Paint paint)

Deploying as an applet

With a deployed applet, users can view the displayed data and edit it but cannot save any changes.

The other two ways of deploying an JViews Gantt application are:

- ♦ Swing application, see Writing an application in *Using the Designer*.
- ♦ Thin client deployed over the Web, see Deploying an application as a DHTML-only thin client.

Creating an applet

The `IlvGanttChart`, `IlvScheduleChart`, and `IlvResourceDataChart` instances and their associated Beans can be used in applets in exactly the same way as in Swing applications. The only limitations are the restricted permissions in applets, for example, it is not possible to save data files from an applet.

The `AbstractExample` class found in **<installdir>/jviews-gantt86/samples/ganttChart/src/shared/AbstractExample.java** extends the Swing JApplet class and is the base class for all of the JViews Gantt samples in **<installdir>/jviews-gantt86/samples**, except for the `servlet` and `thinclient` samples. Therefore, any of these samples can be used directly in an HTML applet tag. For an example of an applet application, see the Gantt chart sample in:

<installdir>/jviews-gantt86/samples/ganttChart

 © Copyright IBM Corp. 1987, 2009

Using JViews products in Eclipse RCP applications

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic describes how to use JViews TGO inside Eclipse or RCP. It shows you how to display network, equipment, table, and tree components embedded in an SWT window.

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic shows you how to display diagrams and dashboards embedded in an SWT window.

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic shows you how to display `IlvGanttChart` or `IlvScheduleChart` objects in an SWT window, together with other SWT or JFace controls.

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse™ development environment and the Eclipse Rich Client Platform (RCP). This topic shows you how to display charts embedded in an SWT window.

Installing the JViews runtime plugin

JViews provides an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. It allows you to use `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable` objects in an SWT window, together with other SWT or JFace controls. In this way, it provides a bridge between the AWT/Swing windowing system and the SWT windowing system.

IBM® ILOG® JViews Gantt provides jar files in the form of a pre-packaged Eclipse™ plugin. The name of this package is `ilog.views.eclipse.gantt.runtime`.

IBM® ILOG® JViews Maps JViews Maps for Defense provides jar files in the form of a pre-packaged Eclipse plugin. The name of this package is `ilog.views.eclipse.maps.runtime`.

IBM® ILOG® JViews Diagrammer provides jar files in the form of a pre-packaged Eclipse plugin. The name of this package is `ilog.views.eclipse.diagrammer.runtime`.

IBM® ILOG® JViews Charts provides jar files in the form of a pre-packaged Eclipse plugin. The name of this package is `ilog.views.eclipse.chart.runtime`.

In order to install the IBM® ILOG® JViews Eclipse plugins, you need to install from the local site as shown below.

For Eclipse 3.3:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates/Find And Install**.
3. In the `Install/Update` dialog box, click **Search for new features to install**.
4. Define a New Local Site with the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`.
5. Select the features you want to install.

For Eclipse 3.4:

1. Launch your Eclipse installation.
2. Go to **Help/Software Updates** and select the **Available Software** tab.
3. Add a new local site: Click **Add Site**, then **Local** and specify the directory `<installdir>/jviews-framework86/tools/ilog.views.eclipse.update.site`
4. Select the features you want to install, and press the **Install** button.

This installation also installs some examples. See Installing and using Eclipse samples for more information.

In your applications, you need the `ilog.views.eclipse.gantt.runtime` plugin and its dependencies:

- `ilog.views.eclipse.gantt.runtime`
- `ilog.views.eclipse.chart.runtime`
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

In your applications, you need the `ilog.views.eclipse.maps.runtime` plugin and its dependencies:

- `ilog.views.eclipse.maps.runtime`
- `ilog.views.eclipse.diagrammer.runtime` (optional)
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

In your applications, you need the `ilog.views.eclipse.maps.runtime` plugin and its dependencies:

- `ilog.views.eclipse.maps.runtime`
- `ilog.views.eclipse.maps.defense.runtime`
- `ilog.views.eclipse.diagrammer.runtime` (optional)
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

This installation also installs some examples. See Installing and using Eclipse samples for more information.

In your applications, you need the `ilog.views.eclipse.chart.runtime` plugin and its dependencies:

- `ilog.views.eclipse.chart.runtime`
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

This installation also installs some examples. See Installing and using Eclipse samples for more information.

In your applications, you need the `ilog.views.eclipse.diagrammer.runtime` plugin and its dependencies:

- `ilog.views.eclipse.diagrammer.runtime`
- `ilog.views.eclipse.framework.runtime`
- `ilog.views.eclipse.utilities.runtime`

Providing access to class loaders

Many services in JViews need to look up a resource. Since the classical way to provide access to resources is a classloader, JViews uses classloaders for this purpose. But in Eclipse/RCP applications, each plugin corresponds to a classloader, and the JViews classloader sees only its own resources, not the application resources. To fix this problem, you can register plugin classloaders with JViews through the `IlvClassLoaderUtil.registerClassLoader` function. Each resource lookup then considers the registered classloaders and, if the plugins are configured accordingly, also considers the dependencies of the registered classloaders.

The code for doing this is usually located in a plugin activator class. For example:

```
public class MyPluginActivator extends AbstractUIPlugin
{
    /**
     * This method is called upon plugin activation
     */
    public void start(BundleContext context) throws Exception {
      super.start(context);
      IlvClassLoaderUtil.registerClassLoader(getClass().getClassLoader());
    }

    /**
     * This method is called when the plugin is stopped
     */
    public void stop(BundleContext context) throws Exception {
      super.stop(context);
      IlvClassLoaderUtil.unregisterClassLoader(getClass().getClassLoader());
    }

  }
```

The overriding of `stop()` is necessary so that, when the plugin gets unloaded, JViews gets notified about the plugin that is going to stop and can drop references to its resources or instances of its classes. The activator plugin is usually also the place where `IlvProductUtil.registerApplication` is called. See section Before you start deploying an application for an example.

The bridge between AWT/Swing and SWT

The bridge between the AWT/Swing windowing system and the SWT windowing system consists of an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT

widget. This class allows you to use `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable` objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object:

```
Composite parent = ...;
IlpNetwork network = new IlpNetwork();
ControlSWTnetwork = new IlvSwingControl(parent, SWT.NONE, network);
```

The bridge between the AWT/Swing windowing system and the SWT windowing system consists of an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. This class allows you to use `IlvDiagrammer`, `IlvJScrollManagerView`, or `IlvJManagerViewPanel` objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object:

```
Composite parent = ...;
IlvDiagrammer diagrammer = new IlvDiagrammer();
ControlSWTdiagrammer = new IlvSwingControl(parent, SWT.NONE, diagrammer);
```

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse development environment and the Eclipse Rich Client Platform (RCP).

JViews provides an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. It allows you to use `IlvChart` or `IlvLegend` objects in an SWT window, together with other SWT or JFace controls. In this way, it provides a bridge between the AWT/Swing windowing system and the SWT windowing system.

The following code shows how to create a bridge object:

```
Composite parent = ...;
IlvChart chart = new IlvChart();
ControlSWTchart = new IlvSwingControl(parent, SWT.NONE, chart);
```

The Standard Widget Toolkit (SWT) is the window toolkit of the Eclipse development environment and the Eclipse Rich Client Platform (RCP).

JViews provides an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. It allows you to use `IlvGanttChart` objects in an SWT window, together with other SWT or JFace controls. In this way, it provides a bridge between the AWT/Swing windowing system and the SWT windowing system.

At the JViews Framework level, the bridge between the AWT/Swing windowing system and the SWT windowing system consists of an `IlvSwingControl` class that encapsulates a Swing JComponent in an SWT widget. This class allows you to use `IlvManager` or `IlvJManagerViewPanel` objects in an SWT window, together with other SWT or JFace controls.

The following code shows how to create a bridge object at the JViews Framework level:

```
Composite parent = ...;
IlvManagerView mgrView = ...;
IlvJManagerViewPanel jmgrView = new IlvJManagerViewPanel(mgrView);
ControlSWTview = new IlvSwingControl(parent, SWT.NONE, jmgrView);
```

Using `IlvSwingControl` instead of the native SWT_AWT class has the following benefits:

- Simplicity: it is easier to use, since you do not have to worry about the details of the `Component` hierarchy (see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*).
- Portability: `IlvSwingControl` also works on platforms that do not have SWT_AWT, like X11/Motif® and MacOS® X 10.4.
- Less flickering: on Linux®/Gtk, flickering is reduced.
- Popup menus: popup menus can be positioned on each `Component` inside the AWT component hierarchy. For details of components, see *http://java.sun.com/javase/6/docs/api/java/awt/Component.html*.
- Better size management: the size management between SWT and AWT (`LayoutManager`) is integrated.
- Focus: it provides a workaround for a focus problem on Microsoft® Windows® platforms.

Note: The `IlvSwingControl` bridge is not supported on all platforms. It is only supported on Windows, UNIX® with X11 (Linux, Solaris™, AIX®, HP-UX®), and MacOS X 10.4 or later.

The `IlvSwingControl` bridge does not support arbitrary JComponents. Essentially, components that provide text editing are not supported. See `IlvSwingControl` for a precise description of the limitations.

Threading modes

You can handle the SWT-Swing user interface events in one or two threads.

Note: Single-thread mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT `Dialogs`, Swing `JDialogs`, or modal `JInternalFrames` in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

- Two-thread mode

  The SWT events are handled in the SWT event thread and AWT/Swing events are handled in the AWT/Swing event thread. This is the default mode.

  You can switch between the two threads by using the SWT method `Display.asyncExec()` and the AWT method `EventQueue.invokeLater()`.

  If your application uses this mode, you must be careful to:

  - Make API calls on SWT widgets only in the SWT event thread. Otherwise, you will get `SWTExceptions` of type `ERROR_THREAD_INVALID_ACCESS`.
  - Make API calls on JComponents, which include `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

Make API calls on JComponents, which include `IlvDiagrammer`, `IlvJScrollManagerView`, and `IlvJManagerViewPanel`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

Make API calls on JComponents, which include `IlvChart` and `IlvLegend`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

You can switch between the two threads by using the SWT method `Display.asyncExec()` and the AWT method `EventQueue.invokeLater()`.

Make API calls on JComponents, which include , only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

At the JViews Framework level, make API calls on JComponents, which include `IlvManager` and `IlvJManagerViewPanel`, only in the AWT/Swing event thread. Otherwise, you risk deadlocks.

♦ Single-thread mode

In single-thread mode, SWT and AWT/Swing events are handled in the same thread.

Single-thread mode reduces the risk of producing deadlocks.

Enable this mode by calling `setAWTThreadRedirect` or `enableAWTThreadRedirect()` early during initialization.

The following example shows how to enable single-thread mode:

```
// Switch single-event-thread mode during a static initialization.
    static {
        IlvEventThreadUtil.enableAWTThreadRedirect();
    }
```

If you are using JComponents other than `IlpNetwork`, `IlpEquipment`, `IlpTree`, and `IlpTable` in your application, your JComponents must use the method `IlvSwingUtil.isDispatchThread()` rather than *`EventQueue.isDispatchThread()`* or *`SwingUtilities.isEventDispatchThread()`*.

For example:

```
// Switch single-event-thread mode during a static initialization.
   static {
       IlvEventThreadUtil.enableAWTThreadRedirect();
   }
```

Note: This mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT `Dialogs`, Swing `JDialogs`, or modal `JInternalFrames` in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

If you are using JComponents other than `IlvDiagrammer`, `IlvJScrollManagerView`, and `IlvJManagerViewPanel` in your application, your JComponents must use the method `isDispatchThread()` rather than *`EventQueue.isDispatchThread()`* or *`SwingUtilities.isEventDispatchThread()`* .

This mode reduces the risk of producing deadlocks. If you are using JComponents other than `IlvChart` and `IlvLegend` in your application, your JComponents must use the method `isDispatchThread()` rather than *`EventQueue.isDispatchThread()`* or *`SwingUtilities.isEventDispatchThread()`*.

Note: This mode is incompatible with AWT/Swing Dialogs. If you use single-thread mode, you cannot use AWT Dialogs, Swing JDialogs, or modal JInternalFrames in your application. There are also some other limitations. See the class `IlvEventThreadUtil` for a precise description of the limitations.

This mode reduces the risk of producing deadlocks. If you are using JComponents other than `IlvGanttChart` in your application, your JComponents must use the method `isDispatchThread()` rather than *http://java.sun.com/javase/6/docs/api/java/awt/EventQueue.html#isDispatchThread()* or *http://java.sun.com/javase/6/docs/api/javax/swing/SwingUtilities.html#isEventDispatchThread()*.

At the JViews Framework level, if you are using JComponents other than `IlvManager` and `IlvJManagerViewPanel` in your application, your JComponents must use the method `isDispatchThread()` rather than `EventQueue.isDispatchThread()` (see *http://java.sun.com/javase/6/docs/api/java/awt/EventQueue.html#isDispatchThread()*) or `SwingUtilities.isEventDispatchThread()` (see *http://java.sun.com/javase/6/docs/api/javax/swing/SwingUtilities.html#isEventDispatchThread()*.)

Printing

Describes how JViews Gantt provides APIs that allow you to print Gantt, Schedule and Resource Data charts for a single or multi-page document without having to scroll the user interface.

In this section

Printing Gantt and Schedule charts
Describes the main classes in the printing API and explains how to use them.

Printing a Resource Data chart
Explains how to use the printing APIs.

© Copyright IBM Corp. 1987, 2009

Printing Gantt and Schedule charts

Describes the main classes in the printing API and explains how to use them.

In this section

Overview
Explains the relation between the classes in the JViews Gantt API.

Introduction
Describes the main classes in the JViews Gantt printing framework and explains the relationship between the classes.

The GanttPrintExample demo
Explains how to run and use this sample application.

Printing Framework API
Describes the properties of the main classes in the Resource Data chart printing framework.

How it works
Explains how to initiate and process a printing task.

Overview

The `IlvGanttChart` and `IlvScheduleChart` classes are UI components designed to display your projects on screen. To distribute and to exchange the projects, you may need to print the projects on paper. You may also need to print not only the visible part of the projects but also the part that is not visible.

JViews Gantt provides APIs that allow you to print the Gantt or Schedule charts in a document (single or multiple pages) without scrolling the UI. These APIs collectively are referred to as the JViews Gantt printing framework.

> **Note**: Before reading this section you should familiarize yourself with sections The generic printing framework and *Printing framework for manager content* of *Advanced Features of IBM® ILOG® JViews Framework*.

Introduction

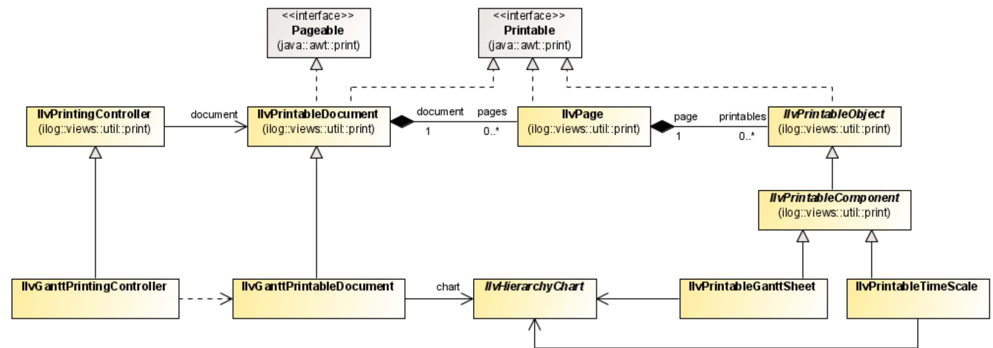
The JViews Gantt printing framework extends the basic IBM® ILOG®JViews Framework printing framework to add support to `IlvGanttChart` and `IlvScheduleChart` objects. The generic classes of the `ilog.views.util.print` package have been subclassed to handle specific Gantt properties.

These classes are:

- `IlvGanttPrintingController`: A Gantt printing controller controls the printing process.
- `IlvGanttPrintableDocument`: A Gantt printable document defines the printing configuration and contains the Gantt data you want to print in a set of pages.
- `IlvPrintableGanttSheet`: A Gantt sheet printable object is used to print a portion of an `IlvGanttSheet` object.
- `IlvPrintableTimeScale` : A time scale printable object is used to print a portion of an `IlvTimeScale`.

For information on how to use these classes, see *Printing Framework API*.

The following figure shows the associations between classes for printing a Gantt chart.



The GanttPrintExample demo

The GanttPrintExample demo is a simple example for printing a Gantt chart.

To run the demo:

1. Open the file:

   <installdir>/jviews-gantt86/samples/print

2. Follow the instructions to run the example given in the `print` demo.

   The source code of this example can be found in :

   <installdir>/jviews-gantt86/samples/print/src/print/GanttPrintExample.java

To use Gantt printing:

1. Create an instance of an `IlvGanttPrintingController`:

```
IlvGanttChart gantt = ...;
IlvGanttPrintingController printController =
            new IlvGanttPrintingController(gantt);
```

2. Invoke on that instance the action you want to see performed, such as `print(boolean)`, `setupDialog(java.awt.Window, boolean, boolean)`, or `printPreview(java.awt.Window)` as shown here:

```
printController.printPreview((java.awt.Frame)gantt.getTopLevelAncestor()
);
```

The following figure shows the result of invoking the `printPreview` method:

Print Preview

Previous | Next | Print... | Setup... | 75 % | Margins | Close

Preview: Page 1 of 2

Printing Framework API

The following classes are involved in the JViews Gantt printing framework:

- *IlvGanttPrintableDocument*
- *IlvGanttPrintingController*
- *IlvPrintableGanttSheet*
- *IlvPrintableTimeScale*

IlvGanttPrintableDocument

The Gantt printable document stores the printed document structure and defines a set of parameters to customize the printing (the printed data window, which part of the Gantt is printed, how the Gantt fits on the page, and so on). The printable document is responsible for creating and populating the pages.

The following table shows the different properties you can customize for printing.

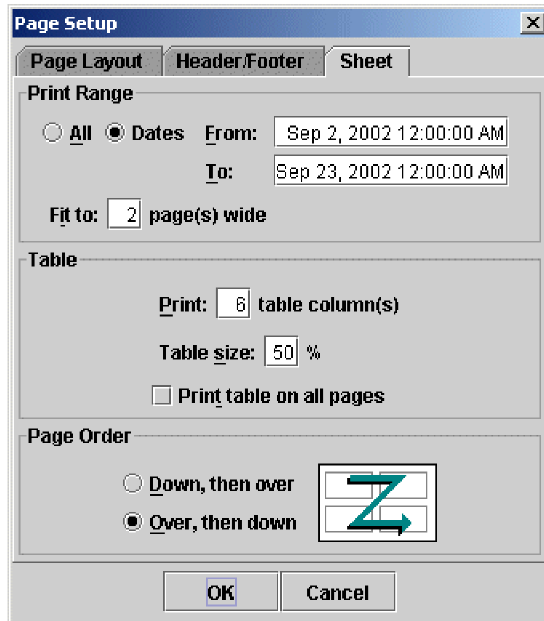
Property	Description
Divider position	Defines the position on the page that separates the table and the Gantt sheet (the value must be between 0 and 1).
Number of pages per band	The JViews Gantt printing framework provides support for multipage printing through the “pages per band” property. This represents the number of horizontal pages you want printed between the start and the end date.
Repeat table	Indicates whether the table should be printed repeatedly on every page.
Start date	The start date of the first printed page. This defines the beginning of the printed data.
End date	The end date for the last printed page in a band. This defines the end of the printed data.
Number of columns of the table	Indicates the number of table columns to be printed.

Note: All these properties are also accessible from the JViews Gantt Print Setup dialog box, which you can invoke by calling the method setupDialog on the `IlvGanttPrintingController` instance.

The following table summarizes the `IlvGanttPrintableDocument` properties.

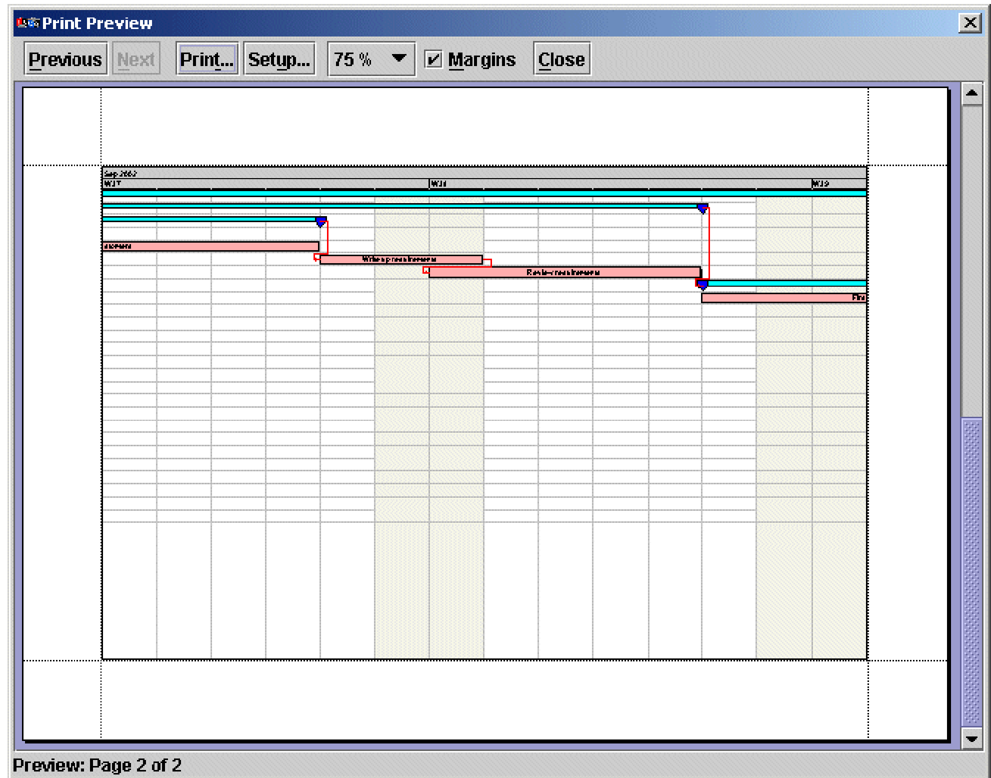
Property	Methods	Default value when automatically created	Field in JViews Gantt Designer
Divider position	`getDividerPosition()` `setDividerPosition(double)`	0.5	Table, Table size
End date of the last page in a band	`getEnd()` `setEnd(java.util.Date)`	The chart visible time + the chart visible duration	Print Range, Dates, To
Number of pages per band	`getPagesPerBand()` `setPagesPerBand(int)`	2	Print Range, Fit to
Whether the table is printed repeatedly on every page	`getRepeatTable()` `setRepeatTable(boolean)`	False	Table, Print table on all pages
Start date of the first page	`getDividerPosition()` `setStart(java.util.Date)`	The chart visible time	Print Range, Dates, From
Number of columns of the table	`getTableColumnCount()` `setTableColumnCount(int)`	The chart table column count	Table, Print *<n>* table column(s)

The following figure shows the default print settings.



With these settings, you get two pages in the Print Preview window. You can see that the table, with its six columns, is only displayed on the first page occupying 50% of the page size.

The following figure shows the second page of the Print Preview dialog.



IlvGanttPrintingController

The printing controller controls the printing process. It initiates the printer job, handles the Setup and Preview dialog boxes, and configures the document accordingly.

The configuration of the document can be done:

- Automatically, by using the following constructor:

  ```
  public IlvGanttPrintingController(IlvHierarchyChart chart)
  ```

  In this case, a printable document is created with the pages oriented in landscape, two pages per band, and all the columns of the Gantt table printed on the first page only (occupying half of the page size). See *Examples*.

- Through code, by creating a Gantt printable document and setting the parameters as described in section *IlvGanttPrintableDocument*.

IlvPrintableGanttSheet

An instance of the class `IlvPrintableGanttSheet` represents the concrete Gantt sheet object that can be printed. It extends the `IlvPrintableComponent` class, which itself is a

subclass of IlvPrintableObject. IlvPrintableGanttSheet lets you print the IlvGanttSheet within a region of the printable area of an IlvPage.

The way the Gantt fills the region is determined by the different document properties. Consequently, you do not need to use the IlvPrintableGanttSheet class directly if you want to print using the parameters provided by the Setup dialog box. You will need this class only if you want to control your IlvPrintableDocument object by creating pages and adding your own IlvPrintableObject instances.

For information on IlvPrintable, IlvPrintableDocument, and IlvPrintableObject, see The generic printing framework in *Advanced Features of IBM® ILOG® JViews Framework*.

IlvPrintableTimeScale

An instance of the class IlvPrintableTimeScale represents the concrete time scale object that can be printed. It also extends IlvPrintableComponent, which, as you saw in *IlvPrintableGanttSheet*, is a subclass of IlvPrintableObject. IlvPrintableTimeScale lets you print a portion of the IlvTimeScale object within an IlvPage.

The way the Gantt fills the region is determined by the different document properties. Consequently you do not need to use the IlvPrintableTimeScale class directly if you want to print using the parameters provided by the Setup dialog box. You will need this class only if you want to control your IlvPrintableDocument by creating pages and adding your own IlvPrintableObject instances.

For information on IlvPrintableDocument, IlvPrintableComponent, and IlvPrintableObject, see The generic printing framework in *Advanced Features of IBM® ILOG® JViews Framework*.

How it works

A printing task is initiated and processed by an `IlvGanttPrintingController` instance:

- by code, via the method `print(boolean)`, or
- from a GUI request using the Setup or Preview dialog, via the methods:

  ```
  IlvPrintingController.printPreview(Window)
  ```

  and

  ```
  IlvPrintingController.setupDialog(Window, boolean, boolean)
  ```

When a printing task is initiated, the document associated with the printing controller is prepared for printing: pages are initialized with the printable objects and added to the document.

> **Note**: See section *IlvGanttPrintingController* for a description of how a document is associated with the `IlvGanttPrintingController`.

This section covers:

- *Handling pages*
- *Populating a page*

Handling pages

Pages of a Gantt document are instances of the `IlvPage` class. They handle a collection of printable objects, instances of `IlvPrintableObject`.

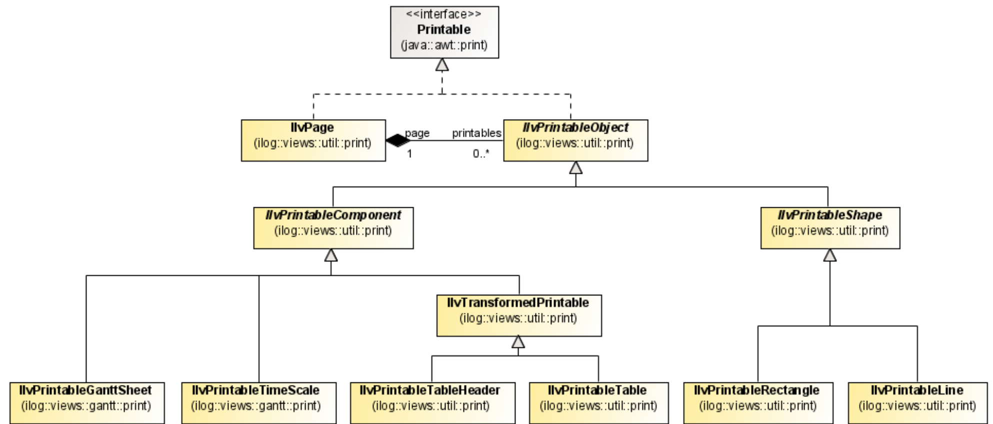
Populating a page

Pages created by an `IlvGanttPrintableDocument` are populated in the `createPages()` method. You may override the `createPages` method if you want to add additional printable objects to the page.

The `createPages` implementation uses the following printable objects:

- ilog.views.util.print. `IlvPrintableTableHeader`, if there are columns to print
- ilog.views.util.print. `IlvPrintableTable`, if there are columns to print
- ilog.views.gantt.print. `IlvPrintableTimeScale`
- ilog.views.gantt.print. `IlvPrintableGanttSheet`
- ilog.views.util.print. `IlvPrintableRectangle`

The following figure shows the associations between classes for populating a page.


<<interface>>
Printable
(java::awt::print)
IlvPage
(ilog::views::util::print)
page
1
printables
0..*
IlvPrintableObject
(ilog::views::util::print)
IlvPrintableComponent
(ilog::views::util::print)
IlvPrintableShape
(ilog::views::util::print)
IlvTransformedPrintable
(ilog::views::util::print)
IlvPrintableGanttSheet
(ilog::views::gantt::print)
IlvPrintableTimeScale
(ilog::views::gantt::print)
IlvPrintableTableHeader
(ilog::views::util::print)
IlvPrintableTable
(ilog::views::util::print)
IlvPrintableRectangle
(ilog::views::util::print)
IlvPrintableLine
(ilog::views::util::print)

Printing a Resource Data chart

Explains how to use the printing APIs.

In this section

Introduction
Describes the APIs provided for printing the Resource Data chart in a single or multi-page document.

The Printing Resource Data chart Example
Describes how to run and use the example.

Printing framework API
Describes the properties of the main classes in the Resource Data chart printing framework.

Introduction

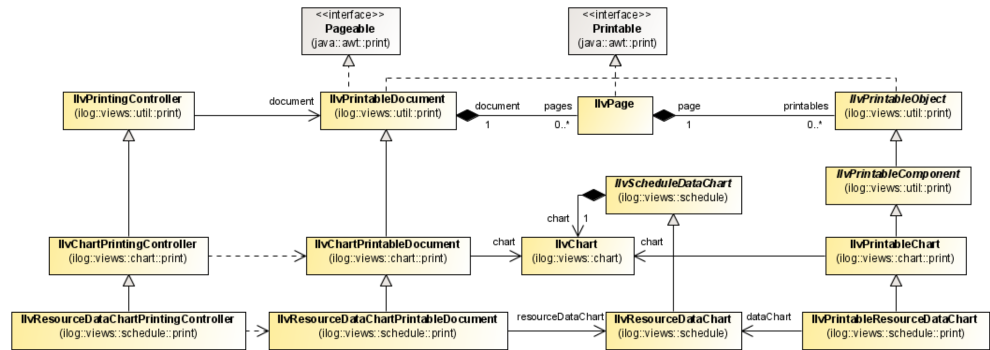
The `IlvResourceDataChart` class is a UI component designed to display numerical information derived from resources in an on screen Gantt data model. To distribute and exchange this numerical information, you may need to print the Resource Data chart on paper. JViews Gantt provides APIs for printing the Resource Data chart in a single or multi-page document without having to scroll the user interface.

The IBM® ILOG® JViews Resource Data chart printing framework extends the basic IBM® ILOG® JViews Charts printing framework to add support to the `IlvResourceDataChart` object.

The generic classes of the `ilog.views.schedule.print` have been subclassed to handle specific Resource Data chart properties. These classes are the following:

- `IlvResourceDataChartPrintingController`: Controls the printing process.
- `IlvResourceDataChartPrintableDocument`: Defines the printing configuration and contains the Resource Data chart data you want to print in a set of pages.
- `IlvPrintableResourceDataChart`: Used to print a part of an `IlvResourceDataChart` object.

The following figure shows the associations between classes for printing a Resource Data chart.



The Printing Resource Data chart Example

The Printing Resource Data chart Example is a simple sample of printing a Resource Data chart.

To run this example:

1. Open the file:

   <installdir>/jviews-gantt86/samples/printResourceData

2. Follow the instructions to run the Resource Data chart Printing sample.

   The source code of this sample can be found in:

   <installdir>/jviews-gantt86/samples/printResourceData/src/print/ResourceDataPrintActions.java

To print the Resource Data chart:

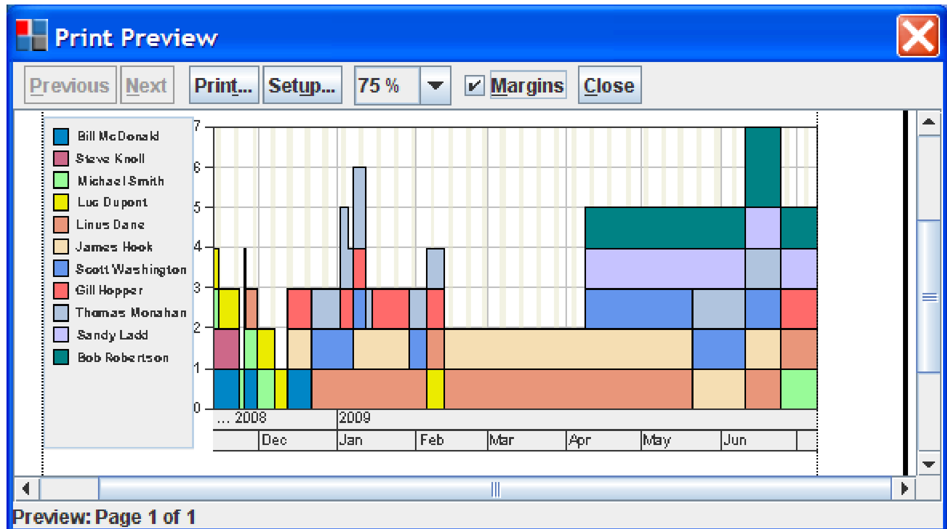
1. Create an instance of an `IlvResourceDataChartPrintingController`:

```
IlvResourceDataChart chart = ...;
IlvResourceDataChartPrintingController printController =
    new IlvResourceDataChartPrintingController(chart);
```

2. Invoke on that instance the action you want to see performed. For example `print()`, `setupDialog()`, or `printPreview()` as shown below:

```
printController.printPreview(JOptionPane.getFrameForComponent(chart));
```

The following figure shows the result of invoking the `printPreview` method.



Printing framework API

The following classes are involved in the Resource Data chart printing framework:

- *IlvResourceDataChartPrintableDocument*
- *IlvResourceDataChartPrintingController*
- *IlvPrintableResourceDataChart*

IlvResourceDataChartPrintableDocument

The `IlvResourceDataChartPrintableDocument` class stores the document print structure and a set of parameters to customize printing, such as:

- The printed data window; that is, the part of the chart to be printed.
- How the chart fits on the page.

This class extends `IlvChartPrintableDocument` and thus supports the properties contained in that class. It is used to create and populate pages to be printed. It also supports a convenient way of defining the printed data window using a start date and a duration. The previous figure shows the customizable properties of this class.

The following table shows the customizable properties of an `IlvResourceDataChartPrintableDocument` instance.

Property	Description	Default
start	The start date of the first printed page. This sets the beginning of the printed data window.	The chart visible time
duration	The duration to the last printed page. This is used to set the end of the printed data window.	The chart visible duration.

Note: All these properties are accessible from the JViews Resource Data chart Print Setup dialog box. You can invoke this dialog box by calling the method `setupDialog` on an `IlvResourceDataChartPrintingController` instance.

IlvResourceDataChartPrintingController

The `IlvResourceDataChartPrintingController` class controls the printing process. It initiates the printer job, handles the Setup and Preview dialog boxes, and configures the document accordingly. This class extends `IlvChartPrintingController`. By default it provides the chart Setup window.

Document configuration can be done in the following ways:

- Automatically, by using the following constructor:

```
public IlvResourceDataChartPrintingController(IlvResourceDataChart chart)
```

When you call this function a single landscape oriented printable document page is created. The printed data window size is equal to the visible data ranges on the x-axis.

- Programatically, by creating a Resource Data chart printable document and setting the parameters as described in *IlvResourceDataChartPrintableDocument* section.

IlvPrintableResourceDataChart

The `IlvPrintableResourceDataChart` class represents a printable Resource Data chart object. It extends the `IlvPrintableChart` class. This class allows you to print the `IlvResourceDataChart` within the printable area of an `IlvPage`.

The way the Resource Data chart fills the region is determined by the document properties. You do not need to call `IlvPrintableResourceDataChart` directly to print using parameters provided by the Setup dialog box. This class is necessary only if you want to control your `IlvPrintableDocument` object by creating pages and adding `IlvPrintableObject` instances.

For information on `IlvPrintableDocument`, and `IlvPrintableObject`, see The generic printing framework in *Advanced Features of IBM® ILOG® JViews Framework*.

To see how `IlvPrintableResourceDataChart` is used, see the print implementation in the Resource Data chart demo. This demo can be found at:

<installdir>/jviews-gantt86/samples/printResourceData

The sample Synchronized Schedule and Load charts shows how to implement a printable document composed of a Gantt chart on top of a Resource Data chart. This sample can be found at:

<installdir>/jviews-gantt86/samples/resourceLoadChart

Critical path analysis

Describes the critical path analysis feature, that is, the process for finding the activities in the project schedule that, if delayed, will hold up completion of the entire project.

In this section

Critical path analysis overview
Explains the facilities provided by JViews Gantt to compute the critical path automatically or manually.

Example
Explains how to run the critical path example and describes the code necessary to analyze the critical path.

Handling errors
Describes the situations that make critical path analysis impossible and explains how to cope with errors.

© Copyright IBM Corp. 1987, 2009

Critical path analysis overview

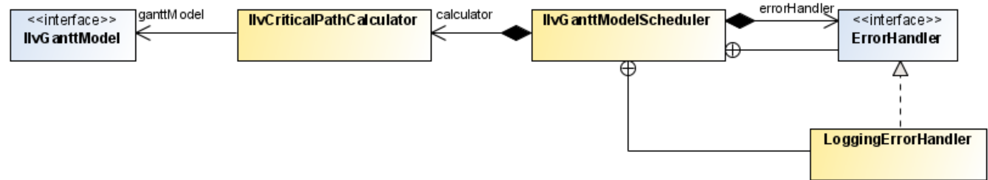
JViews Gantt provides basic facilities for automatically or manually computing the critical path for a Gantt data model. Critical path analysis is the process of finding those activities in the data model that, if delayed, will hinder completion of the entire project schedule.

The `ilog.views.gantt.model` package contains the following classes that perform critical path analysis:

- `IlvCriticalPathCalculator`: Performs manual critical path analysis of a Gantt data model on request.

- `IlvGanttModelScheduler`: Uses an instance of `IlvCriticalPathCalculator` to perform automatic critical path analysis of a Gantt data model each time the activities in the schedule change.

The following figure shows the relationship of these classes.



JViews Gantt performs critical path analysis by rescheduling all activities in the Gantt data model to occur as soon as possible after the project start time. This is done by modifying the activity start and end times. Preceding and succeeding constraints are taken into account in the analysis. Activities that are computed to have a total slack time that is less than a threshold are considered to be on the critical path. The default threshold value is zero.

Note: Activity reservations are not taken into account and resource leveling is not performed when critical path analysis is performed.

In addition to updating each activity's start and end times, critical path analysis sets several additional properties for each activity. If the default property names conflict with properties that are already being used in your Gantt data model, property naming methods are available for both `IlvCriticalPathCalculator` and `IlvGanttModelScheduler`. These methods are used so the class instances can be customized to set different property names for your activities.

The following table shows the activity properties set by critical path analysis.

Description	Property type	Default activity property	Methods to change property name
Earliest start time	Date	earliestStart	`String getEarliestStartProperty()`
			`void setEarliestStartProperty( String propertyName)`
Earliest finish time	Date	earliestFinish	`String getEarliestFinishProperty()`
			`void setEarliestFinishProperty( String propertyName)`
Latest start time	Date	latestStart	`String getLatestStartProperty()`
			`void setLatestStartProperty( String propertyName)`
Latest finish time	Date	latestFinish	`String getLatestFinishProperty()`
			`void setLatestFinishProperty( String propertyName)`
Total slack	IlvDuration	totalSlack	`String getTotalSlackProperty()`
			`void setTotalSlackProperty( String propertyName)`
Critical	Boolean	critical	`String getCriticalProperty()`
			`void setCriticalProperty( String propertyName)`

Example

A basic sample Java™ application, `CriticalPathExample.java`, illustrates the critical path capabilities of the IBM® ILOG® JViews Gantt. Open the sample index file found at:

<installdir>/jviews-gantt86/samples/criticalPath.

The instructions in `index.html` explain how to run the Critical Path example. The source code of this example can be found in:

<installdir>/jviews-gantt86/samples/criticalPath/src/criticalPath.

To analyze the critical path:

1. Create an `IlvGanttModelScheduler` that will automatically analyze the critical path of a Gantt data model as its activities change:

```
IlvGanttModel model = ...
IlvGanttModelScheduler scheduler = new IlvGanttModelScheduler(ganttModel)
;
scheduler.setAutoScheduling(true);
```

2. Use an `IlvGanttModelSchedule` to manually analyze the critical path of a Gantt data model in the following way:

```
IlvGanttModel model = ...
IlvGanttModelScheduler scheduler = new IlvGanttModelScheduler(ganttModel)
;
scheduler.schedule();
```

3. It is also possible to use an `IlvCriticalPathCalculator` instance to manually analyze the critical path:

```
IlvGanttModel model = ...
IlvCriticalPathCalculator calc = new IlvCriticalPathCalculator(ganttModel)
;
try {
  calc.computeSchedule();
} catch (IlvConstraintCycleException ex)
}
```

Handling errors

Critical path analysis cannot be performed if cyclical dependencies exist between the activities and constraints in your Gantt data model. The `IlvCriticalPathCalculator.computeSchedule()` method will throw an `IlvConstraintCycleException` if such a cyclical dependency is detected. By contrast, the `IlvGanttModelScheduler` class does not directly throw exceptions when a cyclical dependency is found. Instead, it notifies its currently registered error handler of the problem. This allows you to easily implement a custom error handler that can inform the user in a manner that is appropriate to your application's user interface. The default error handler simply logs the exception. The following code shows a custom error handler that displays detected cycles in a Swing `JLabel` object:

```
JLabel errorText = ...
IlvGanttModel model = ...
IlvGanttModelScheduler scheduler = new IlvGanttModelScheduler(ganttModel);
scheduler.setErrorHandler(new IlvGanttModelScheduler.ErrorHandler() {
  public void error(IlvConstraintCycleException ex,
                    IlvGanttModel ganttModel) {
    errorText.setText("Cycle at " + ex.getActivity().getName());
  }
});
```

Loading data on demand

Explains the mechanism used to load data into memory as it is needed.

In this section

Vertical load-on-demand
Explains the mechanism for loading data into memory as it is needed.

Horizontal load-on-demand
Explains how a resource-oriented Schedule chart can defer the loading of reservation data based upon the currently visible time interval.

© Copyright IBM Corp. 1987, 2009

Vertical load-on-demand

Explains the mechanism for loading data into memory as it is needed.

In this section

Overview
Explains the difference between horizontal and vertical load-on-demand.

Running the Database Gantt example
Describes how to run the sample.

Understanding the Database Gantt example
Explains the classes and methods used to visualize a read-only relational database containing scheduling data.

Overview

JViews Gantt provides a mechanism for loading data into memory as it is needed for display. This mechanism is called load-on-demand and is very valuable when visualizing large scheduling data sets. The JViews Gantt load-on-demand mechanism consists of two separate parts. The first is called "vertical load-on-demand" and can be used to defer loading of row data in both the Gantt and Schedule charts. The second is called "horizontal load-on-demand" and is available only in the resource-oriented Schedule chart. It allows you to defer loading of reservation data based upon the current visible time interval being displayed.

Vertical load-on-demand refers to the ability to defer loading of row-oriented information until it is needed for display. Row-oriented information is activity data for a Gantt chart and resource data for a Schedule chart. Vertical load-on-demand is facilitated by the design of JViews Gantt and also requires appropriate design of the `IlvGanttModel` Implementation. The default in-memory data model implementation, `IlvDefaultGanttModel`, does not support load-on-demand. The Database examples are provided to illustrate the basics of how to implement a load-on-demand data model.

The classes provided in these examples can be customized for your own use, or they can serve as a source of ideas for your own implementation:

- *Running the Database Gantt example* explains how to start the example.
- *Understanding the Database Gantt example* explains the design and purpose of the example.

Running the Database Gantt example

The source code file of the Database Gantt example application can be found in:

<installdir>/jviews-gantt86/samples/databaseGantt/src/database/DBGanttExample.java

To run the example:

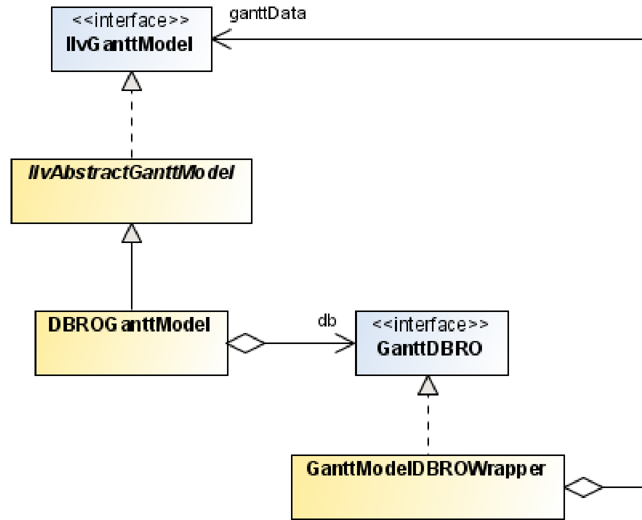
1. Ensure that the `Ant` utility is properly configured. If not, read Starting the samples for instructions on how to configure `Ant` for JViews Gantt.
2. Go to the directory where the sample is installed and type:

```
ant run
```

Understanding the Database Gantt example

The Database Gantt example application visualizes a read-only relational database containing scheduling data. The Gantt chart is bound to an instance of the `DBROGanttModel` class. This data model implementation is designed to query a relational database defined by the `GanttDBRO` interface. The `GanttModelDBROWrapper` class implements the `GanttDBRO` interface by simulating a database view of an existing Gantt data model.

The following figure shows the class relationships of the Database Gantt example:



The `DBROGanttModel` data model implementation is designed to load scheduling data on-demand from an underlying relational database defined by the `GanttDBRO` interface. The `GanttDBRO` interface defines four inner interfaces that define the record structure of the activity, resource, constraint, and reservation data.

The following table show the GanttDBRO inner interfaces.

Data model entity	Database record interface
Activities	`GanttDBRO.ActivityRecord`
Resources	`GanttDBRO.ResourceRecord`
Constraints	`GanttDBRO.ConstraintRecord`
Reservations	`GanttDBRO.ReservationRecord`

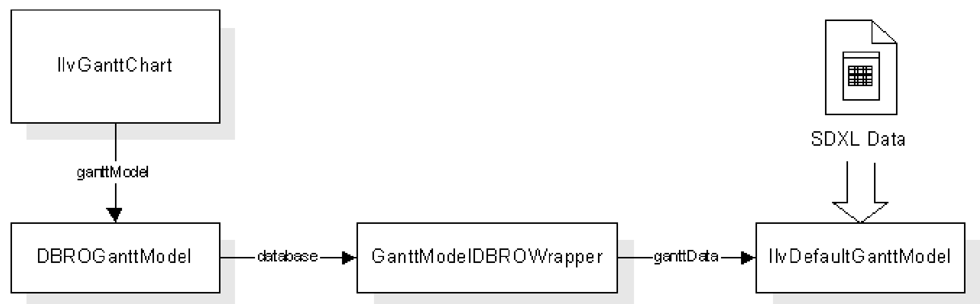
Each data model entity has a String lookup key that is unique among its instances. For activities and resources, this can be the same as the ID property, but this is not a requirement. The `GanttDBRO` interface defines the following database query methods:

The following table show the GanttDBRO database query methods.

Data model entity	GanttDBRO methods
Activities	`String queryRootActivityKey()`
	`ActivityRecord queryActivity(String key)`
	`String queryActivityParent(String key)`
	`String[] queryActivityChildren(String key)`
Resources	`String queryRootResourceKey()`
	`ActivityRecord queryResource(String key)`
	`String[] queryResourceChildren(String key)`
	`String queryResourceParent(String key)`
Constraints	`String[] queryConstraints()`
	`String[] queryConstraintsFromActivity(String activityKey)`
	`String[] queryConstraintsToActivity(String activityKey)`
	`ConstraintRecord queryConstraint(String key)`
Reservations	`String[] queryReservations()`
	`String[] queryReservationsForActivity(String activityKey)`
	`String[] queryReservationsForResource(String resourceKey)`
	`String[] queryReservationsForResource(String resourceKey, Date start, Date end)`
	`ReservationRecord queryReservation(String key)`

The `GanttModelDBROWrapper` class simulates a `GanttDBRO` database by creating in-memory tables and keys from an existing Gantt data model. When you open an XML schedule data file in the Database Gantt Example, a standard `IlvDefaultGanttModel` instance is created from the data. This data model is then wrappered by an instance of `GanttModelDBROWrapper` that is bound to a `DBROGanttModel` instance.

The following figure shows the object relationships of the Database Gantt example.



When an activity row is first displayed in the Gantt chart, the chart calls the method `queryActivityChildren` of the database. This is to determine whether the activity is a parent or a leaf row so that it can be rendered properly. The keys of the activity children are then cached in the `DBROGanttModel`. Then, when the activity row is expanded, the chart will call the `queryActivity` method of the database for each newly visible child row. This is to obtain the activity properties that are displayed. You can monitor this behavior by selecting Display Database Queries from the File menu. This will log all accesses from the `DBROGanttModel` to the `GanttModelDBROWrapper` database implementation onto the system console.

In order to achieve the same vertical load-on-demand capabilities for your application, you can create an implementation of the `GanttDBRO` interface that connects to the source of your scheduling data. Your scheduling data schema should be organized as four separate tables for activity, resource, constraint, and reservation records, and each table must have a primary key string that uniquely identifies each record. You can use the source code for the `GanttModelDBROWrapper` class to get ideas on how to design your implementation. Once you have created a `GanttDBRO` implementation, you can bind it to a `DBROGanttModel` instance and then to your chart, as follows:

```
IlvGanttModel dbModel = new DBROGanttModel(aGanttDBRO);
aChart.setGanttModel(dbModel);
```

Horizontal load-on-demand

Explains how a resource-oriented Schedule chart can defer the loading of reservation data based upon the currently visible time interval.

In this section

Overview
Describes the technology and features used in the example.

Running the Database Schedule example
Explains how to run the Database Schedule example.

Understanding the Database Schedule example
Describes the classes and code used in this example and explains how they relate to the GUI and user actions.

Overview

Horizontal load-on-demand is a capability of the resource-oriented Schedule chart to defer loading of reservation data based upon the currently visible time interval. As the chart is scrolled or zoomed horizontally to display different time intervals, the Schedule chart queries the data model for new reservations that need to be displayed.

The Database Schedule example application illustrates this feature:

- *Running the Database Gantt example* explains how to start the example.
- *Understanding the Database Gantt example* explains the design and purpose of the example

Running the Database Schedule example

The source code file of the Database Schedule example application is named `DBScheduleExample.java` and can be found in:

`<installdir>/jviews-gantt86/samples/databaseSchedule/src/database/DBScheduleExample.java`

To run the example,

1. Ensure that the Ant utility is properly configured.

   If not, see the Starting the samples for instructions on how to configure Ant for JViews Gantt.

2. To run the example as an application, go to the directory where the example is installed and type:

   ```
   ant runschedule
   ```

Understanding the Database Schedule example

The Database Schedule example application visualizes the same relational database of scheduling data as the *Understanding the Database Gantt example*. Therefore, the Schedule chart performs vertical load-on-demand of resource-oriented row information in the same manner that the Gantt chart defers loading activity-oriented row information. In addition, the Database Schedule example application illustrates the horizontal load-on-demand capability of the Schedule chart.

Here are the `IlvScheduleChart` methods that control this feature:

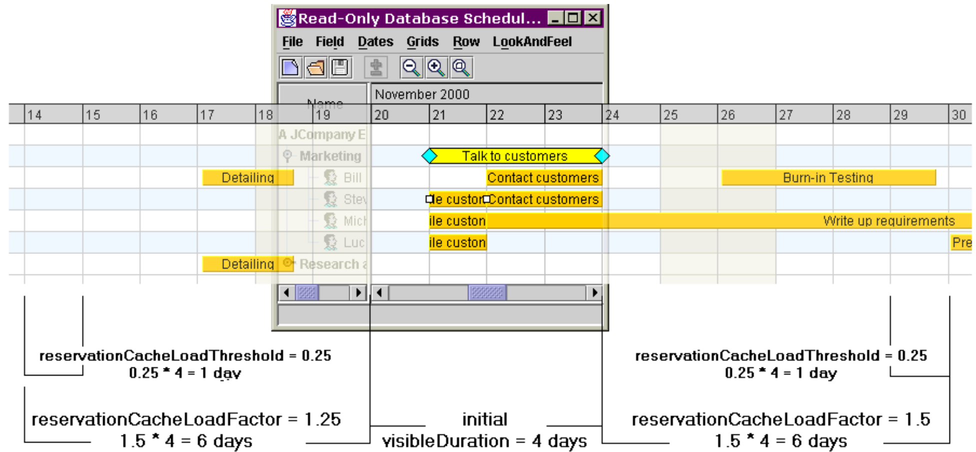
- `boolean isReservationCachingEnabled()`
- `void setReservationCachingEnabled(boolean enabled)`
- `void setReservationCachingEnabled(boolean enabled)`
- `void setReservationCachingEnabled(boolean enabled)`
- `void setReservationCachingEnabled(boolean enabled)`
- `void setReservationCachingEnabled(boolean enabled)`

By default, the reservation caching property of an `IlvScheduleChart` is disabled. The Database Schedule example explicitly enables reservation caching in its `createGanttModel` method:

```
schedule.setReservationCachingEnabled(true)
```

Once reservation caching is enabled, the Schedule chart uses the values of its `reservationCacheLoadThreshold` and `reservationCacheLoadFactor` properties to determine how often and by how much it should query the data model for reservations that need to be displayed as the chart is scrolled horizontally. Both values are floating point numbers that are multiplied by the current visible duration displayed by the chart. The load threshold indicates how far the chart must be scrolled before it queries the data model for fresh reservations to display. The load factor determines the time duration that the chart will use when it queries the data model for new reservations.

The following figure shows the relationship between these settings,



The Schedule chart is initially displayed with a visible duration of 4 days, from November 20 through November 23. The chart `reservationCacheLoadThreshold` is set to its default value of 0.25 and the `reservationCacheLoadFactor` is set to its default value of 1.5. When the chart is first displayed, it will query the data model for all reservations assigned to the visible resource rows and for which the reserved activity intersects the time interval of November 14 through November 29, a total of 16 days. The chart does this by invoking the method `IlvGanttModel.reservationIterator (IlvResource resource, IlvTimeInterval interval)`. The time interval for which the chart queries the data model is computed as:

from: `visibleStartTime` – (`visibleDuration` * `reservationCacheLoadFactor`)

to: `visibleEndTime` + (`visibleDuration` * `reservationCacheLoadFactor`)

If the `reservationCacheLoadFactor` is set to its minimum value of 0, the chart will only query the data model for the exact visible time interval. The larger the load factor, the larger the time span that the chart will query the data model each time. In the above example, the reservations named “Detailing”, “Burn-in Testing”, and “Write up requirements” have all been loaded into the chart’s internal cache, even though “Detailing” and “Burn-in Testing” are not currently visible. The reservation named “Prepare Demo”, to the far right side, is not currently cached by the chart because it is outside the computed time interval.

The Schedule chart `reservationCacheLoadThreshold` determines the trigger point at which the chart will query the data model for fresh reservations to display. In this example, the default value of 0.25 multiplied by a `visibleDuration` of 4 days gives a trigger threshold of 1 day. This means that when the chart is scrolled horizontally to within 1 day of the time span currently cached, the chart will query the data model for new reservations to display based upon the `reservationCacheLoadFactor` formula presented earlier.

Here is the sequence of events in more detail:

1. The chart is initially displayed with a visible time interval of November 20 through November 23, a total of 4 days. The chart caches reservations for the time interval of November 14 through November 29, a total of 16 days.

2. The chart is scrolled forward in time to the 4 day visible time interval of November 25 through November 28.

3. The chart is scrolled forward a bit more, so that the beginning of November 29 just becomes visible. This triggers the `reservationCacheLoadThreshold` and the chart queries the data model for the reservations based upon the `reservationCacheLoadFactor` and the currently visible time interval. This will be the 16 days centered around November 25 through November 29, and is computed to be the time interval of November 19 through December 4. However, the reservations for November 19 through November 29 are already cached by the chart. Therefore, the chart only queries the data model for the reservations in the time interval of November 30 through December 4.

The same logic is applied to load reservations from the data model when the Schedule chart is scrolled backwards in time, is zoomed, or the visible time interval is changed by invoking the appropriate APIs. You can monitor how the Database Schedule example application queries the data model by selecting Display Database Queries from the File menu. This will log all queries to the system console.

Document type definition for SDXL

The DTD used is the following:

```
<!--
    This is the DTD for IBM® ILOG JViews Schedule Data Exchange Language.

    Version 5.5, Dec 20, 2002
-->

<!-- ISO date format -->
<!ENTITY % Datetime "CDATA">

<!ENTITY % Text "CDATA">

<!-- Must be an activity ID in the document -->
<!ENTITY % ActivityID "CDATA">

<!-- Must be an resource ID in the document -->
<!ENTITY % ResourceID "CDATA">

<!ENTITY % ConstraintType "(Start-Start|Start-End|End-Start|End-End)">

<!ELEMENT activity (activity|property)*>
<!ATTLIST activity
    id    ID         #REQUIRED
    name  %Text;     #REQUIRED
    start %Datetime; #REQUIRED
    end   %Datetime; #REQUIRED
    info  %Text;     #IMPLIED >

<!ELEMENT activities (activity)+>
<!ATTLIST activities
    dateFormat %Text; #IMPLIED >

<!ELEMENT resource (resource|property)*>
<!ATTLIST resource
    id       ID     #REQUIRED
    name     %Text; #REQUIRED
    quantity %Text; #IMPLIED
    info     %Text; #IMPLIED >

<!ELEMENT resources (resource)+>
<!ATTLIST resources>

<!ELEMENT reservation (property)*>
<!ATTLIST reservation
    activity %ActivityID; #REQUIRED
    resource %ResourceID; #REQUIRED
    info     %Text;       #IMPLIED >

<!ELEMENT reservations (reservation)+>
<!ATTLIST reservations >
```

© Copyright IBM Corp. 1987, 2009

```
<!ELEMENT constraint (property)*>
<!ATTLIST constraint
    from %ActivityID;     #REQUIRED
    to   %ActivityID;     #REQUIRED
    type %ConstraintType; #REQUIRED
    info %Text;           #IMPLIED >

<!ELEMENT constraints (constraint)+>
<!ATTLIST constraints >

<!ELEMENT title (#PCDATA)>
<!ELEMENT desc (#PCDATA)>

<!ELEMENT schedule (title?, desc?, resources?, activities?,
                    constraints?, reservations?) >
<!ATTLIST schedule
    version %Text; #REQUIRED >
<!ELEMENT property (#PCDATA)>
<!ATTLIST property
     name       %Text;     #REQUIRED
     javaClass  %Text;     #IMPLIED >
```

Index

A

abstract interfaces **14**
`AbstractExample` class **296**
`AbstractGanttExample` class **182**
activities
  and reservations **34**
  changing start/end time **256, 258**
  creating using the mouse **260**
  CSS pseudoclasses **134**
  expanding/collapsing **190**
  ID selectors **131**
  in the Gantt sheet **196, 198**
  populating the data model **27**
  querying from JDBC **67**
  rendering **197**
  root, parent, leaf **27**
  rows **198**
  styling **125**
activity data
  load on demand **334**
  record structure **336**
`activity` element type **125**
activity factory **30, 260**
activity graphic
  popup menus **262**
activity graphics
  description **197**
  layout **205**
  moving **256**
  resizing **258**
  selecting **255**
  stacking order **205**
activity renderers **72**
  customizing **221**
  factory **223**
  target objects **128**
`activityProperty` function **146**
`addResource` method
  `IlvGanttModel` interface **28**
analysis
  critical path **326**
Apache Xerces parser implementation **39**
applet, creating **282, 296**
architecture
  of a data model **19**
  of Gantt chart and Schedule chart beans **173**
  of Resource Data chart **268**
  of the Gantt sheet **196**
attribute matching **87**
  CSS2 syntax **98**
  dynamic behavior **87**
AWT
  packages, importing **182**

B

background color property **176**
Bean properties
  for `IlvBasicActivityBar` **125**
  for `IlvChart.Area` **151**
  for `IlvConstraintGraphic` **141**
  for `IlvGrid` **157**
  for `IlvResourceDataChart` **148**
  for `IlvScale` **155**
  for `IlvTimeScale` **155**
  for the `point` model object type **162**
  for the `series` model object type **162**
  in the CSS engine **86**
  property editor`setAsText` method **86**
Beans
  creating, CSS recursion **92**
  Gantt chart and Schedule chart **173**
    using, basic steps **177**
  Resource Data chart **267**
  using in applets **296**
bounded, scroll bar operation mode **192**

© Copyright IBM Corp. 1987, 2009

C

Cascade layout, in Schedule chart **206**
cascading
  CSS2 syntax **98**
  definition **82**
cell editor **244**
cell renderer **243**
`cellUpdated` method
  `IlvAbstractJTableColumn` class **244**
changing time indicators **203**
changing visibility
  of time indicators **203**
chart area component
  styling, Bean properties **151**
`chart` CSS element type **148**
chart grid, styling **157**
chart legend, styling **152**
chart renderer, styling **154**
chart scales, styling **155**
`chartArea` CSS element type **151**
`chartGrid` CSS element type **157**
`chartLegend` CSS element type **152**
`chartRenderer` CSS element type **154**
`chartScale` CSS element type **155**
class loader **299**
`class` property name **88**
class relationships
  in Database Gantt example **336**
classes
  critical path **326**
collapsing activities and resources **190**
color, Gantt Bean property **176**
combinator, in selector **80**
comparing attribute patterns **98**
connecting to data
  custom data models **72**
  Swing TableModel instances **49**
    complex mappings **62**
    dynamic behavior **60**
    read-write support **59**
    reading data from CSV files **61**
  via JDBC **65**
  XML data **37**
constraint
  popup menus **262**
`constraint` CSS element type **137**
constraint data
  record structure **336**
constraint factory **33, 261**
constraint graphics
  not movable **256**
  selecting **255**
  target objects, Bean properties **141**
constraints
  automatic removal **32**
  creating **32**
    using the mouse **261**
  CSS pseudoclasses **145**
  in the activity Gantt sheet **198**
  in the Gantt sheet **196**
  querying from JDBC **67**
  representation **198**
  styling **137**
`createActivity` method
  `IlvActivityFactory` interface **30**
`createConstraint` method
  `IlvConstraintFactory` interface **33**
`createGanttModel` method
  `IlvScheduleChart` class **342**
`createPages` method
  `IlvGanttPrintableDocument` class **317**
`createReservation` method
  `IlvReservationFactory` interface **35**
`createResource` method
  `IlvResourceFactory` interface **30**
creating
  time indicator **200**
creating current time indicators **202**
critical path
  analysis **326**
  classes **326**
  errors **329**
  example **328**
  rescheduling **326**
  threshold **326**
CSS
  `!important` token **82**
  and CSS2 **98**
  applying to Java objects **83**
  `class` property name **88**
  customizing charts with **75**
  data model **87**
    user-defined types **87**
  origins **79**
  property **108**
  recursion **91**
  renderer target objects **128**
  styling constraints **137**
  tag **87**
  transition symbols **72**
  user-defined properties **107**
  W3C **79**
CSS classes **87**
  and object IDs **88**
  for `IlvGeneralActivity` **133**
  for `IlvGeneralConstraint` **144**
  for Resource Data chart data **164**
CSS element types
  `activity` **125**
  `chart` **148**

chartArea **151**
chartGrid **157**
chartLegend **152**
chartScale **155**
constraint **137**
for Gantt and Schedule chart components **118**
for Resource Data chart components **148**
number, for rule priority **81**
point **161**
syntax, use of quotes **98**
CSS engine **86**
CSS examples **17**
Gantt chart and Schedule chart **105**
Resource Data chart **111**
CSS model object types
series **161**
CSS pseudoclasses
for activities **134**
for constraints **145**
CSS syntax
cascading **82**
classes and object IDs **88**
declarations **81**
inheritance **82**
presentation **80**
priority **81**
selectors **80**
minimal building blocks **72**
using in style sheets **77**
CSS2 (Cascading Style Sheet level 2) **78**
CSS2 syntax **98**
attribute matching **98**
cascading **98**
empty string **99**
enhancement **98**
null value **99**
pseudoclasses **89, 98**
pseudoelements **89, 98**
CSV (Comma-Separated Values) files, reading Gantt data from **61**
custom data models, implementing **72**
custom functions
in expressions **95**
registering **97**
custom icon **218**
CustomActivity class
instantiating **220**
CustomActivityCompositeGraphicRendererFactory example class for composite graphics **236**
CustomActivityRenderer.java file **224**
CustomGanttExample.java sample file **216**
customizable properties
time indicator **202**
customizing
activity rendering **221**
charts
applying a CSS **108**
Gantt charts **183, 215**
Gantt data model **219**
SDXL readers and writers **42**
table columns **239**
time indicator **202**
customizing default renderer
time indicator **200**
customizing time indicator
graphic properties **202**
CustomLeafActivityCompositeGraphic example class for composite graphics **232**
CustomLeafActivityCompositeGraphicRenderer example class for composite graphics **234**
CustomParentActivityCompositeGraphic example class for composite graphics **232**
CustomParentActivityCompositeGraphicRenderer example class for composite graphics **234**

D

data
reading
from CSV files **61**
from Swing TableModel instances **49**
from XML files **37**
via JDBC **65**
rendering in Gantt sheet **198**
styling
Gantt chart and Schedule chart **123**
Resource Data charts **159**
data model
architecture **19**
binding Gantt chart components **19**
binding the Beans **183**
creating **182**
custom, implementing **72**
customizing **219**
definition **14**
for CSS **87**
implementations
abstract **16**
concrete **17**
connection to a JDBC database **18**
connection to Swing TableModel instances **18**
default **17**
simple **17**
indirection **89**
lookup keys for entities **336**
populating **26**
from Swing TableModel instances **49**
from XML files **37**
using JDBC queries **67**
data nodes **242**

visibility states **190**
data series (Resource Data charts)
properties **168**
rendering **85**
database examples **16**
Gantt chart
class relationships **336**
object relationships **336**
running **335**
understanding **336**
Schedule chart
running **340**
understanding **342**
`Date` Java class **20**
declarations
CSS syntax **81**
inheritance **82**
style rule **78, 80**
default graphic
time indicator **200**
default readers and writers **21**
default renderer
time indicator **200**
`DefaultTableModel` Swing class **59**
`DefaultTreeCellRenderer` Swing API **241, 243**
defining a renderer
time indicator **200**
defining new renderer
time indicator **200**
deploying
Gantt application **186**
depopulating
in-memory data model **26**
divider position
printable document **312**
`draw` method
`IlvActivityRenderer` interface **197**
Duration
printable document **322**
duration
time scale **213**
duration of an activity
API **20**
dynamic behavior
attribute matching **87**

E

Eclipse Rich Client Platform **297**
class loader **299**
runtime plugin **297**
element types
See CSS element types
emphasis
time scale **213**
empty string
CSS2 syntax **99**
`enableAWTThreadRedirect` method
`IlvEventThreadUtil` class **301**
end date, printable document **312**
end time
changing **256, 258**
creating a constraint **261**
drawing a constraint **261**
end-to-end, constraint type **32**
end-to-start, constraint type **32**
errors
critical path **329**
errors in CSS **94**
`EventQueue` class
`isDispatchThread` method **301**
example
critical path **328**
examples
CSS **17**
Custom Gantt chart **216**
database **16, 72**
Database Gantt **335**
Database Schedule **340**
Filter **72**
Gantt chart and Schedule chart **179**
Gantt CSS and Schedule CSS **105**
Gantt printing **310**
harbor.mdb **69**
Resource Data CSS **111**
expanding activities and resources **190**
expressions **93**
CSS **94**

F

factories
activities **30**
constraints **33**
custom activity renderer **223**
reservations **35**
resources **30**
font
property
for Gantt Beans **176**
foreground color
property
for Gantt Beans **176, 188**
`formatDate` function **135**
`formatDuration` function **135**
From activity **261**
functions
`activityProperty` **146**
custom **95**
`formatDate` and `formatDuration` **135**
standard **94**

G

Gantt application, deploying **186**

Gantt chart
activity Gantt sheet **198**
adding to the user interface **184**
and styling **85**
and vertical load-on-demand **333**
binding to the data model **19**
class overview **10**
creating constraints **261**
CSS example **105**
customizing **183, 215**
with style sheets **118**
example **180**
interacting with **253**
moving activity graphics **256**
popup menus **262**
scrolling **192**
table columns **242**
using **187**
Gantt chart bean
binding to the data model **183**
creating **183**
example **180**
using **177**
visual properties **188**
Gantt Data Model
connecting to business data **272**
instantiating **272**
Gantt data model **85**
binding instances to a chart **272**
binding to a chart **272**
customizing **219**
JDBC read-only mode **70**
JDBC read-write mode **70**
passing JDBC data to **70**
Resource Data chart **268**
Gantt sheet **195**
adding time indicator **203**
architecture **196**
interacting with, using the mouse **259**
rendering data **198**
scrolling **192**
`ganttRowIterator` method
`IlvGanttSheet` class **196**
`GenericEventListener` interface
`inform` method **244**
introduction **244**
`getActivityFactory` method
`IlvHierarchyChart` class **30**
`getActivityRenderer` method
`IlvActivityGraphic` class **197**
`getBounds` method
`IlvActivityRenderer` interface **224**
`getColumnClass` method
`TableModel` class **54**
`getColumnName` method
`TableModel` class **54**
`getConstraint` method
`IlvConstraintGraphic` class **198**
`getDividerPosition` method
`IlvGanttPrintableDocument` class **312**
`getEnd` method
`IlvGanttPrintableDocument` class **312**
`getGanttModel` method
`IlvHierarchyChart` class **19**
`getGanttRowCount` method
`IlvGanttSheet` class **196**
`getGanttSheet` method
`IlvHierarchyChart` class **176**
`getID` method
`IlvSDMModel` interface **89**
`getMax/getMinVisibleTime` methods
`IlvHierarchyChart` class **192**
`getName` method
`IlvDataSet` interface **162**
`getPagesPerBand` method
`IlvGanttPrintableDocument` class **312**
`getRepeatTable` method
`IlvGanttPrintableDocument` class **312**
`getReservation` method
`IlvReservationGraphic` class **198**
`getReservationCacheLoadFactor` method
`IlvScheduleChart` class **342**
`getReservationCacheLoadThreshold` method
`IlvScheduleChart` class **342**
`getRootActivity` method
`IlvGanttModel` interface **29**
`getRootResource` method
`IlvGanttModel` interface **29**
`getSelectedGraphics` method
`IlvHierarchyChart` class **255**
`getStart` method
`IlvGanttPrintableDocument` class **312**
`getTable` method
`IlvHierarchyChart` class **176, 247**
`getTableColumnCount` method
`IlvGanttPrintableDocument` class **312**
getting current time indicator **203**
getting specific time indicator **203**
getting time indicators **203**
`getValue` method
`IlvJTableColumn` interface **243**
`getVisibleGanttRowAt` method
`IlvGanttSheet` class **196**
`getVisibleGanttRowCount` method
`IlvGanttSheet` class **196**

H

hiding the table view **176**
hierarchical structure

expanding/collapsing **190**
of activities **27**
of resources **27**
horizontal load-on-demand
description **340**
horizontal scrolling **192**

I

icon
custom **218**
tree column **241**
ID selectors
#timeScale **155**
#xGrid **157**
#yGrid **157**
for activities **131**
yScale **155**
identifiers of objects
and CSS classes **88**
IlvAbstractGanttModel class **26**
IlvAbstractJTableColumn class **243**
cellUpdated method **244**
IlvAbstractUserDefinedProperty class **249, 250**
IlvActivity interface **10, 14, 27, 51, 128, 260**
IlvActivityCompositeGraphicRenderer class **222, 234**
IlvActivityCompositeRenderer class **92, 128, 222, 223, 224**
IlvActivityFactory interface **30**
createActivity method **30**
IlvActivityGraphic class
description **197**
getActivityRenderer method **197**
setActivityRenderer method **197**
IlvActivityGraphicMoveInteractor class **256**
IlvActivityGraphicRenderer class **128, 222, 234**
IlvActivityLayout interface **205**
IlvActivityRenderer interface **87, 88, 124, 128**
description **197**
draw method **197**
getBounds method **224**
isRedrawNeeded method **224**
IlvActivityRendererFactory interface **88, 125, 128**
IlvActivitySummary class **223**
IlvActivityUserDefinedProperty class **250**
IlvAxis class **271**
IlvBasicActivityBar class **92, 125, 134**
IlvBasicActivityLabel class **128**
IlvBasicActivitySymbol class **128**
IlvBasicTimeScaleVisibilityPolicy class **211**
IlvChart class **268**
IlvChart instance **271**
IlvChartRenderer class **168**
IlvChartRenderer implementation **278**
IlvConfigurableTableColumn class **249, 251**
IlvConstraint interface **10, 14, 32, 198, 261**
IlvConstraintFactory interface
createConstraint method **33**
description **33**
IlvConstraintGraphic class **87, 88, 124, 137, 141**
description **198**
getConstraint method **198**
IlvConstraintGraphicFactory interface **88, 137, 141**
IlvConstraintType class **32, 54**
IlvConvert class **250**
IlvCriticalPathCalculator class **326**
IlvCSSFunction class **95**
IlvCurrentTimeIndicator class **201**
IlvDataAnnotation class **166**
IlvDataSet interface **268**
getName method **162**
IlvDataSource interface **271**
IlvDataValue class **277**
IlvDayView class **10, 19**
IlvDefaultGanttModel class **334**
IlvDefaultTableCellRenderer class **251**
IlvDuration class **20, 94**
IlvEventThreadUtil class
enableAWTThreadRedirect method **301**
setAWTThreadRedirect method **301**
IlvFilterGanttModel class **72**
IlvFixedTimeIndicator class **201**
IlvFormattedNumberProperty class **224**
IlvGanttChart class **102, 174, 177, 185, 190, 248**
printing **308**
using in applets **296**
IlvGanttConfiguration class **175**
IlvGanttDocumentReader class **47**
readGanttModel method **47**
IlvGanttDocumentWriter class
writeGanttModel method **44**
IlvGanttGridRenderer interface **271, 279**
IlvGanttModel implementation **270, 272**
IlvGanttModel interface **28, 183, 268, 270, 282**
addResource method **28**
definition **14**
design for load-on-demand **334**
getRootActivity method **29**
getRootResource method **29**
instantiating **177**
moveActivity method **29**
moveResource method **29**
removeActivity method **29**
removeResource method **29**
setRootActivity method **29**
setRootResource method **29**
IlvGanttPrintableDocument class

createPages method **317**
description **312**
getDividerPosition method **312**
getEnd method **312**
getPagesPerBand method **312**
getRepeatTable method **312**
getStart method **312**
getTableColumnCount method **312**
setDividerPosition method **312**
setEnd method **312**
setPagesPerBand method **312**
setRepeatTable method **312**
setStart method **312**
setTableColumnCount method **312**
IlvGanttPrintingController class **310**
description **315**
IlvGanttReaderException class **48**
IlvGanttRow class **196**
IlvGanttSelectInteractor class **256**
IlvGanttSheet
controlling time indicators **203**
methods controlling time indicators **203**
IlvGanttSheet class **175, 203**
description **196**
ganttRowIterator method **196**
getGanttRowCount method **196**
getVisibleGanttRowAt method **196**
getVisibleGanttRowCount method **196**
IlvGanttStreamWriter class **45**
IlvGanttTimeScale class **208, 279**
IlvGeneralActivity class **88, 132, 248, 250**
CSS classes **133**
description **107, 113, 219**
properties **132**
pseudoclasses **134**
IlvGeneralActivity.Factory class **30**
IlvGeneralConstraint class **88, 107, 113, 219**
CSS classes **144**
properties **143**
IlvGeneralConstraint.Factory class **33**
IlvGeneralPath class **200, 202**
IlvGeneralReservation class **107, 113, 219**
IlvGeneralReservation.Factory class **35**
IlvGeneralResource class **107, 113, 162, 164, 219, 248, 250**
IlvGeneralResource.Factory class **30**
IlvGraphic
new renderer for time indicator **200**
IlvGraphic class **88, 128, 200, 201, 222**
IlvGrid class **271**
IlvHierarchyChart class **30, 59, 97, 102, 104**
getActivityFactory method **30**
getGanttModel method **19**
getGanttSheet method **176**
getMax/getMinVisibleTime methods **192**
getSelectedGraphics method **255**
getTable method **176, 247**
setActivityFactory method **30**
setDisplayingConstraints method **198**
setGanttModel method **19, 176**
setMax/setMinVisibleTime methods **192**
static constants **193**
IlvHierarchyNode interface **27, 251**
IlvJDBCGanttModel class **14, 51, 66**
IlvJTable class **175, 251**
IlvJTableColumn interface **251**
getValue method **243**
implementation **242**
isCellEditable method **243**
setValue method **243**
IlvLegend class **152**
IlvLine class **200**
IlvLinearTimeConverter interface **213**
IlvMakeActivityInteractor class **260**
IlvMakeConstraintInteractor class **261**
IlvManagerView class
pushInteractor method **255, 260**
IlvMonthView class **10, 19**
IlvPrintableGanttSheet class **315**
IlvPrintableResourceDataChart class **320**
description **323**
IlvPrintableTimeScale class **316**
IlvPrintingController class
print method **317**
printPreview method **317**
setupDialog method **317**
IlvReservation interface **10, 14, 34, 198, 260**
IlvReservationDataPolicy interface **277**
IlvReservationFactory interface
createReservation method **35**
description **35**
IlvReservationGraphic class
description **198**
getReservation method **198**
IlvReservationLoadData **277**
IlvResource interface **10, 14, 27**
IlvResourceDataChart class **102, 268, 272, 279**
printing **320**
using in applets **296**
IlvResourceDataChartPrintableDocument class **320**
description **322**
IlvResourceDataChartPrintingController class **320**
description **322**
IlvResourceDataSet class **87, 268, 277**
IlvResourceFactory interface **30**
IlvResourceUserDefinedProperty class **250**

IlvRowSetTableModel class **51**
IlvScale class **155**
IlvScheduleChart class **30, 97, 102, 177, 185, 190, 248, 342**
  createGanttModel method **342**
  getReservationCacheLoadFactor method **342**
  getReservationCacheLoadThreshold method **342**
  isReservationCachingEnabled method **342**
  printing **308, 309**
  setReservationCacheLoadFactor method **342**
  setReservationCacheLoadThreshold method **342**
  setReservationCachingEnabled method **342**
  using in applets **296**
IlvScheduleDataChart class **102, 268**
  setStyleSheets method **104**
IlvSDMModel interface
  getID method **89**
IlvSingleChartRenderer class
  isFilled method **165**
IlvStairChartRenderer class **168, 278**
IlvStringColumn class **251**
IlvStringProperty interface **249, 250, 251**
IlvStylable interface **102**
IlvSwingControl class **297**
IlvSwingUtil class
  isDispatchThread method **301**
IlvTableActivity class **14, 52**
IlvTableConstraint class **14, 52**
IlvTableGanttModel class **14, 51**
IlvTableModelEvent class **60**
IlvTableReservation class **14, 53**
IlvTableResource class **14, 52**
IlvTextFieldTableEditor class **251**
IlvTimeConverter interface **213**
IlvTimeIndicator class **200, 201**
IlvTimeInterval class **20, 21**
IlvTimeScale class **155, 175, 208, 279**
IlvTimeScale.html class **279**
IlvTimeScaleRow class **208**
IlvTimeScaleVisibilityPolicy interface **210**
IlvTimeScrollable interface **279**
IlvTimeWidthVisibilityPredicate class **211**
IlvTreeColumn class **241**
IlvUserPropertyHolder interface **107, 113, 124, 132, 143, 162, 164, 250**
IlvVisibilityPredicate interface **211**
IlvVisibleTimeScaleRows class **211**
IlvWeekendGrid **271, 279**
IlvWeekTimeScaleRow class **208**
implementations
  general
    referencing user-defined properties **107**
implementations of data model
  abstract **16**
  concrete **17**
  connection to a JDBC database **18**
  connection to Swing TableModel instances **18**
  default **17**
  simple **17**
import statement **82**
in-memory data model
  depopulating **26**
inform method
  GenericEventListener interface **244**
inherit token **82**
inheritance
  CSS syntax **82**
  of declarations **82**
input source
  creating **46**
  parsing **46**
interactors
  creating a constraint **261**
  creating an activity or reservation **30, 260**
  moving a graphic **256**
  selecting a graphic **255**
isCellEditable method
  IlvJTableColumn interface **243**
isDispatchThread method
  EventQueue class **301**
  IlvSwingUtil class **301**
isEventDispatchThread method
  SwingUtilities class **301**
isRedrawNeeded method
  IlvActivityRenderer interface **224**
isReservationCachingEnabled method
  IlvScheduleChart class **342**

J

JApplet Swing API **182**
JApplet Swing class **296**
Java objects
  applying CSS to **83**
  attribute matching **98**
JAXP **39**
JDBC
  retrieving data via **65**
JLabel Swing API **242**
JTable Swing API **176**
JTextField Swing API **242**
JTree Swing API **241**
JViews Charts
  compared to Resource Data chart **271**

L

leaf activity **27**
  rendering **218, 223**
`leaf` activity pseudoclass **134**
leaf resource **27**
levels
  for SDXL readers and writers **39**
literal
  CSS declaration **86**
load factor **342**
load threshold **342**
load-on-demand
  database query methods **336**
  design **334**
  horizontal **340**
  introduction **334**
  vertical **333**

M

`main` method **182**
memory
  popup menus **262**
`milestone` activity pseudoclass **134**
minimal building blocks of a selectors **72**
model indirection **89**
model property name **89**
model-view separation **72**
move interactor **256**
`moveActivity` method
  `IlvGanttModel` interface **29**
`moveResource` method
  `IlvGanttModel` interface **29**
moving graphics **256**
multiple selection **255**

N

nonlinear
  time scale **213**
null value
  CSS2 syntax **99**
number of columns
  printable document **312**

O

object identifiers **88**
object relationships
  database Gantt example **336**
output stream
  creating **44**
  writing a document to **45**

P

pages per band
  printable document **312**
parent activity **27**
  rendering **223**
`parent` activity pseudoclass **134**
parent resource **27**
parsing an input source **46**
`point` CSS element type **161, 162**
popup Menus **262**
popup menus
  activity graphic **262**
  constraint **262**
  Gantt chart **262**
  memory **262**
  register **262**
Pretty layout, in Schedule chart **205**
`print` method
  `IlvPrintingController` class **317**
printable document
  description **312**
  divider position **312**
  Duration **322**
  end date **312**
  number of columns **312**
  pages per band **312**
  repeat table **312**
  Start date **322**
  start date **312**
printing **308**
  examples
    resource data chart **321**
  Gantt example **310**
  printing framework **309**
printing controller
  configuration **315**
  description **315**
printing framework
  Gantt **309**
  resource data chart **320**
printPreview **321**
`printPreview` method
  `IlvPrintingController` class **317**
priority
  CSS syntax **81**
`priority` property
  user-defined **219**
`PriorityColumn` class **242, 247**
properties
  CSS **108**
  customizing time indicator graphic **202**
  of `IlvGeneralActivity` **132**
  `tags` **88**
  user-defined for activities **107**
  user-defined in CSS **107**
pseudoclasses **98**
  CSS2 syntax **89**
  divergences from CSS2 **98**
  for activities **134**
  for constraints **145**
pseudoelements

CSS2 syntax **89**
divergences from CSS2 **98**
`pushInteractor` method
`IlvManagerView` class **255, 260**

Q

queries to a JDBC database **67**

R

RCP **297**
read-only database connection
Gantt data model **70**
read-write database connection
Gantt data model **70**
readers, SDXL **39**
customizing **42**
default **21**
levels **39**
`readGanttModel` method
`IlvGanttDocumentReader` class **47**
record structure **336**
recursion
CSS **91**
refjavagantt
IlvSimpleCompositeChartRenderer **278**
register
popup menus **262**
registering custom functions **97**
`removeActivity` method
`IlvGanttModel` interface **29**
`removeResource` method
`IlvGanttModel` interface **29**
renderers
to style activities **72**
rendering data
Gantt sheet **198**
repeat table
printable document **312**
representing current time
time indicator **201**
representing specific time
time indicator **201**
rescheduling
critical path **326**
reservation data
load on demand **340**
record structure **336**
reservation factory **35, 260**
reservation graphics
description **198**
duplicating **257**
layout **205**
moving **256**
resizing **258**
selecting **255**
reservations
creating **34**
creating using the mouse **260**
in the Gantt sheet **196**
in the resource Gantt sheet **198**
querying from JDBC **68**
resolving URLs **89**
Resource Data
computing **272**
displaying **272**
resource data
computing **277**
displaying **278**
load on demand **334**
record structure **336**
Resource Data chart
Bean **267**
class overview **10**
compared to IBM® ILOG JViews Charts **271**
CSS element types **148**
CSS examples **111**
data
selector patterns **161**
styling **159**
styling **85, 147**
synchronizing **273**
x-axis **279**
x-grid **279**
Resource Data chart bean
using **267**
using, basic steps details **270**
Resource Data charts
architecture **268**
definition **267**
resource display modes **275**
resource factory **30**
resources
as rows in the Gantt sheet **198**
displaying **275**
expanding/collapsing **190**
Gantt sheet **198**
populating the data model **27**
querying from JDBC **67**
root, parent, leaf **27**
selecting for display **275**
root activity/resource **27**
rows
changing height **184**
data, load on demand **334**
description **196**
layout of reservation graphics **205**
of a time scale
creating **208**
visibility **211**
visibility **190**
rules

See style rules
runtime plugin **297**

S

Scalable Vector Graphics (SVG)
supported/unsupported CSS properties **301**
Schedule chart
and horizontal load-on-demand **340**
and styling **85**
class overview **10**
creating activities and reservations **260**
CSS example **105**
customizing with style sheets **118**
example **185**
Gantt sheet **198**
moving reservation graphics **256**
scrolling **192**
table columns **242**
using **187**
Schedule chart bean
example **185**
using **185**
basic steps **177**
visual properties **188**
schedule data
serializing **43**
scheduling data
for Gantt and Schedule chart CSS examples **107**
for the Resource Data chart CSS example **113**
scrolling in the Gantt sheet
horizontally **192**
vertically **193**
SDXL
creating
a document **43**
a stream writer **44**
an `IlvGanttDocumentWriter` **44**
an input source **46**
an output stream **44**
design criteria **39**
overview **39**
parsing an input source **46**
readers and writers
API f **39**
customizing **42**
default **21**
reading
a Gantt data model **47**
from a file **43, 46**
scenarios **39**
writing
a document to an output stream **45**
a Gantt data model to a document **44**
`selected` activity pseudoclass **134**
selection interactor
creating a move interactor **256**
description **255**
installing **255**
selectors
combinator **80**
CSS **80**
element patterns
Resource Data chart data **161**
minimal building blocks **72**
style rule **78, 80**
transitions **87**
serializing schedule data **43**
`series` CSS element type **162**
`series` CSS model object type **161**
`setActivityFactory` method
`IlvHierarchyChart` class **30**
`setActivityRenderer` method
`IlvActivityGraphic` class **197**
`setAsText` method
property editor of Bean property **86**
`setAWTThreadRedirect` method
`IlvEventThreadUtil` class **301**
`setDisplayingConstraints` method
`IlvHierarchyChart` class **198**
`setDividerPosition` method
`IlvGanttPrintableDocument` class **312**
`setEnd` method
`IlvGanttPrintableDocument` class **312**
`setGanttModel` method
`IlvHierarchyChart` class **19, 176**
`setMax/setMinVisibleTime` methods
`IlvHierarchyChart` class **192**
`setPagesPerBand` method
`IlvGanttPrintableDocument` class **312**
`setRepeatTable` method
`IlvGanttPrintableDocument` class **312**
`setReservationCacheLoadFactor` method
`IlvScheduleChart` class **342**
`setReservationCacheLoadThreshold` method
`IlvScheduleChart` class **342**
`setReservationCachingEnabled` method
`IlvScheduleChart` class **342**
`setRootActivity` method
`IlvGanttModel` interface **29**
`setRootResource` method
`IlvGanttModel` interface **29**
`setStart` method
`IlvGanttPrintableDocument` class **312**
`setStyleSheets` method
`IlvScheduleDataChart` class **104**
`setTableColumnCount` method
`IlvGanttPrintableDocument` class **312**
`setupDialog` method

IlvPrintingController class **317**
setValue method
IlvJTableColumn interface **243**
sharing an empty string **99**
showing the table view **176**
Simple layout, in Schedule chart **205**
specificity of style rules **81**
standard functions
in expressions **94**
Standard Widget Toolkit **297**
Start date
printable document **322**
start date
printable document **312**
start time
changing **256, 258**
creating a constraint **261**
drawing a constraint **261**
start-to-end, constraint type **32**
start-to-start, constraint type **32**
stream writer, creating **44**
style rules
declaration **78, 80**
example **78**
general template **78**
selector **78, 80**
specificity and priority **81**
syntax **80**
two kinds **117**
style sheets
and style rules **80**
applying and disabling styles **101**
example of a simple one **108**
for Gantt and Schedule chart components **118**
for Resource Data chart components **148**
specifying rendering attributes
of activities and constraints **124**
of Resource Data chart data series **160**
syntax **78, 80**
styles
applying **102**
disabling **104**
styling
activities **125**
ID selectors **131**
renderer target objects **128**
chart area component **151**
chart grid **157**
chart legend **152**
chart renderer **154**
chart scales **155**
constraints **137**
graphic target objects **141**
examples **75**
Gantt and Schedule chart components **118**
Gantt chart and Schedule chart data **123**
Resource Data chart
components **147**
data **159**
Swing API
DefaultTreeCellRenderer **241, 243**
IlvJTable **176**
importing packages **182**
JApplet **182, 296**
JLabel **242**
JTextField **242**
JTree **241**
TableColumn **242**
TreeCellRenderer interface **241**
Swing classes
DefaultTableModel **59, 60**
TableModel **51**
SwingUtilities class
isEventDispatchThread method **301**
syntax enhancement
CSS2 syntax **98**

T

table columns
customizing **239**
TableColumn Swing API **242**
TableModel Swing class **51**
getColumnClass method **54**
getColumnName method **54**
tag
CSS data model **87**
tags property
for identifying CSS classes **88**
of resources **164**
threshold
critical path **326**
Tile layout, in Schedule chart **206**
time indicators
adding to Gantt sheet **203**
changing **203**
changing their visibility **203**
creating **200**
creating current **202**
customizable properties **202**
customizing **202**
customizing default renderer **200**
default graphic **200**
default renderer **200**
defining a renderer **200**
defining new renderer **200**
getting all time indicators **203**
getting current time indicator **203**
getting specific **203**
IlvGraphic as new renderer **200**

representing current time **201**
representing specific time **201**
time intervals
changing **192**
computing **342**
definition **21**
time scale
changing rows **208**
create **213**
duration **213**
emphasis **213**
nonlinear **213**
row visibility **211**
scrolling **192**
using **207**
time, API **20**
To activity **261**
tokens
cascading priority **82**
transitions
selector **87**
symbols **72**
tree column icon **241**
`TreeCellRenderer` Swing API **241**

U

unbounded, scroll bar operation mode **192**
URL
resolving **89**
user-defined
`priority` property **219**
user-defined properties
and general data-model implementation **107**
data model classes supporting **39**
user-defined type
CSS data model **87**

V

vertical load-on-demand
description **333**
design **334**
vertical scrolling **193**
`VERTICAL_SCROLLBAR_XXX` static constants **193**
visibility
of activity/resource rows (data nodes) **190**
of time scale rows **210**

W

W3C
CSS **79**
`writeGanttModel` method
`IlvGanttDocumentWriter` class **44**
writers, SDXL **39**
customizing **42**
default **21**
levels **39**

X

`Xdefault`
`X Window System` **79**
Xerces parser implementation **39**
XML
scheduling data file **107**
serializing schedule data **39**