ILOG JViews

Component Suite 5.5

Gantt User's Manual

December 2002

Table of Contents

About This Manual

This manual presents the Gantt module of ILOG JViews and describes how to use its API.

What Is in This Manual

This manual contains the following chapters:

- Chapter 1, *Introducing the Gantt Module* describes the four entities on which the Gantt module is based.
- Chapter 2, *Basic Concepts* presents the architecture of the data model and the Gantt Beans.
- Chapter 3, *Getting Started with the Gantt Module* is a detailed presentation of the two basic examples that illustrate the default implementation of a Gantt chart and a Schedule chart.
- Chapter 4, *Advanced Features* is based on the third example, which shows some of the customizing features offered by the Gantt Chart module.
- Chapter 5, *Load-on-Demand* presents a mechanism, called load-on-demand, used for loading data into memory as it is needed for display.
- Chapter 6, *Schedule Data Serialization and Exchange with SDXL* describes the ILOG JViews Gantt package that allows users to serialize schedule data to SDXL files.

- Chapter 7, *Styling* describes the CSS mechanism as applied to control the appearance of Gantt or Schedule charts.
- Chapter 8, *The Gantt Printing Framework* describes how to use the print framework to print the Gantt.
- Chapter 9, *Thin-Client Support for Web Applications* describes how to create lightweight DHTML clients that interact with Gantt applications running on the server.

The appendixes provide auxiliary and reference information as follows:

- Appendix A, *Document Type Definition for SDXL* gives the Document Type Definition of the SDXL language.

At the end of the manual you will find a *Glossary* containing definitions of the basic technical terms used in this manual.

Notes

The following conventions apply throughout this manual:

- The terms “Gantt Chart” and “Schedule Chart” refer to the two high-level Beans provided as default implementations of the Gantt module.
- On the other hand, the terms “Gantt chart” and “Schedule chart” refer to two different ways of representing the scheduling information contained in the data model. See Chapter 1, *Introducing the Gantt Module* for more information.

CHAPTER

1

Introducing the Gantt Module

The Gantt module of ILOG JViews consists of a library of JViews graphic framework-based classes meant to display an abstract *data model* of scheduling information as a Gantt chart. A Gantt chart is a type of schedule diagram where data from a table is displayed as horizontal bars along a time scale.

The Gantt module has been designed with a clear separation between the data model and its visualization. To reflect this, this chapter is divided as follows:

- "The Data Model" presents the scheduling information model whose content is displayed as Gantt charts.
- "The Charts" presents the graphical representation part of the Gantt module.

The Data Model

The data model is the part of the Gantt module that contains the scheduling information you want to display. Scheduling data consists of four abstract entities:

- Activities
- Constraints
- Resources

- Reservations

The Gantt module comes with default implementations of these entities. However, if these are not suited to your particular application, you can create your own user-defined entities/implementations (see Chapter 4, *Advanced Features* for a customization example).

Activities

An activity is a task that must be completed. Activities are hierarchical in nature. This means that a main activity, called *parent activity*, can be broken down into subactivities, called *child activities*.

In addition to its name and identifier, an activity is defined by its start time and end time, which determine an interval called the *duration* of the activity. If the start and end times are identical, the duration is equal to 0. A zero-duration activity is commonly called a *milestone*. Typically, milestones are not rendered by the same graphic object as activities with a non-zero duration.

Constraints

A constraint is a type of condition set between two activities. Constraints can have one of the following types: start-to-start, start-to-end, end-to-start, or end-to-end. The source activity—that is, the activity whose start or end controls the start or end of another activity—is called the *From* activity. Conversely, the target activity—that is, the activity whose start or end depends on the start or end of another activity—is called the *To* activity.

Constraints are represented by arrowed polyline links (see the class `IlvConstraintGraphic`).

Resources

Resources are means that enable an activity to be completed. They can be persons, premises, equipment, and so forth. Like activities, resources are also hierarchical in nature. For example, if resources are people, the parent resource is a department while the child resources are the individual employees. Likewise, you may want to group resources by physical location or by type of machinery.

Reservations

When a resource is assigned to an activity, this assignment is called a *reservation*. In the terminology of the Gantt module, a reservation represents the assignment of *one* resource to *one* activity. An activity can have multiple resources reserved and similarly, a resource can be reserved for more than one activity. The activity that reserves the resource cannot be changed after the reservation is created.

The Charts

Using the full power of the Gantt module API, you can view a data model containing scheduling information through a wide variety of graphical representations. The Gantt Chart and Schedule Chart Beans encapsulate the most commonly used scheduling displays.

The Gantt Chart Bean is designed to show activities while the Schedule Chart Bean is designed to show resources (see the figures in sections *Gantt Chart* and *Schedule Chart*). This section describes the common features shared by both representations as well as what differentiates them.

> ***Note:*** *Visualization services provided by the ILOG JViews Gantt module are based solely on Swing components, which work with light-weight user interfaces.*

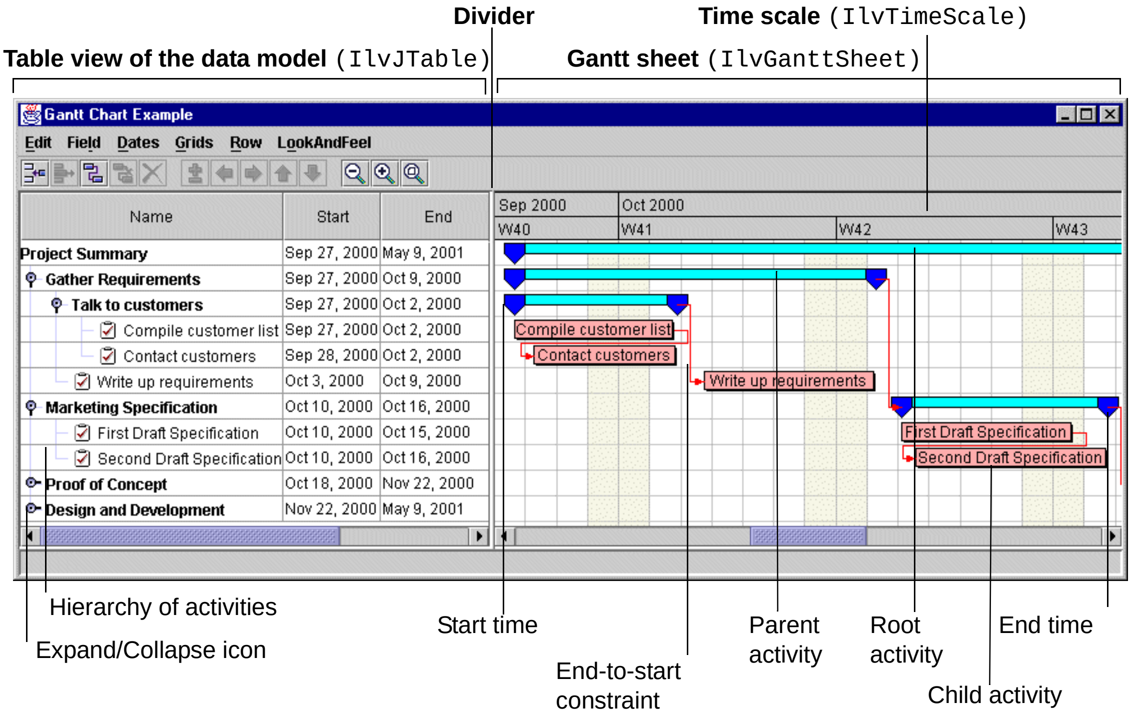
Common Features

Gantt charts are instances of the `IlvGanttChart` class and Schedule charts are instances of the `IlvScheduleChart` classes. Both are subclasses of the `IlvHierarchyChart` class, which itself derives from the Swing class `JPanel`. The charts share the following features (see Figure 1.1, for example):

- The left-hand part of either chart is a table view, an instance of `IlvJTable`, which is a subclass of the standard Swing class `JTable`.
- The right-hand part is a Gantt sheet, an instance of the class `IlvGanttSheet`, which is a special `IlvManagerView` object.
- Just above the Gantt sheet appears a zoomable time scale (an instance of the class `IlvTimeScale`).
- A standard, adjustable divider separates the left-hand part from the right-hand part.

Gantt Chart

A Gantt chart is designed to show activities. There is one row for each activity.
The hierarchy table on the left displays activity information from the data model. The Gantt sheet on the right shows how the activities are positioned on the time scale.



Name	Start	End
Project Summary	Sep 27, 2000	May 9, 2001
Gather Requirements	Sep 27, 2000	Oct 9, 2000
Talk to customers	Sep 27, 2000	Oct 2, 2000
Compile customer list	Sep 27, 2000	Oct 2, 2000
Contact customers	Sep 28, 2000	Oct 2, 2000
Write up requirements	Oct 3, 2000	Oct 9, 2000
Marketing Specification	Oct 10, 2000	Oct 16, 2000
First Draft Specification	Oct 10, 2000	Oct 15, 2000
Second Draft Specification	Oct 10, 2000	Oct 16, 2000
Proof of Concept	Oct 18, 2000	Nov 22, 2000
Design and Development	Nov 22, 2000	May 9, 2001

Divider

Time scale (`IlvTimeScale`)

Table view of the data model (`IlvJTable`)

Gantt sheet (`IlvGanttSheet`)

Hierarchy of activities

Expand/Collapse icon

Start time

End-to-start constraint

Parent activity

Root activity

End time

Child activity

Figure 1.1 The Gantt Chart Example Application

Each row represents an activity. Each column of the table displays a property of the activity. Each row in the Gantt sheet contains an activity graphic (an instance of the class `IlvActivityGraphic`) that represents the duration of the activity. A row can also display other properties of the activity, such as start time and end time.

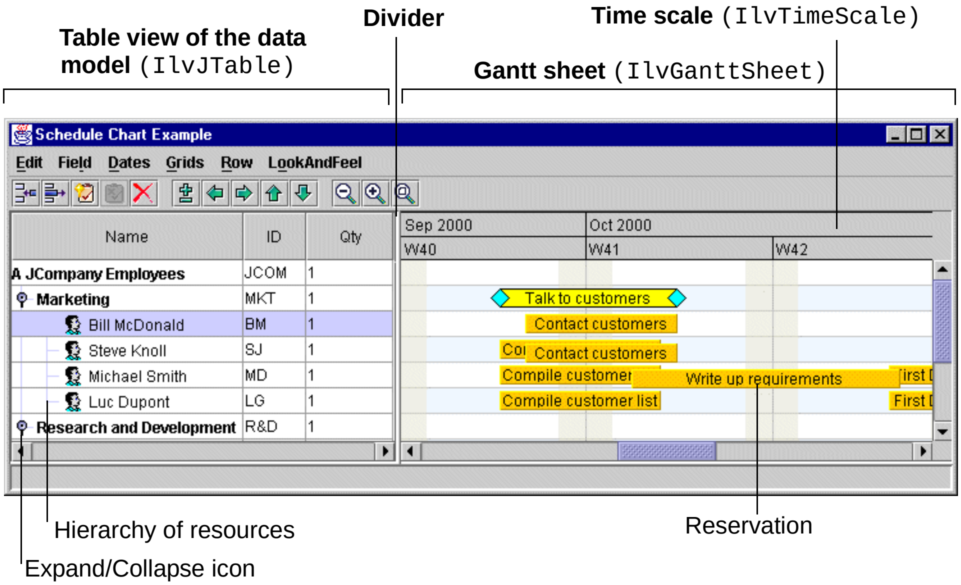
In the default implementation, activities with no children are displayed as simple horizontal bars labeled with the activity name. Activities with children are displayed as horizontal bars of a different color, delimited by special symbols at the end. These attributes are completely customizable.

In a Gantt sheet, constraints between activities are represented by directional polyline links. The type of the constraint determines how the link is attached to the activity graphics.

In the default implementation of the Gantt chart, resources and reservations are not represented in the Gantt sheet. Instead, the resources reserved by each activity are displayed in the `Resources` column of the table.

Schedule Chart

A Schedule chart is designed to show how resources are scheduled. There is one row for each resource. The resource table on the left displays resource information from the data model. The Gantt sheet on the right shows the resource reservations.



Name	ID	Qty
A JCompany Employees	JCOM	1
Marketing	MKT	1
Bill McDonald	BM	1
Steve Knoll	SJ	1
Michael Smith	MD	1
Luc Dupont	LG	1
Research and Development	R&D	1

Figure 1.2 *The Schedule Chart Example Application*

Each row represents a resource. Each column in the table displays a property of the resource. Each row in the Gantt sheet contains 0, 1, or more reservation graphics to represent the activities for which the matching resource has been reserved.

Because the same resource can be reserved for more than one activity during the same time span (see *Reservations* on page 12), it could happen that several reservation graphics occupy the same horizontal area in the same row. To address this problem, a specific activity layout algorithm is used to position the bars for best legibility. Four different layout algorithms are provided to manage potentially overlapping reservation graphics. See *Activity Layouts* on page 47 for more information.

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. Constraint links can be displayed in the Schedule Chart if each activity reserves at most 1 resource.

In the next chapter, you will learn more about the architecture of the data model and the Gantt Beans.

CHAPTER

2

Basic Concepts

In the previous chapter, you learned that the Gantt module offers predefined Beans to get both activity-oriented and resource-oriented Gantt representations of an abstract data model based on four entities: activities, constraints, resources, and reservations. You are now going to learn more about these basic concepts.

This chapter is divided as follows:

- The Data Model Architecture
- The Gantt Beans

The Data Model Architecture

Figure 2.1 shows how the separable model architecture of the Gantt module allows different data model implementations to be bound to different visualizations.

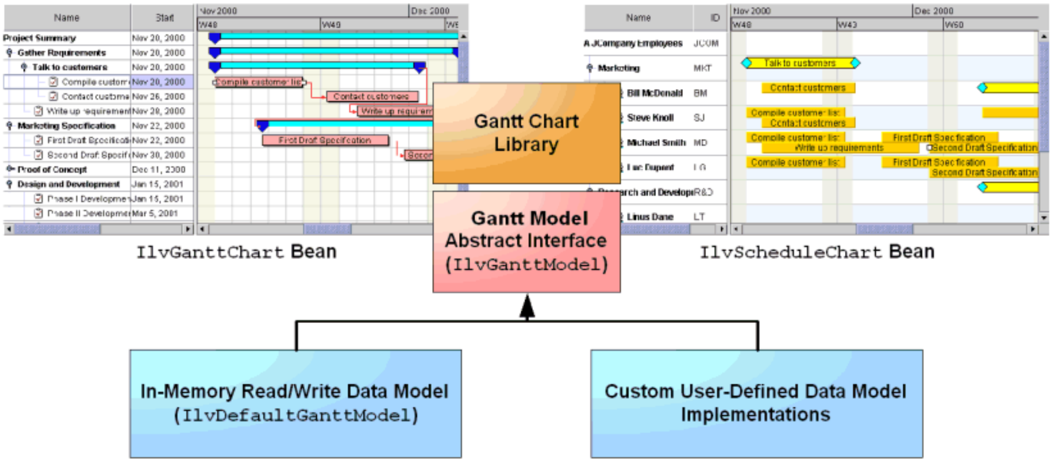

Gantt Chart Library
Gantt Model Abstract Interface (IlvGanttModel)
IlvGanttChart Bean
IlvScheduleChart Bean
In-Memory Read/Write Data Model (IlvDefaultGanttModel)
Custom User-Defined Data Model Implementations

Figure 2.1 *Gantt Module Architecture*

Model-View Separation

The traditional design of user-interface objects divides each component into three parts: model, view, and controller (hence its name of MVC architecture). In this classic design, the model manages the data or values represented by the component, the view manages the way the component is displayed, and the controller handles user interaction with the component. The Gantt module is based on the Swing variant of MVC called *separable model architecture*. This design provides all the benefits of complete model-view separation while being easier to use because it bundles the view and controller parts together.

Just as each Swing component defines the abstract model interface that it represents, the Gantt module defines an abstract scheduling data interface that it is able to represent. We refer to this interface or one of its concrete implementations as the Gantt *data model*. If you are familiar with the standard Swing set of components, you are then also familiar with this concept. For example, the `JList` component is a visualization of data defined by the abstract `ListModel` interface. In a similar manner, the Gantt module displays data defined by the abstract `IlvGanttModel` interface. This interface is described in the next section.

Data Model Classes

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`. All five interfaces are included in the `ilog.views.gantt` package. A brief description of each follows:

- `IlvGanttModel` interface: defines the overall Gantt data model and is a container for the other four entities.
- `IlvActivity` interface: represents an activity or task that must be completed in the schedule.
- `IlvConstraint` interface: represents an activity-to-activity scheduling constraint.
- `IlvResource` interface: represents a resource that can be allocated to an activity to enable its completion.
- `IlvReservation` interface: represents the allocation of a resource to an activity.

> ***Reminder:*** *You can read section The Data Model on page 11 for a reminder of the basic entities.*

Figure 2.2 shows the relationships between the five interfaces that compose the Gantt data model:

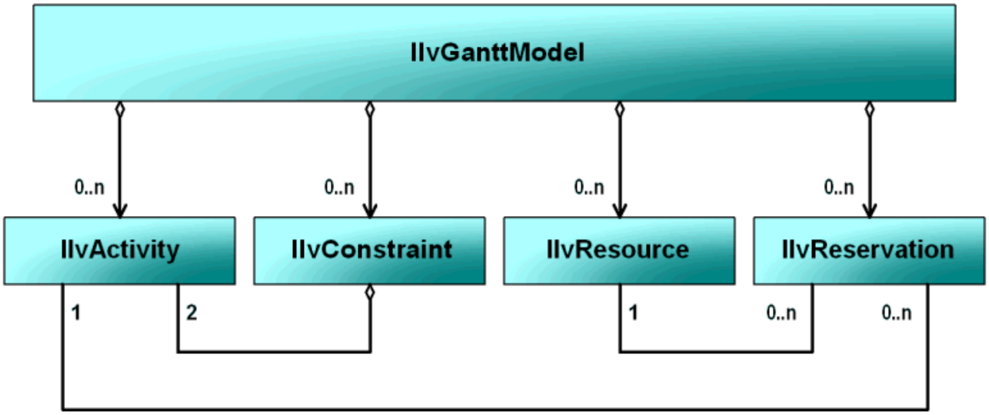


Figure 2.2 *Relationships Between Gantt Data Model Interfaces*

For each of these abstract interfaces, three levels of implementation are available to the user. 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. How to extend these abstract classes and create your own custom data model is an advanced topic not covered in this manual. You can, however, see a demonstration of this by examining the Database examples that are installed in:

```
<installdir>/demos/gantt/database
```

Also provided are two concrete data model implementations that are completely memory-based. The first is the simple default data model implementation used throughout the examples, except for the Database and CSS examples. The second is a general data model implementation that supports user-defined properties and is used in the CSS example.
A summary of the provided data model interfaces and implementation classes is shown in Table 2.1. The data model interfaces are included in the `ilog.views.gantt` package. The abstract and default data model implementations are included in the `ilog.views.gantt.model` package. The general data model implementation is included in the `ilog.views.gantt.model.general` package. How to extend the concrete data model implementations is an advanced topic not covered in this section (see Chapter 4, *Advanced Features* for a customization example).

Table 2.1 *Gantt Data Model Interfaces and Provided Implementations*

Data Model Interface	Abstract Implementation	Default Memory-Based Implementation	General Memory-Based Implementation
`IlvGanttModel`	`IlvAbstractGanttModel`	`IlvDefaultGanttModel`	
`IlvActivity`	`IlvAbstractActivity`	`IlvSimpleActivity`	`IlvGeneralActivity`
`IlvResource`	`IlvAbstractResource`	`IlvSimpleResource`	`IlvGeneralResource`
`IlvConstraint`	`IlvAbstractConstraint`	`IlvSimpleConstraint`	`IlvGeneralConstraint`
`IlvReservation`	`IlvAbstractReservation`	`IlvSimpleReservation`	`IlvGeneralReservation`

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.

> ***Note:*** *You can create your own data model in its entirety, but we recommend using the abstract classes as a starting point. See the Database examples to see how to do this.*

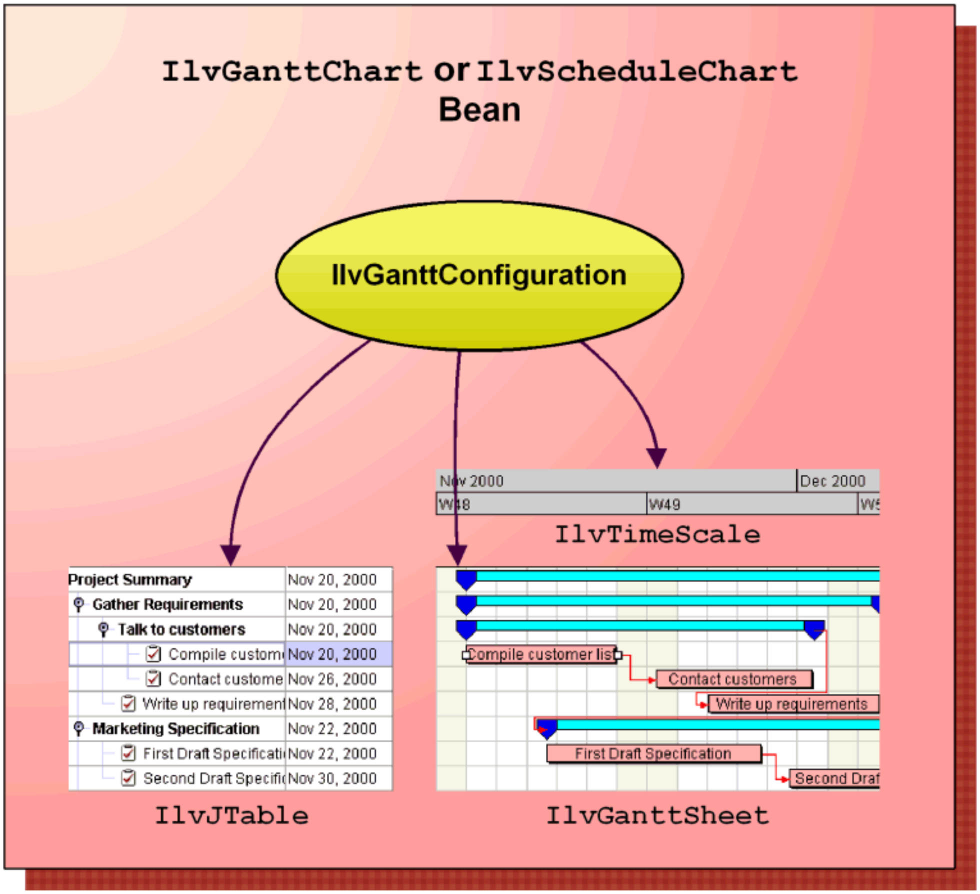
The Gantt Beans

The ILOG JViews Gantt module features two high-level Beans, called Gantt Chart Bean and Schedule Chart Bean. Their API is based on the classes `IlvScheduleChart` and `IlvGanttChart`, both subclasses of `IlvHierarchyChart`. Chapter 3, *Getting Started with the Gantt Module*, provides more detail on the API of these three classes.

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 three main classes of the Gantt module API.

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 illustrated in Figure 2.3:



Project Summary	Nov 20, 2000
Gather Requirements	Nov 20, 2000
Talk to customers	Nov 20, 2000
☑ Compile custom	Nov 20, 2000
☑ Contact custome	Nov 26, 2000
☑ Write up requirement	Nov 28, 2000
Marketing Specification	Nov 22, 2000
☑ First Draft Specificati	Nov 22, 2000
☑ Second Draft Specific	Nov 30, 2000

Figure 2.3   Structure of Gantt Beans

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` method, inherited by `IlvGanttChart` and `IlvScheduleChart` from their base class `IlvHierarchyChart` (`ilog.views.gantt` package).

However, 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`.

In the next chapter, you will start with two examples and learn more about:

- the time API,
- the data model API,
- the Gantt Beans.

CHAPTER

3

Getting Started with the Gantt Module

In this chapter, you will start with two examples and learn more about the time API, the data model API, the Gantt Beans, and the specific features of the Gantt sheet.

This chapter covers the following topics:

- Basic Steps for Using the Gantt Chart and Schedule Chart Beans
- Examples
- Time and Duration
- The Gantt Data Model
- The Gantt Beans

Basic Steps for Using the Gantt Chart and Schedule Chart Beans

As mentioned in Chapter 1, *Introducing the Gantt Module*, 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. Import the necessary Gantt packages.
2. Create a Gantt data model object by instantiating the interface `IlvGanttModel` and fill it with activities, constraints, resources, and reservations.

3. Instantiate a Gantt Chart Bean from the `IlvGanttChart` class or a Schedule Chart Bean from the `IlvScheduleChart` class.
4. Attach the data model to the chart instance.
5. Customize the default settings and appearance of the chart, if necessary.
6. 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, `GanttExample.java`, demonstrates how to use the activity-oriented Gantt chart while the second example, `ScheduleExample.java`, demonstrates how to use the resource-oriented Schedule chart. Section *Examples* 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 the Gantt module. In fact, these basic chart applications are used as the basis for all the other examples supplied with the Gantt module.

Examples

This section describes how to run the two chart examples supplied with the Gantt module and shows how the source code for the examples implements the six basic steps.

Gantt Chart

The basic steps for using the Gantt Chart Bean are illustrated in an example Java application provided. The source code file of the example is named `GanttExample.java` and can be found in the directory:

```
<installdir>/demos/gantt/charts
```

To run the example, ensure that the Ant utility is properly configured. If not, see the instructions on how to configure Ant for ILOG JViews in:

```
<installdir>/html/installation.html
```

Then, you can go to the directory where the example is installed and type:

```
ant rungantt
```

to run the example as an application.

Most of the code in the Gantt Chart example is to handle 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);
  }
  ...
}
```

Step 1 – Importing the Gantt Chart Packages

In the `GanttExample.java` file, we first import the packages that are common to all the Gantt examples:

```
import shared.*;
import shared.data.*;
import shared.swing.*;
```

Then we import the Gantt Chart package that we need:

```
import ilog.views.gantt.*;
import ilog.views.gantt.action.*;
import ilog.views.gantt.model.*;
import ilog.views.gantt.property.*;
import ilog.views.gantt.swing.*;
```

We also import the various Swing and AWT packages necessary to build the rest of the user interface of the example:

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
```

Both the Gantt Chart example and the Schedule Chart example (see *Schedule Chart* on page 28) derive from a common superclass, `AbstractExample`, that is itself a subclass of `AbstractExample`. These classes contain the code that is shared between all the Gantt examples. Because the example 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);
  }
  ...
}
```

Step 2 – Creating the Gantt Data Model

We create and populate the data model in the `createGanttModel` method of the `AbstractGanttExample` superclass. In this manner, all of the Gantt examples create the same data model unless they specifically override this method:

```
IlvGanttModel model = createGanttModel();
...
protected IlvGanttModel createGanttModel()
{
  ...
}
```

In the `createGanttModel` method, we first instantiate the class `IlvDefaultGanttModel`:

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

This creates an empty in-memory data model. The next step is to populate the data model with scheduling information, which is done in the `populateGanttModel` method. More information about this task is provided in section *The Gantt Data Model* on page 32.

As an introduction, here is how you might add the first two activities to the data model:

```
IlvActivity rootActivity = new IlvSimpleActivity(...);
model.setRootActivity(rootActivity);
IlvActivity childActivity = new IlvSimpleActivity(...);
model.addActivity(childActivity, rootActivity);
```

Step 3 – Creating the Gantt Chart Bean Instance

Next, you create an instance of the Gantt Chart Bean by implementing the `createChart()` method:

```
protected IlvHierarchyChart createChart()
{
  return new IlvGanttChart();
}
```

Step 4 – Binding the Gantt Chart to the Data Model

Binding the Gantt Chart Bean to the data model enables the chart to display the contents of the data model:

```
chart.setGanttModel(model);
```

Step 5 – Customizing the Chart

The chart is customized in the `customizeChart()` method. In the `GanttExample.java` file, we perform 5 basic customizations:

1. Several activities are expanded so that all their children are initially visible. For example:

   ```
   chart.expandAllRows(anActivity);
   ```

2. The width of the table columns are increased. For example:

```
chart.getTable().getColumn("Name").setPreferredWidth(175);
```

3. The time interval initially displayed by the chart is set:

```
chart.setVisibleTime(aDate);
chart.setVisibleDuration(new IlvDuration(...));
```

4. An instance of `WeekendGrid` is set as the background vertical grid of the chart:

```
chart.getGanttSheet().setVerticalGrid(new WeekendGrid());
```

5. 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, you could change the default height of the displayed rows:

```
chart.setRowHeight(25);
```

Or change the font used to label the horizontal time scale:

```
chart.setTimeScaleFont(new Font(...));
```

Step 6 – Adding the Gantt Chart to the User Interface

The Gantt Chart must now be displayed. We add it to the center of the container panel provided as the argument to the `init` method, which we have set to have a `BorderLayout` attribute:

```
container.add(gantt, BorderLayout.CENTER);
```

The container panel will be the `contentPane` of the JApplet when the example is run as an applet or it will be the `contentPane` of the frame when the example is run as an application.

Schedule Chart

The basic steps for using the Schedule Chart Bean are illustrated in another example Java application. They are almost exactly the same as those for the Gantt Chart Bean, described in the *Gantt Chart* section. The source code of the example is named `ScheduleExample.java` and can be found in the same directory:

```
<installdir>/demos/gantt/charts
```

To run the example, ensure that the Ant utility is properly configured. If not, see the instructions on how to configure Ant for ILOG JViews in:

```
<installdir>/html/installation.html
```

Then, go to the directory where the example is installed and type:

```
ant runschedule
```

to run the example as an application.

The Schedule Chart example is almost identical to the Gantt Chart example. 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 example:

```
 ...
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 the Gantt module are located in the JAR file:

`<installdir>/classes/gantt.jar`

The graphic presentations of the Gantt module are based upon the ILOG JViews Graphics Framework.You will also need to include these classes located in:

`<installdir>/classes/jviews.jar`

The jviewsall.jar File

The `jviewsall.jar` file is supplied to allow you to easily integrate all the ILOG JViews Beans into the palette of your favorite IDE. See the instructions in Chapter *"Graphics Framework JavaBeans"* of the *Graphics Framework User's Manual* for more details on how to do this. It includes:

- all the classes from the files `jviews.jar` and `gantt.jar`,
- all the other ILOG JViews modules.

Time and Duration

Throughout the Gantt module, time is represented by the standard `java.util.Date` class, duration by `IlvDuration`, and time intervals by `IlvTimeInterval`. This section discusses how to instantiate and use these classes.

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. The Gantt module 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)`

Here is an example of 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:*** *Note, that 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)
```

Here is an example of 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)
```

Here is an example of 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)
```

The Gantt Data Model

As explained in section *The Data Model Architecture* on page 17, the Gantt module data model is designed with complete model-view separation.

This section is divided as follows:

- "Class Overview" discusses the overall architecture of the Gantt *data model*.
- "Binding the Gantt Chart Beans to the Data Model" illustrates how to bind the Gantt Chart Beans to the data model.
- "Populating the Data Model" explains how to populate the data model with scheduling information.

Class Overview

The scheduling data displayed by the Gantt Chart and Schedule Chart Beans are defined by the abstract `IlvGanttModel` interface, which defines the overall data model and acts as an intelligent container for the other four data model entities (see Figure 2.1 on page 18), namely:

- Activities are 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.

This section shows you how to:

- Bind a Gantt chart or Schedule chart to a data model.
- Populate the data model with scheduling data.

> ***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.*

Binding the Gantt Chart Beans to the Data Model

As illustrated in section *Examples*, use the following method to bind an `IlvGanttChart` or `IlvScheduleChart` object to a data model:

```
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, which is the common superclass of both charts.

Populating the Data Model

The data model can be populated before or after a chart has been bound to it. Initial data will be immediately displayed by the chart when it binds to the data model. Data populated after the chart has been bound will cause the chart to update dynamically to reflect the new data in the data model. Because of this, if you have a large dataset we recommend that you populate your data model before the chart is bound. This will provide the best performance.

Activities and Resources

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 *resource*. Conversely, an activity or resource with no children is called a *leaf activity* or *resource*. The hierarchical trees of activities and resources must start somewhere and we call these 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 first step in populating a data model with activities is to establish the root `IlvActivity` object:

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

At this point, you can now 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 we add a child activity to the root activity that was created in the data model:

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

In an identical manner, the first step in populating the data model with resources is to establish the root `IlvResource` object:

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

At this point, you can now 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 we add a child resource to the root resource just added to the data model:

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

By continuing to add activities and resources in this manner, you can populate the Gantt data model with your scheduling data. The following `IlvGanttModel` methods allow you to manipulate the activities and resources within the data model:

Table 3.1 *IlvGanttModel Methods for Populating a Data Model with Activities and Resources*

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

The Activity and Resource Factories

The previous section shows how you can populate the data model by explicitly instantiating concrete implementations of the `IlvActivity` and `IlvResource` interfaces. In this case, the code created instances of `IlvSimpleActivity` and `IlvSimpleResource`, the default implementations provided with the Gantt module. However, the Gantt Chart and Schedule Chart Beans provide a more 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 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 (see *Manipulating Gantt Data with the Gantt Sheet* on page 48 for more information on interactors). By default, the chart activity factory is an instance of `IlvSimpleActivityFactory`. However, 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 IlvSimpleActivity(...);
```

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

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

> ***Note:*** *The class* `IlvSimpleActivityFactory` *creates each new activity with a default name and identifier of "New Activity". You will probably want to modify these default attributes before adding the new activity to your data model.*

The Gantt Chart examples use this technique to populate the data model.

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 `IlvSimpleResourceFactory`. 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* `IlvSimpleResourceFactory` *class creates each new resource with a default name and identifier of "New Resource". 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* on page 12). 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 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 IlvSimpleActivity(...);
ganttModel.setRootActivity(rootActivity);
IlvActivity child1 = new IlvSimpleActivity(...);
ganttModel.addActivity(child1, rootActivity);
IlvActivity child2 = new IlvSimpleActivity(...);
ganttModel.addActivity(child2 rootActivity);
// Create a constraint between child1 and child2
IlvConstraint constraint =
    new IlvSimpleConstraint(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 IlvSimpleConstraintFactory 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. 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 IlvSimpleActivity(...);
ganttModel.setRootActivity(rootActivity);
IlvActivity childActivity = new IlvSimpleActivity(...);
ganttModel.addActivity(childActivity, rootActivity);
IlvResource rootResource = new IlvSimpleResource(...);
ganttModel.setRootResource(rootResource);
IlvResource childResource = new IlvSimpleResource(...);
ganttModel.addResource(childResource, rootResource);
// Create a reservation between childActivity and childResource
IlvReservation r = new IlvSimpleReservation(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 *The Activity and Resource Factories* on page 35 and *The Constraint Factory* on page 37, 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 `IlvSimpleReservationFactory` 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);
```

The Gantt Beans

This section discusses the Gantt and Schedule Chart Beans that are provided with this module. Both `IlvGanttChart` and `IlvScheduleChart` are subclasses of the common base class `IlvHierarchyChart`. Therefore, both chart Beans have a similar architecture and have many properties and attributes in common.

This section is divided as follows:

- “Chart Visual Properties” discusses such properties as color, font, scroll bar, and so on.
- “Expanding, Collapsing, and Hiding/Showing Rows” describes the properties of the hierarchical structure of activities or resources.
- “Scrolling in the Gantt Sheet” describes the API for horizontal and vertical scrolling in the Gantt sheet.

Chart Visual Properties

The Gantt and Schedule Chart Beans have many properties that control their visual appearance. For example, to change the font used in the column headers of the table, you can use:

```
myChart.setTableHeaderFont(new Font(...));
```

Or, if you want to change the foreground color of the horizontal time scale to blue you can use:

```
myChart.setTimeScaleForeground(Color.blue);
```

Here is a list of the `IlvHierarchyChart` methods that control the visual properties of the charts:

Table 3.2 *Methods for Control of Chart Visual Properties*

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)`

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)`

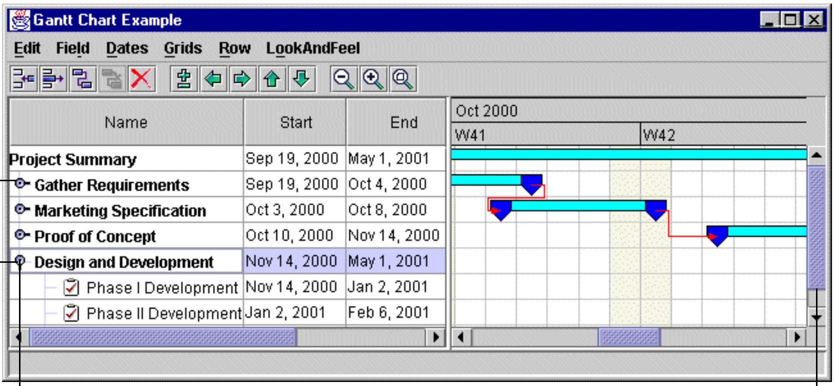
Expanding, Collapsing, and Hiding/Showing Rows

The `IlvGanttChart` class displays the hierarchical structure of activities such that each row represents one activity. Similarly, the `IlvScheduleChart` class displays the hierarchical tree of resources so that each row represents a resource. In both charts, rows are numbered from 0 starting with the root activity or resource. Row numbering ignores vertical scrolling and is not affected by the current vertical scrolling position of the chart. For the rest of this discussion, we will refer to the activities in an `IlvGanttChart` and to the resources in an `IlvScheduleChart` as *data nodes*. This will allow us to talk about the common behavior of both charts in a concise manner. We also define the following terms related to the visibility of data nodes and the rows they are displayed on:

- An *expanded* data node is one that is visible and shows its children, making them visible also.
- A *collapsed* data node is one that hides its children. 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 children, 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.

> ***Note:** Scrolling through a window changes the **display** status of a row, not its **visibility** status.*

- A *hidden* data node is the opposite of visible. It is a child of a collapsed parent and is not represented by a row.



Name	Start	End
Project Summary	Sep 19, 2000	May 1, 2001
Gather Requirements	Sep 19, 2000	Oct 4, 2000
Marketing Specification	Oct 3, 2000	Oct 8, 2000
Proof of Concept	Oct 10, 2000	Nov 14, 2000
Design and Development	Nov 14, 2000	May 1, 2001
Phase I Development	Nov 14, 2000	Jan 2, 2001
Phase II Development	Jan 2, 2001	Feb 6, 2001

Collapsed data node. (Its child activities are hidden.)

Expanded data node

The first two child activities of this expanded data node are both visible and displayed. You can tell from the scroll bar that the data node has more visible child activities that are not displayed.

***Figure 3.1** Expanded/Collapsed and Visible/Display Statuses*

You can use the `IlvHierarchyChart` methods listed in Table 3.3 to access and control the expand, collapse, and visibility status of the rows in a chart.

> ***Reminder:*** `IlvHierarchyNode` *is the abstract superclass of both activities and resources.*

The actual value used by the API will be an activity for an `IlvGanttChart` and a resource for an `IlvScheduleChart`.

Table 3.3 *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

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)`

For example, you can scroll a chart horizontally to the beginning of an activity:

```
IlvActivity activity = ...
IlvTime 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.
- *Bounded* mode is enabled when the methods `setMinVisibleTime` and `setMaxVisibleTime` 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()
```

Displaying Gantt Data in the Gantt Sheet

The Gantt sheet is the right part of a Gantt Chart or Schedule Chart Bean and as such, it is an important graphic component. It is designed both for the graphical display of the Gantt data in a Gantt model and as an interactive interface that allows end users to manipulate the Gantt data by means of interactors implemented for this purpose.

This section is divided as follows:

- *Gantt Sheet Architecture* illustrates the architecture of the Gantt sheet and introduces some important concepts related to the Gantt sheet. The main purpose of this subsection is to outline how the Gantt data appears in a Gantt sheet.
- *Describing the Gantt Sheet* illustrates several predefined interactors designed for the Gantt sheet. This subsection shows end users how to manipulate Gantt data displayed in a Gantt sheet.
- *Activity Layouts* describes the four layouts you can choose from to handle the way activity graphics are arranged in the Gantt rows of a Schedule chart.

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`. The Gantt sheet is illustrated in sections *Activity Gantt Sheet* on page 45 and *Resource Gantt Sheet* on page 46.

The Gantt sheet is a user interface component designed for two main purposes:

- to display the data of a given Gantt model graphically, namely:
  - activities and constraints in a Gantt chart, or

  - 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.

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` to enumerate the visible ones.
- A Gantt row contains one or more activity graphics to represent activities.

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 section *Activity Renderers*). The activity renderer defined for an activity graphic can be accessed or changed by the `getActivityRenderer` and `setActivityRenderer` methods of the `IlvActivityGraphic` class. (See Figure 3.2).

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` method of the `IlvActivityRenderer` interface is called.

Describing the Gantt Sheet

The data of a given Gantt model is rendered differently depending on whether the Gantt sheet is in a Gantt chart or in a Schedule chart.

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* on page 12 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.

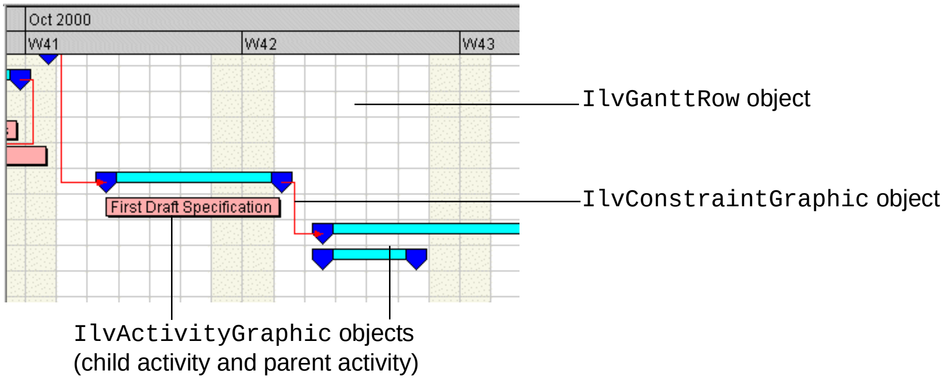

Oct 2000
W41
W42
W43
First Draft Specification
IlvGanttRow object
IlvConstraintGraphic object
IlvActivityGraphic objects
(child activity and parent activity)

Figure 3.2 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.

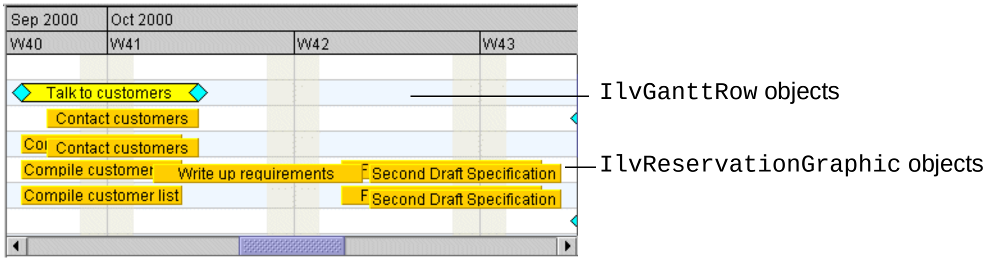


Figure 3.3 Gantt Sheet in a Schedule Chart

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 1 resource, constraint links can be displayed in the Schedule Chart by calling the `setDisplayingConstraints` method.

Activity Layouts

As explained in section *Resource Gantt Sheet* on page 46, 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, you can choose among three other layout algorithms. The following layouts are presented:

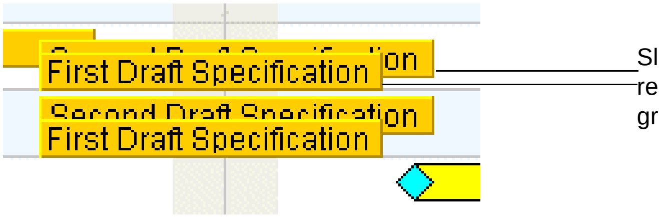
- *Simple Layout*
- *Pretty Layout*
- *Tile Layout*
- *Cascade Layout*

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).

Pretty Layout

Figure 3.4 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.


First Draft Specification
Second Draft Specification
First Draft Specification

Slightly offset reservation graphics

Figure 3.4 *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.

Tile Layout

Figure 3.5 shows activity graphics in Tile layout. Compare the fourth row with the one in Figure 3.4. The concurrent reservation graphics are placed at different y positions. The height of each reservation graphic is calculated so that it does not overlap.

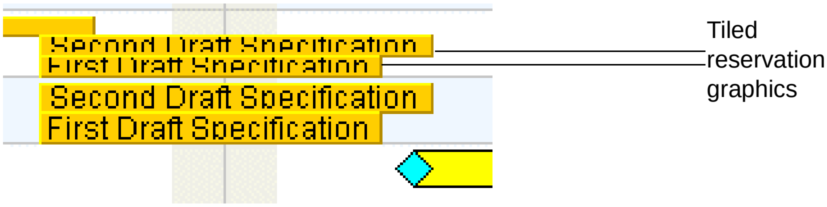

Second Draft Specification
First Draft Specification
Second Draft Specification
First Draft Specification
Tiled reservation graphics

Figure 3.5 *Reservation Graphics in Tile Layout*

Manipulating Gantt Data with the Gantt Sheet

Several predefined interactors are implemented in the Gantt sheet for the following purposes:

- Selecting Graphics
- 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. Figure 3.6 shows a selected activity graphic and a selected constraint graphic.

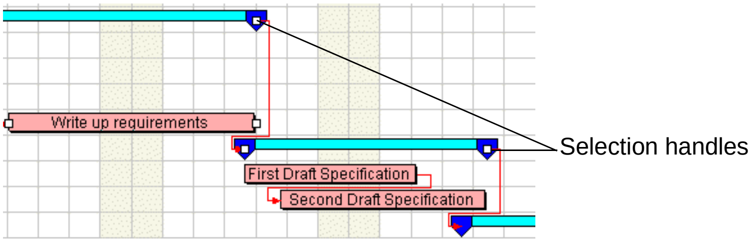


Figure 3.6 *Selected Activity Graphic and Constraint Graphic*

Installing the Selection Interactor

A predefined interactor, `IlvGanttSelectInteractor`, handles the graphic selection. To install this interactor, call the `pushInteractor` 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 “manually”.

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 on 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 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. Notice 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.

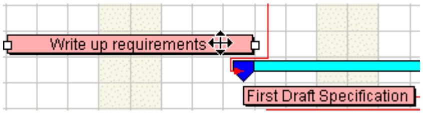


Figure 3.7 *Moving a Selected Activity Graphic*

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 do so, press the Ctrl key 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.

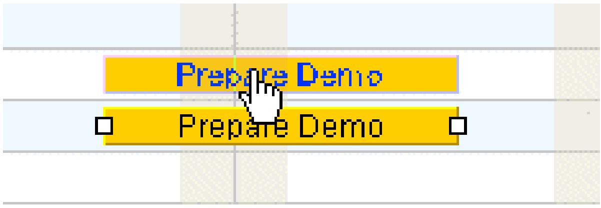


Figure 3.8 *Duplicating a Reservation Graphic*

To abort duplication while you are dragging the mouse, press the Escape key.

Resizing Activity and Reservation Graphics

A selected activity or reservation graphic is marked by two handles. You can resize a selected graphic by 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.

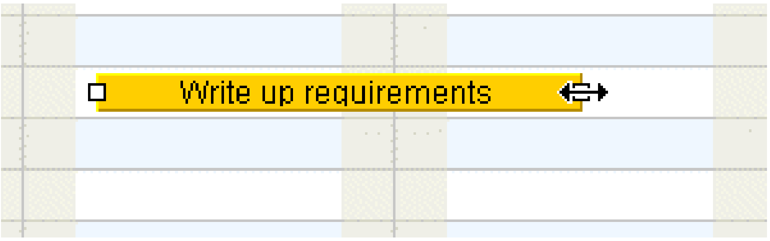


Figure 3.9 Resizing a Selected Activity Graphic

Interacting with the Gantt Sheet Using the Mouse

Creating Activities and Reservations

In a Schedule chart, you can also create activity and reservation graphics using the mouse. To do so, you must install the appropriate interactor to the Gantt sheet:

1. Create an instance of the class `IlvMakeActivityInteractor`.
2. Attach this interactor to the Gantt sheet by calling the `pushInteractor` 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` 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.

CHAPTER

4

Advanced Features

This chapter leads you through several overviews and tutorials on how to do advanced customizations of the Gantt module. The following topics are covered:

- The Custom Gantt Example
- Customizing the Gantt Data Model
- Customized Activity Rendering
- Customized Table Columns

You can see a demonstration of advanced activity rendering using the Java2D API by examining the Java2D example located in:

```
<installdir>/demos/gantt/java2d
```

The Java2D example is not covered in this manual.

The Custom Gantt Example

The Custom Gantt Chart example is supplied with the Gantt module to illustrate several advanced customization techniques that can be applied to the charts. The `CustomGanttExample.java` example application extends the `GanttExample.java` example, previously discussed in section *Gantt Chart* on page 24. This application file simplifies the source code so that it only contains the customizations that override the default behavior of the Gantt chart. This section is divided as follows:

- *Running the Custom Gantt Example* shows you how to run the Custom Gantt example.
- *Customization Overview* gives a brief overview of the customized portions of the chart, which are discussed in more detail in subsequent sections.

Running the Custom Gantt Example

The source code file of the Custom Gantt example application is named `CustomGanttExample.java` and can be found in the directory:

```
<installdir>/demos/gantt/customData
```

To run the example, ensure that the Ant utility is properly configured. If not, see the instructions on how to configure Ant for ILOG JViews in:

```
<installdir>/html/installation.html
```

Then, go to the directory where the example is installed and type:

```
ant run
```

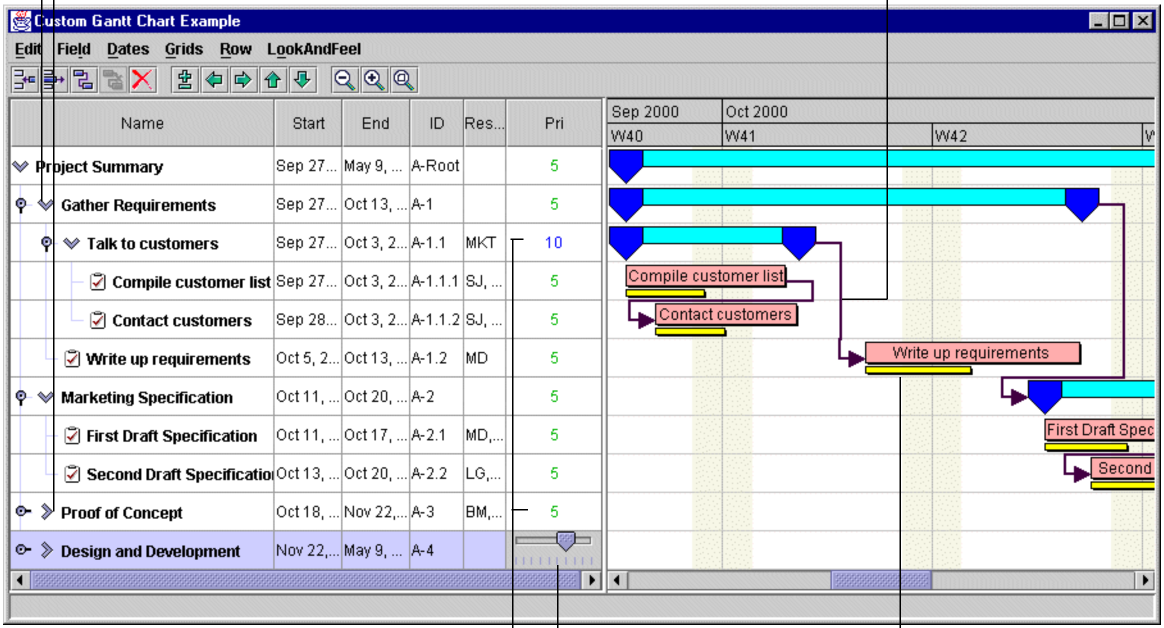
to run the example as an application.

Customization Overview

When the Custom Gantt example launches, the application looks like this:

Custom icons on parent activity rows show expanded and collapsed rows.

Constraint graphics show different colors, line widths, and connection types.



Name	Start	End	ID	Res...	Pri
Project Summary	Sep 27...	May 9, ...	A-Root		5
Gather Requirements	Sep 27...	Oct 13, ...	A-1		5
Talk to customers	Sep 27...	Oct 3, 2...	A-1.1	MKT	10
Compile customer list	Sep 27...	Oct 3, 2...	A-1.1.1	SJ, ...	5
Contact customers	Sep 28...	Oct 3, 2...	A-1.1.2	SJ, ...	5
Write up requirements	Oct 5, 2...	Oct 13, ...	A-1.2	MD	5
Marketing Specification	Oct 11, ...	Oct 20, ...	A-2		5
First Draft Specification	Oct 11, ...	Oct 17, ...	A-2.1	MD,...	5
Second Draft Specificatio	Oct 13, ...	Oct 20, ...	A-2.2	LG,...	5
Proof of Concept	Oct 18, ...	Nov 22,...	A-3	BM,...	5
Design and Development	Nov 22,...	May 9, ...	A-4		

The color of the priority level changes depending on its value.

Priority slider

The length of the yellow bar is proportional to the activity priority level.

Figure 4.1 *The Custom Gantt Chart Example Application*

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 priority property has been added to each activity in the Gantt data model. See *Customizing the Gantt Data Model* on page 56.
- Each leaf activity is rendered with a customized graphic, which represents the activity priority as a horizontal yellow bar. See *Customized Activity Rendering* on page 62.
- Each parent activity is displayed in the tree column with a custom icon that depends on whether the activity is expanded or collapsed. See *Customized Table Columns* on page 66.
- 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 *Customized Table Columns* on page 66.

Customizing the Gantt Data Model

This section takes you through a tutorial on how a priority property was added to each activity in the Custom Gantt Chart example (see the figure in *Customization Overview* on page 54). You can later apply the concepts set out hereafter to add your own properties to activities, resources, constraints, or reservations.

> ***Note:*** *Although this tutorial illustrates how to extend the* `IlvSimpleActivity` *class in order to add a custom property, you may find that the* `IlvGeneralActivity` *class provides all the functionality that you need.*

The CustomActivity Class

We start by extending the class `IlvSimpleActivity`. This gives us a memory-based activity implementation to which we can add additional properties. Our activity implementation is located in the file:

`<installdir>/demos/gantt/customData/CustomActivity.java`

First, we import the core Gantt package:

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

so that we can reference the classes `IlvSimpleActivity`, `IlvTimeInterval`, and so on. We also duplicate all of the `IlvSimpleActivity` constructors. At this point, `CustomActivity` does not provide any additional functionality and its skeleton looks like this:

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

public class CustomActivity extends IlvSimpleActivity {

  public CustomActivity (String id, String name, IlvTimeInterval interval) {
    super(id, name, interval);
  }

  public CustomActivity (String id, String name, Date start, Date end) {
    super(id, name, start, end);
  }

  public CustomActivity (String id, String name, Date start,
                         IlvDuration duration) {
    super(id, name, start, duration);
  }

}
```

We now add the priority as a private integer property to the class and provide public accessors and modifiers:

```
private int priority = 5;

public int getPriority () {
  return priority;
}

public void setPriority (int priority) {
  this.priority = priority;
}
```

We have also decided that valid priority values must be within the range from 1 to 10. As a consequence:

- We create two constants that define the minimum and maximum allowed priority values, and

  ```
  public static final int HIGHEST_PRIORITY = 1;
  public static final int LOWEST_PRIORITY = 10;
  ```

- We modify the `setPriority` method accordingly:

  ```
  public void setPriority (int priority) {
    int newPriority =
       Math.min(LOWEST_PRIORITY, Math.max(HIGHEST_PRIORITY, priority));
    this.priority = newPriority;
  }
  ```

> ***Note:** Another way of constraining the priority values would be to throw an* `IllegalArgumentException` *when the user attempts to set an invalid priority value, but trimming the value to fit within the allowed limits suits the needs of the example better.*

Adding Property Events

We have now added a priority property to the `CustomActivity` class, but there is currently no way for the Gantt chart display mechanism to know when the priority of an activity has changed so that the display can be updated. This is done by creating an event class that `CustomActivity` will fire whenever its priority is modified. Our `PriorityEvent` class is implemented in the file:

`<installdir>/demos/gantt/customData/PriorityEvent.java`

The `PriorityEvent` class extends the `ActivityPropertyEvent` class, which is the superclass of all property events fired by activity implementations. As you can guess, the following superclasses should be used when creating your own property events:

- ActivityPropertyEvent to extend activity events;
- ResourcePropertyEvent to extend resource events;
- ConstraintPropertyEvent to extend constraint events;
- ReservationPropertyEvent to extend reservation events.

The ActivityPropertyEvent class is designed to notify interested listeners of a change in an Object-type property value. However, the priority property we have created is a primitive int type. Therefore, we design our PriorityEvent extension to convert the int priority value back and forth to an Integer for storage within the event:

```
public class PriorityEvent extends ActivityPropertyEvent {

  public PriorityEvent(CustomActivity activity,
                       int oldPriority,
                       int newPriority,
                       boolean aboutToChangeEvent) {
    super(activity,
          new Integer(oldPriority),
          new Integer(newPriority),
          aboutToChangeEvent);
     }

  public int getOldPriority() {
    return ((Integer)getOldValue()).intValue();
  }

  public int getNewPriority() {
    return ((Integer)getNewValue()).intValue();
  }

  public void setNewPriority(int priority) {
    setNewValue(new Integer(priority));
  }

}
```

Next, we need to modify the setPriority method of the CustomActivity class so that it will fire a PriorityEvent to all interested listeners whenever the priority value is changed. The first thing to remember is that listeners do not register to receive events directly from each instance of IlvActivity. Rather, listeners register with the Gantt data model by using the addActivityListener method. The same is true for resources, constraints, and reservations.

The following methods of the class `IlvGanttModel` can be used to register and unregister listeners for various data model events:

Table 4.1 Methods for Registering or Unregistering Listeners

Listeners	Methods

Activity hierarchy	`void addActivityHierarchyListener(ActivityHierarchyListener aListener)`
	`void removeActivityHierarchyListener(ActivityHierarchyListener aListener)`

Activities	`void addActivityListener(ActivityListener aListener)`
	`void removeActivityListener(ActivityListener aListener)`

Resource hierarchy	`void addResourceHierarchyListener(ResourceHierarchyListener aListener)`
	`void removeResourceHierarchyListener(ResourceHierarchyListener aListener)`

Resources	`void addResourceListener(ResourceListener aListener)`
	`void removeResourceListener(ResourceListener aListener)`

Constraints	`void addConstraintListener(ConstraintListener aListener)`
	`void removeConstraintListener(ConstraintListener aListener)`

Reservations	`void addReservationListener(ReservationListener aListener)`
	`void removeReservationListenerReservationListener aListener)`

The class `CustomActivity` inherits the `fireEvent` method from the class `IlvAbstractActivity`. This method routes an event from the activity to the data model, and then to all registered listeners. Each abstract implementation of the four data model entities has a similar `fireEvent` method that subclasses can use to route events properly to interested listeners. The methods are:

```
IlvAbstractActivity.fireEvent(ActivityEvent e)
```

```
IlvAbstractResource.fireEvent(ResourceEvent e)
```

```
IlvAbstractConstraint.fireEvent(ConstraintEvent e)
```

```
IlvAbstractReservation.fireEvent(ReservationEvent e)
```

Here is our new `setPriority` method, which now fires a `PriorityEvent` once the priority value of the activity has been modified:

```
public void setPriority (int priority) {
  int newPriority =
       Math.min(LOWEST_PRIORITY, Math.max(HIGHEST_PRIORITY, priority));
  // Don't fire any events unless the priority value actually changes.
  if (this.priority == newPriority)
    return;
  // Save the old priority value for constructing the changed event
  int prevPriority = this.priority;
  // Set the new priority value
  this.priority = newPriority;
  // Create the priority changed event and fire it.
  PriorityEvent changedEvent =
       new PriorityEvent(this, prevPriority, newPriority, false);
  fireEvent(changedEvent);
}
```

Notice that the last argument to the `PriorityEvent` constructor is `false`. This creates an event to signal that the priority value has been *changed*, whereas if that last argument were set to `true`, this would signal an *about-to-change* event. This type of event is discussed in the next section.

About-to-Change Events

For many data model properties, including the priority property of our `CustomActivity` class, it is desirable to notify interested listeners just before a property value is about to be notified. This gives listeners an opportunity to constrain the proposed new value of the property or to veto the property change completely. We call this an *about-to-change* event. To create such an event, set the last argument to the event constructor to `true`.

Here is the final version of our `setPriority` method with the addition of firing a priority about-to-change event:

```
public void setPriority (int priority) {
  int newPriority =
       Math.min(LOWEST_PRIORITY, Math.max(HIGHEST_PRIORITY, priority));
  // Don't fire any events unless the priority value is modified.
  if (this.priority == newPriority)
    return;
  // Create the priority about-to-change event and fire it.
  PriorityEvent aboutToChangeEvent =
       new PriorityEvent(this, this.priority, newPriority, true);
  fireEvent(aboutToChangeEvent);
  // Check if a listener vetoed the property modification
  if (aboutToChangeEvent.isVetoed())
    return;
  // Save the old priority value for constructing the changed event
  int prevPriority = this.priority;
  // Use the potentially constrained value from the
  // about-to-change event to set the new priority
  this.priority = aboutToChangeEvent.getNewPriority();
  // Create the priority changed event and fire it.
  PriorityEvent changedEvent =
       new PriorityEvent(this, prevPriority, newPriority, false);
  fireEvent(changedEvent);
}
```

Creating CustomActivity Instances

Now that the `CustomActivity` class is fully developed, we need to have the example application create instances of this class rather than instances of the default `IlvSimpleActivity` class. This is done by creating an `IlvActivityFactory` implementation that instantiates `CustomActivity` objects, and then by telling the chart to use this factory instead of the default.

We have designed the activity factory to be a nested class of `CustomActivity` called `CustomActivity.Factory`. The factory has a single method, `createActivity`, which creates an instance of `CustomActivity` upon request by the application. Here is the code:

```
public class CustomActivity extends IlvSimpleActivity {
  ...
  public static class Factory implements IlvActivityFactory {

    public IlvActivity createActivity (IlvTimeInterval interval) {
      return new CustomActivity("CA", "Custom Activity", interval);
    }
  }
  ...
}
```

We then tell the Custom Gantt Chart example to use our new factory by overriding its `populateGanttModel` method:

```
protected void populateGanttModel(IlvGanttModel model) {
  // Change the default activity factory
  gantt.setActivityFactory(new CustomActivity.Factory());
  ...
  // Call super to create data model entities from the current factories
  super.populateGanttModel(model);
}
```

Customized Activity Rendering

The Custom Gantt Chart example represents the `CustomActivity` instances in our data model with customized rendering. We can summarize the rendering as follows:

- Parent activities are rendered in the default manner, by a light blue bar and dark blue end markers. The renderer is an instance of the class `IlvActivitySummary`.
- Leaf activities are rendered as a composite of the default `IlvActivityBar`, which displays the name of the activity, with a thin yellow activity bar whose length is proportional to the priority level of the activity (see the figure in *Customization Overview* on page 54 and Figure 4.2). We use the `IlvActivityCompositeRenderer` class to create a renderer that groups the two simpler renderers into one.

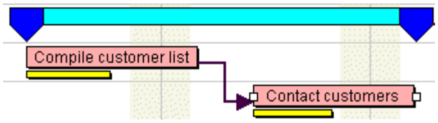


Figure 4.2 *Rendering of CustomActivity Instances*

In addition to customizing the activities, we must create both a customized renderer for the leaf activities and an activity renderer factory that will create the correct renderer upon request. We then tell the Custom Gantt Chart example to use the new activity renderer factory by calling its `setActivityRendererFactory` method. The rest of this section is divided as follows:

- *The Custom Activity Renderer Factory* discusses the renderer factory and the way it is attached to the chart.
- *The Custom Activity Renderer Class* discusses the customized activity renderer that the factory creates for the charted leaf activities.

The Custom Activity Renderer Factory

The factory that creates activity renderers for the Custom Gantt Chart example 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 skeleton of the factory is given here:

```
public class CustomGanttExample extends GanttExample {
  ...
  class CustomActivityRendererFactory
    extends IlvDefaultActivityRendererFactory {

    // The custom renderer for leaf activities
    IlvActivityCompositeRenderer renderer;

    public CustomActivityRendererFactory () {
      renderer = new CustomActivityRenderer();
    }

    // This is the main factory method which creates renderers for the chart.
    public IlvActivityRenderer createActivityRenderer(IlvActivity activity) {
      // Return the default renderer for parent activities
      if (activity.getChildCount() > 0)
        return super.createActivityRenderer(act);
      // Return the customized renderer for leaf activities
      return renderer;
    }
  }
}
```

As you can see from the bold lines in the code sample, the main factory method, `createActivityRenderer`, returns the renderer of its superclass for parent activities. This will be the default `IlvActivitySummary` renderer that displays the blue bar with end markers. For leaf activities, `createActivityRenderer` returns a custom renderer that is initialized in the factory constructor. The next section, *The Custom Activity Renderer Class*, goes into the details of how to create this renderer.

We then tell the Custom Gantt Chart example to use our new factory before any activities are initially rendered. This is done in the `populateGanttModel` method because we know that the chart has been created, but the data model has not been populated with activities yet:

```
protected void populateGanttModel(IlvGanttModel model) {
  ...
  // Change the default activity renderer factory.
  gantt.setActivityRendererFactory(new CustomActivityRendererFactory());
  // Call super to create data model entities which will be rendered
  // according to the current factories
  super.populateGanttModel(model);
}
```

The Custom Activity Renderer Class

We start creating our custom renderer by extending the class `IlvActivityCompositeRenderer`. This allows us to create a renderer as a grouping of simpler renderers. Our custom renderer implementation is located in the file:

`<installdir>/demos/gantt/customData/CustomActivityRenderer.java`

We add the 2 bars in the constructor of the `CustomActivityRenderer` class. A standard `IlvActivityBar` renderer is added to the composite to create the upper bar. By default, this displays the name of the activity on a pink background. The only customization is to add a bottom margin in order to make room for the lower yellow bar.

The lower bar is a `PriorityBar` object, which is a subclass of `IlvActivityBar`. Its background color is set to yellow and a top margin is added, which is the space occupied by the upper bar. The skeleton of the `CustomActivityRenderer` class looks like this:

```
public class CustomActivityRenderer extends IlvActivityCompositeRenderer
{
  public CustomActivityRenderer()
  {
    IlvActivityBar mainBar = new IlvActivityBar();
    mainBar.setBottomMargin(0.3f);
    addRenderer(mainBar);

    IlvActivityBar priorityBar = new PriorityBar();
    priorityBar.setTopMargin(0.75f);
    priorityBar.setBackground(Color.yellow);
    addRenderer(priorityBar);
  }
```

When any type of graphic is selected in the Gantt sheet, an instance of `IlvSelection` is created to visually represent the selected state. When an `IlvActivityGraphicObject` is selected, it asks its renderer to create the `IlvSelection` instance by calling the `IlvActivityRenderer.makeSelection` method. All the `IlvActivityRenderer` implementations provided in the Gantt module return an instance of `IlvActivityGraphicSelection`, which extends from `IlvSelection`. The `IlvActivityGraphicSelection` class visually represents the selected state of an activity by displaying 2 squares at either end of the activity graphic, centered in the vertical direction.

In the case of our `CustomActivityRenderer`, the standard `IlvActivityGraphicSelection` would display the 2 squares centered vertically in the blank space between the two horizontal bars. In order to display the selection squares centered vertically along the upper bar, we override the `makeSelection` method of the `CustomActivityRenderer`:

```
  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;
      }
    };
  }
```

The `PriorityBar` class is a subclass of `IlvActivityBar` and is defined as an inner class of `CustomActivityRenderer`. We set its displayed property to `null` so that no text is rendered:

```
class PriorityBar extends IlvActivityBar
{
  public PriorityBar()
  {
    setDisplayedProperty(null);
  }
}
```

For the length of the `PriorityBar` to be rendered proportionally to the priority value of the `CustomActivity` object, we override the `getBounds` method. We also override the `isRedrawNeeded` method so that the activity is redrawn automatically whenever its priority value changes. These 2 methods look like this:

```
  public IlvRect getBounds(IlvActivityGraphic ag, IlvTransformer t)
  {
    IlvRect bounds = super.getBounds(ag, t);
    CustomActivity activity = (CustomActivity)ag.getActivity();
    bounds.width = (1.0f * bounds.width * activity.getPriority()) /
CustomActivity.LOWEST_PRIORITY;
    return bounds;
  }

  public boolean isRedrawNeeded(ActivityEvent evt)
  {
    return super.isRedrawNeeded(evt) ||
      (evt instanceof PriorityEvent && ((PriorityEvent)evt).isChangedEvent());
  }
```

Finally, we override the `getToolTipText` method so that when the user pauses the mouse over the priority bar, the priority value of the `CustomActivity` will be displayed:

```
public String getToolTipText(IlvActivityGraphic ag, IlvPoint p, IlvTransformer
t)
 {
    CustomActivity activity = (CustomActivity)ag.getActivity();
    return "Priority " + activity.getPriority();
  }
priobar.setDisplayedProperty(null);
priobar.setBackground(Color.yellow);
renderer.addRenderer(priobar);
```

Customized Table Columns

In this section, we show you how to customize an existing column in the table portion of the Gantt or Schedule chart and also how to define a new type of column and add it to the table. We start by showing how to customize the tree icons in the `Name` column. Then we show how to create the `Pri` column, which displays the priority value of each custom activity.

Tree Column Icons

The first column in the table is an instance of the `IlvTreeColumn` class. This type of column uses a render 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 we 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. Our custom table-column implementation is located in the file:

`<installdir>/demos/gantt/customData/PriorityColumn.java`

This class is an extension of `IlvAbstractJTableColumn` and thereby implements the `IlvJTableColumn` interface. We start out by duplicating each of the superclass constructors and adding an `init` method, which is where we 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);
    init();
  }

  public PriorityColumn (Object headerValue, int width) {
    super(headerValue, width);
    init();
  }

  private void init () {
    ...
  }

}
```

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.

Our 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 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 we want instead, we have to create a `TableCellRenderer` and `TableCellEditor` object for the column explicitly.

Support Methods

Before we review the column renderer and editor, some `IlvJTableColumn` methods must be implemented in the class `PriorityColumn`:

- The method `getValue` returns the priority for an activity. Because priorities are stored as primitive `int` values, we wrap the priority in an `Integer` object.
- The method `setValue` sets the priority for an activity. The priority will be passed in as an `Integer` because of the way we coded `getValue`.
- The method `isEditable` is overridden to return `true` so that the priority values can be edited.

These three methods look like this:

```
public Object getValue (IlvHierarchyNode activity) {
  if (activity instanceof CustomActivity)
    return new Integer(((CustomActivity)activity).getPriority());
  else
    return null;
}

public void setValue (IlvHierarchyNode activity, Object value) {
  // Should never happen, but we'll check anyway
  if (!(activity instanceof CustomActivity && value instanceof Integer))
    return;
  ((CustomActivity)activity).setPriority(((Integer)value).intValue());
}

public boolean isEditable (IlvHierarchyNode activity) {
  return activity instanceof CustomActivity;
}
```

The Cell Renderer

The renderer of the priority column is created in the `createRenderer` method as an extension of the Swing `DefaultTableCellRenderer` class. Only two methods are overridden:

- The method `setValue` is overridden to use a `NumberFormat` to format the priority value into a text string for display.
- The method `getTableCellRendererComponent` is overridden 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:

```
private NumberFormat formatter = NumberFormat.getInstance();

private 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;
}
```

Now that we can create a customized renderer for the column, we tell the underlying `TableColumn` object to use it in the `init` method:

```
private void init () {
  TableColumn swingColumn = getTableColumn();
  swingColumn.setCellRenderer(createRenderer());
  ...
}
```

The last part we must define with regard to rendering the priority values is to have the `PriorityColumn` automatically refresh any activities whose value has changed. This change will be signalled from the Gantt data model by the `PriorityEvent` that we created in section *Adding Property Events* on page 57. When the column is added to the table, its setGanttConfiguration method will be called. We use this opportunity to register the column to receive the desired PriorityEvent notifications. To act as an event listener, the column must implement the GenericEventListener interface and its inform method. In the inform method, the column calls the cellUpdated method of its superclass to refresh the activity whose priority has changed. The relevant code looks like this:

```
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 PriorityEvent's.
    if (this.ganttConfig != null)
      this.ganttConfig.addListener(this, PriorityEvent.class);
  }

  // GenericEventListener implementation
  public void inform (java.util.EventObject event) {
    // We only register for PriorityEvent's, but be safe anyway.
    if (!(event instanceof PriorityEvent))
      return;
    PriorityEvent pEvent = (PriorityEvent) event;
    // Make sure that the event is not a pre about-to-change event.
    // It would do no harm, but would be an unnecessary repaint.
    if (pEvent.isChangedEvent())
      cellUpdated((IlvHierarchyNode)event.getSource());
  }

}
```

The Cell Editor

To be able to use a Swing JSlider as the editor for the priority column, we have created 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 simple implementations of the TableCellEditor interface.

We then create the editor in the createEditor method of the class PriorityColumn and tell the underlying TableColumn object to use it in the init method:

```
private TableCellEditor createEditor () {
  return new SliderEditor();
}

private void init () {
  TableColumn swingColumn = getTableColumn();
  ...
  swingColumn.setCellEditor(createEditor());
}
```

Adding the Column to the Table

Now that we have designed the PriorityColumn class, we need to add an instance of it to the table. This is done by adding the column to the IlvJTable instance of the chart, which can be obtained 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"));
}
```

CHAPTER

5

Load-on-Demand

The Gantt module 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 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.

This chapter covers the following topics:

- *Vertical Load-On-Demand* and the Database Gantt example.
- *Horizontal Load-On-Demand* and the Database Schedule example.

Vertical Load-On-Demand

Vertical load-on-demand refers to the ability to defer loading of row-oriented information until it is needed for display. This means activity data for a Gantt chart and resource data for a Schedule chart. Vertical load-on-demand is facilitated by the design of the Gantt module 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 creating a load-on-demand data model implementation. 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

The source code file of the Database Gantt example application is named `DBGanttExample.java` and can be found in the directory:

```
<installdir>/demos/gantt/database
```

To run the example, ensure that the Ant utility is properly configured. If not, see the instructions on how to configure Ant for ILOG JViews in:

```
<installdir>/html/installation.html
```

Then, go to the directory where the example is installed and type:

```
ant rungantt
```

to run the example as an application.

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 key class relationships are shown in Figure 5.1:

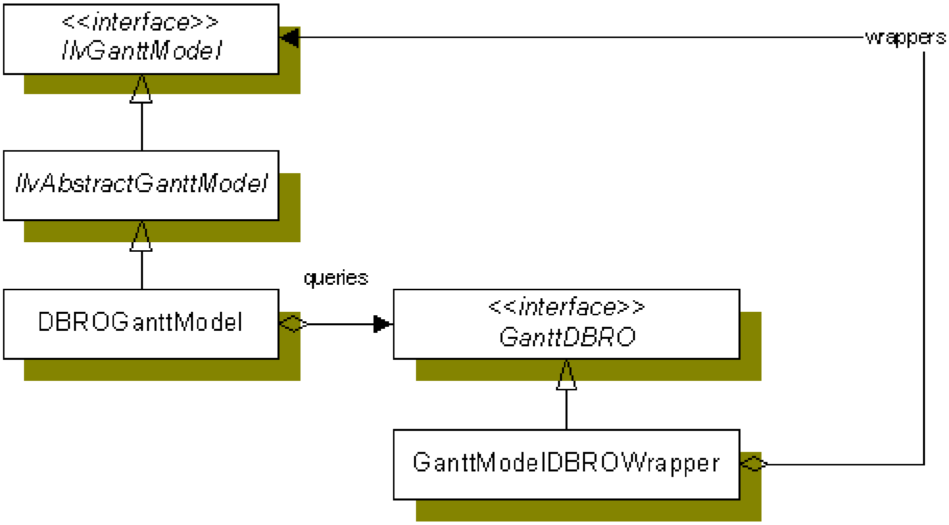


Figure 5.1 The Class Relationships of the Database Gantt Example

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 4 inner interfaces that define the record structure of the activity, resource, constraint, and reservation data:

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 it is not required. The `GanttDBRO` interface defines the following database query methods:

Activities	`String queryRootActivityKey()`
	`ActivityRecord queryActivity(String key)`
	`String queryActivityParent(String key)`
	`String[] queryActivityChildren(String key)`

Resources	`String queryRootResourceKey()`
	`ActivityRecord queryResource(String key)`
	`String queryResourceParent(String key)`
	`String[] queryResourceChildren(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` is created from the data. This data model is then wrapped by an instance of `GanttModelDBROWrapper` that is bound to a `DBROGanttModel` instance. The object relationships look like this:

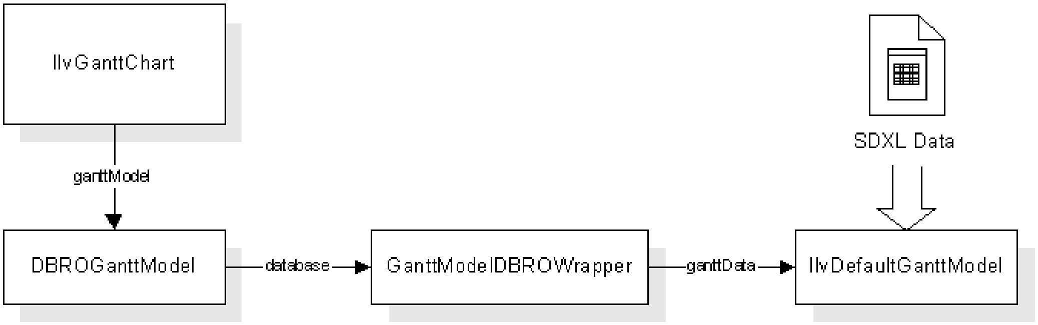


Figure 5.2 Object Relationships of the Database Gantt Example

When an activity row is first displayed in the Gantt chart, the chart calls the `queryActivityChildren()` method 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's 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 4 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

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 Schedule Example

The source code file of the Database Schedule example application is named `DBScheduleExample.java` and can be found in the directory:

```
<installdir>/demos/gantt/database
```

To run the example, ensure that the Ant utility is properly configured. If not, see the instructions on how to configure Ant for ILOG JViews in:

```
<installdir>/html/installation.html
```

Then, go to the directory where the example is installed and type:

```
ant runschedule
```

5. Load-on-Demand

to run the example as an application.

Understanding the Database Schedule Example

The Database Schedule example application visualizes the same relational database of scheduling data as the Database Gantt example (see Figure 5.2). 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)
```

```
float getReservationCacheLoadThreshold()
```

```
void setReservationCacheLoadThreshold(float loadThreshold)
```

```
float getReservationCacheLoadFactor()
```

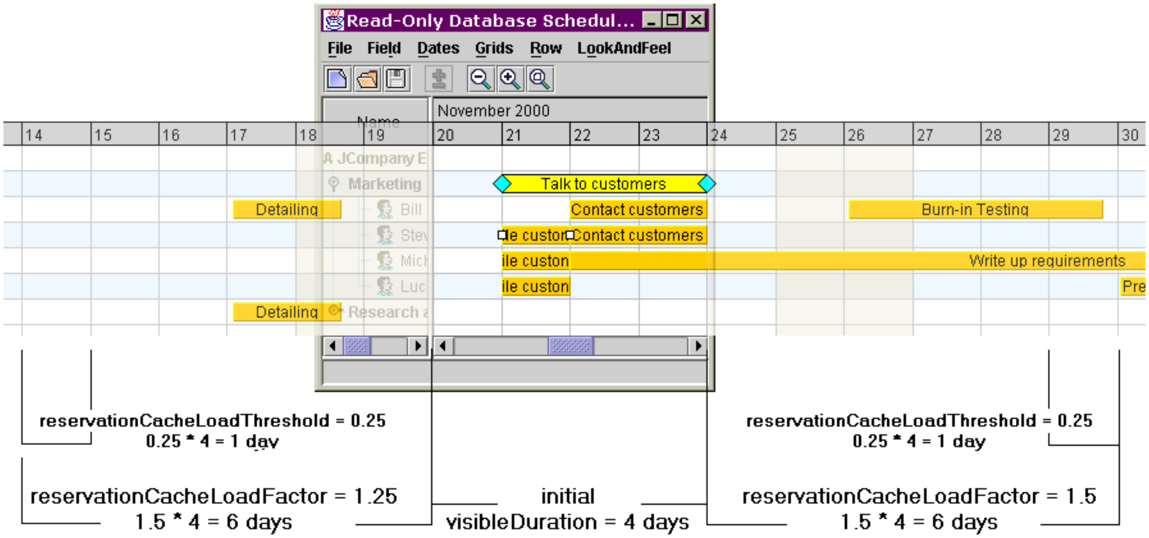
```
void setReservationCacheLoadFactor(float loadFactor)
```

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 relationship between these settings is best understood by referring to the following figure:



The Schedule chart is initially displayed with a visible duration of 4 days, from November 20 through November 23. The chart's `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 `IlvGanttModel.reservationIterator (IlvResource resource, IlvTimeInterval interval)` method. 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's `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.

CHAPTER

6

Schedule Data Serialization and Exchange with SDXL

The ILOG JViews Gantt package allows users of the Gantt to serialize schedule data to Schedule Data Exchange Language (SDXL) files. This is accomplished using the classes in the `ilog.views.gantt.xml` package.

This chapter describes SDXL and how to use this language to serialize schedule data. The chapter contains the following sections:

- *Schedule Data Exchange Language Overview*. This section gives an overview of the language. It presents the design criteria of the language as well as some scenarios on how it can be used.
- *Serializing Schedule Data*. This section describes how to serialize (write and read) `IlvGanttModel` objects by using the `ilog.views.gantt.xml` package.
- *Customization of SDXL*. This section shows how to extend SDXL to meet the needs of the users. It also shows how to customize the default readers and writers provided by the `ilog.views.gantt.xml` package.
- *Schedule Data Exchange Language Specification*. This section defines SDXL in detail.

Schedule Data Exchange Language Overview

The Gantt Chart (`IlvGanttChart`) and the Schedule Chart (`IlvScheduleChart`) are designed to visualize and to edit schedule data in a 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 ILOG JViews. 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, we can imagine the following scenarios:

- You use an ILOG 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 into SDXL files and distribute them by means of a lightweight ILOG JViews Gantt program that does not need database connections.
- You use an ILOG JViews Gantt program to manage your schedules. You can save your schedules into 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 ILOG JViews Gantt program in order to visualize them.

Package For Reading and Writing SDXL

The package named `ilog.views.gantt.xml` contains all the classes allowing you to serialize schedule data to SDXL files:

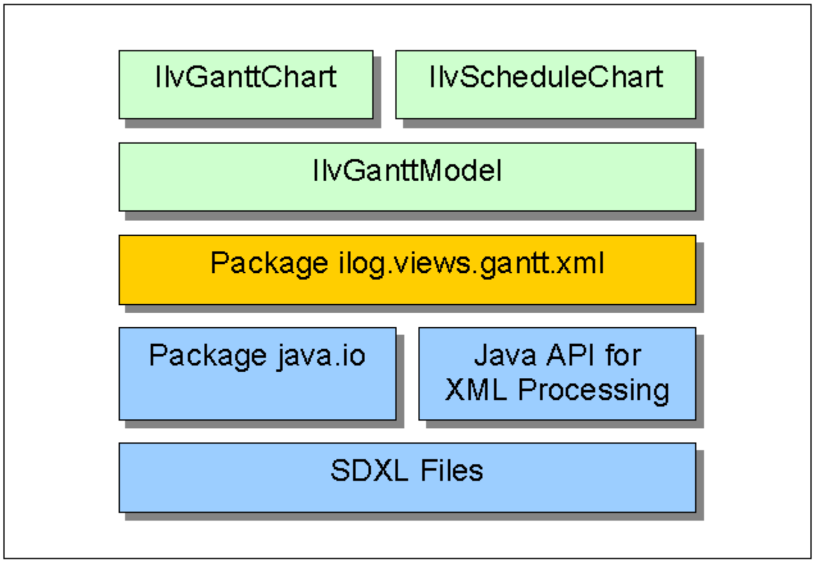


Figure 6.1 *Package ilog.views.gantt.xml and Related Packages*

Figure 6.1 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.

In order to use the `ilog.views.gantt.xml` package, you need not only the jar files for the ILOG JViews Gantt package but also the following extra jar files:

- `jaxp.jar`
- `crimson.jar`

You can find these files in the `<installdir>/classes` directory.

These two jar files are parts of the JAVA API for XML Processing (JAXP) Optional Package that provides basic features for reading, manipulating, and generating XML documents through pure Java APIs. It is a thin and lightweight API that provides a standard way to seamlessly integrate any XML-compliant parser with a Java application. See `http://java.sun.com/xml/download.html`, for more information.

SDXL Example

You can examine the following file for an example of SDXL:

```
<installdir>/data/gantt/sdxl.sdxl
```

Serializing Schedule Data

The following sections show how to use the main classes of the `ilog.views.gantt.xml` package. You will learn first how to write an `IlvGanttModel` to an SDXL file and then how to read back an `IlvGanttModel` from an SDXL file.

You can find examples on how to use the `ilog.views.gantt.xml` package in the `<installdir>/demos/gantt/xml` directory. The two examples that show how to serialize schedule data are `XMLGanttExample.java` and `XMLScheduleExample.java`. See `<installdir>/demos/gantt/xml/index.html` for more information.

Writing an IlvGanttModel to an SDXL File

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

1. Use JAXP to create an instance of `org.w3c.dom.Document`.
2. Use the `ilog.views.gantt.xml` package to create an instance of `IlvGanttDocumentWriter`.
3. Use the `IlvGanttDocumentWriter` to write your `IlvGanttModel` to the document you created in Step 1.
4. Use the `java.io` package to create a `java.io.OutputStream` object for the SDXL file you want to write to.
5. Use the `ilog.views.gantt.xml` package to create an instance of `IlvGanttStreamWriter`.
6. Use the stream writer to write the document to the output stream you created in Step 4.

These 6 steps are now described in more detail:

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.

First you need to import the packages:

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

Get an instance of the `DocumentBuilderFactory`:

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

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

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

Use the `DocumentBuilder` to create the document object:

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

See the `<installdir>/demos/gantt/xml/XMLGanttActions.java` file for a concrete implementation.

> ***Note:*** *This section presented one way of creating an* `org.w3c.dom.Document`*. You may, of course, use other ways to create document objects.*

Create an IlvGanttDocumentWriter

The `ilog.views.gantt.xml` package of ILOG JViews provides the `IlvGanttDocumentWriter` class. This section shows you how to create an instance of this class.

First import the package:

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

Then, create the `IlvGanttDocumentWriter`:

```
IlvGanttDocumentWriter documentWriter = new IlvGanttDocumentWriter();
```

The document writer is ready to write an `IlvGanttModel` to a document.

Writing a Gantt Model to a Document

The `IlvGanttDocumentWriter` has a method called `writeGanttModel`. You can call this method to write your Gantt model:

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

The `document` argument is the document object you created in the previous step. The `yourGanttModel` argument is the `IlvGanttModel` you want to write to the document.

Creating an OutputStream

You can use the `java.io` package to create an `OutputStream`. First import the `java.io` package:

```
import java.io
```

Then, create an `OutputStream` for the file you want to write to:

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

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

> ***Note:*** *This section presented one way of creating an* `OutputStream` *but you may, of course, use other ways.*

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.

Import the ilog.views.gantt.xml package:

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

Then, create the stream writer:

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

The stream writer is now ready to write a document object to an OutputStream.

Writing a Document to an OutputStream

Now that you have the OutputStream and the document, call the writeDocument method of the stream writer created in the previous step to write the document to the OutputStream.

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

Then, close the output stream:

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

Your IlvGanttModel is now written to an SDXL file.

How to Read an IlvGanttModel from an SDXL File

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

1. Use JAXP to create an InputSource for the file you want to read.
2. Use JAXP to parse the InputSource in order to get an org.w3c.dom.Document.
3. Use the ilog.views.gantt.xml package to create an instance of IlvGanttDocumentReader.
4. 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.

These steps are now seen in more detail.

Creating an InputSource

You can use JAXP to create an InputSource for the SDXL file you want to read.

First you need to 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”;
```

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 a JAXP parser.

> ***Note:*** *This section presented one way of creating an* `InputSource` *but you may, of course, use other ways.*

Parsing an InputSource

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

First import the packages:

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

Get an instance of the `DocumentBuilderFactory`:

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

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

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

Use the `DocumentBuilder` to parse the input source:

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

You now need an `IlvGanttDocumentReader` to read the document.

Creating an IlvGanttDocumentReader

The `ilog.views.gantt.xml` package of ILOG JViews provides the `IlvGanttDocumentReader` class. This section shows you how to create an instance of this class.

First import the package:

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

Then, create the `IlvGanttDocumentReader`:

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

The document reader is ready to read a document.

Reading a Gantt Model from a Document

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

First import the ilog.views.gantt and ilog.views.gantt.model packages:

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

Then, create a default Gantt model:

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

Read the Gantt 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. Here is an example of how to handle such exceptions:

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

Customization of SDXL

The ilog.views.gantt.xml package provides the basic readers and writers to serialize schedule data. If you customize the Gantt model, you might need to customize the ilog.views.gantt.xml package in order to serialize the customized Gantt model. This section shows you how to customize the default readers and writers provided by the ilog.views.gantt.xml package.

Overview of ilog.views.gantt.xml

The ilog.gantt.views.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 model is exclusively defined by these classes, the `ilog.views.gantt.xml` package contains all the classes you need to serialize your schedule data.

Table 6.1 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 2 default implementations. The “simple” implementations read and write the default data model classes, and the “general” implementations read and write the data model classes that support user-defined properties. These can be used as-is.

Table 6.1 *Default Readers and Writers in ilog.views.gantt.xml*

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 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 of these 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 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` from a document. The document is the unique input of the package. To read a Gantt model the user must provide a document object. (See section *How to Read an IlvGanttModel from an SDXL File* on page 86 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` to an `OutputStream`. The `OutputStream` is the unique output point of the package. Users who want to write an SDXL file should provide an `OutputStream`. (See section *Writing an IlvGanttModel to an SDXL File* on page 84 for information on how to create an `OutputStream` for an SDXL file by using the `java.io` package.

Customizing Readers and Writers

The default readers and writers provided by the JViews Gantt module are enough to serialize default Gantt models. If you created a customized Gantt model, such as the `<installdir>/demos/gantt/customData` example, 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. Now we will see how to customize the `IlvSimpleActivityReader` and `IlvSimpleActivityWriter` by subclassing them.

In the `<installdir>/demos/gantt/customData` example, we have created a customized activity called `CustomActivity` that is a subclass of `IlvSimpleActivity`. `CustomActivity` has one more property than `IlvSimpleActivity`. This property is the priority. Now we will see how to make a reader and a writer for the `CustomActivity`:

We create a writer for the `CustomActivity` by creating a subclass of `IlvSimpleActivityWriter`:

```
/**
 * This class extends an <code>IlvSimpleActivityWriter</code>.
 * The method <code>writeActivity()</code> is overridden to
 * write a <code>CustomActivity</code> instead of an
 * <code>IlvSimpleActivity</code>.
 */
public class CustomActivityWriter
```

```
  extends IlvSimpleActivityWriter
{
  /**
   * Overrides the method <code>writeActivity()</code>
   * to write a <code>CustomActivity</code> instead of an
   * <code>IlvSimpleActivity</code>.
   * @param elem The element to write to.
   * @param activity The activity to write.
   */
  public void writeActivity(Element elem, IlvActivity activity,
                          IlvGanttDocumentWriter.Context writeContext)
    throws Exception
  {
    if(!(activity instanceof CustomActivity))
      throw new Exception("A CustomActivity expected.");

    super.writeActivity(elem, activity);
    int priority = ((CustomActivity)activity).getPriority();
    elem.setAttribute("priority", Integer.toString(priority));
  }
}
```

This writer is capable of writing a `CustomActivity` to a customized SDXL file:

```
<activity id="A-3.1.2" name="Detailing" start="11-10-2000 6:21:33"
                       end="14-10-2000 6:21:33" priority="5" />
```

You should notice that the `activity` element has an extra property named `priority`. You have to define a new DTD for the customized SDXL file.

To read back the customized activity, we create a reader for the `CustomActivity` by creating a subclass of `IlvSimpleActivityReader`:

```
/**
 * This class extends an <code>IlvSimpleActivityReder</code>.
 * The method <code>readActivity()</code> is overridden to
 * read a <code>CustomActivity</code> instead of an
 * <code>IlvSimpleActivity</code>.
 */
public class CustomActivityReader
  extends IlvSimpleActivityReader
{
  /**
   * Overrides this method to read a <code>CustomActivity</code>
   * instead of an <code>IlvSimpleActivity</code>.
   * @param elem The element to read from.
   * @return The new activity.
   */
  public IlvActivity readActivity(Element elem,
                                  IlvGanttDocumentReader.Context readContext)
    throws IlvGanttReaderException
  {
    CustomActivity activity = (CustomActivity)super.readActivity(elem);
    String priority = elem.getAttribute("priority");
    if(priority==null)
      throw new IlvGanttReaderException(elem,
        "Attribute \"priority\" not found");

    activity.setPriority(Integer.valueOf(priority).intValue());

    return activity;
  }

  /**
   * Overrides to create a <code>CustomActivity</code>
   * instead of an <code>IlvSimpleActivity</code>.
   */
  protected IlvActivity createActivity(String id,
                                       String name,
                                       Date start,
                                       Date end
                                       )
  {
    return new CustomActivity(id, name, start, end);
  }
}
```

For more details on how to use these customized readers and writers, refer to the following files in `<installdir>/demos/gantt/xml`:

- `CustomActivityReader.java`
- `CustomActivityWriter.java`
- `XMLCustomGanttExample.java`

See `<installdir>/demos/gantt/xml/index.html` for more details.

Schedule Data Exchange Language Specification

SDXL is an application of the W3C XML language. It is designed to serialize all the contents of a given `IlvGanttModel`. It describes scheduling data in terms of the following elements:

- *Activity Elements*
- *Resource Elements*
- *Constraint Elements*
- *Reservation Elements*
- *Schedule Element*
- *Property Element*

See also Appendix A, *Document Type Definition for SDXL* for the Document Type Definition of the SDXL language.

Activity Elements

There are 3 types of `activity` elements:

- *Activity Element* - for simple activities.
- *Activity Group Element*- for parent activities.
- *Activities Element*- for the root of the activities.

Activity Element

```
<!ELEMENT activity (activity|property)*>
<!ATTLIST activity
    id    ID         #REQUIRED
    name  %Text;     #REQUIRED
    start %Datetime; #REQUIRED
    end   %Datetime; #REQUIRED >
```

Example:

```
<activity
    id="A1"
    name="Gather Requirements"
    start="09/12/1999 10:15:50"
    end="24/12/1999 10:15:50"
/>
```

The `activity` element is used to serialize an `IlvActivity` object. An `activity` element is defined by 4 attributes:

- `id` - The ID of a given activity must be unique within the set of all activities and resources in the schedule.
- `name` - The name of the activity.
- `start` - The start time of the activity.
- `end` - The end time of the activity.

and can have any number of `property` elements.

Activity Group Element

An activity group is defined in the same way as a simple activity except that it contains nested simple activity elements. Use this element to serialize parent activities:

```
<activity id="a1" name="name1" start=”1/1/1999” end=”1/9/99”>
   <activity id="a2" name="n2" start=”1/1/99” end=”1/3/99”/>
   <activity id="a3" name="n3" start=”1/3/99” end=”1/5/99”/>
   <activity id="a4" name="n4" start=”1/5/99” end=”1/9/99”/>
</activity>
```

Activities Element

```
<!ELEMENT activities (activity)+>
<!ATTLIST activities
     dateFormat %Text; #IMPLIED >
```

An `activities` element groups all activity elements in a `schedule` element. In other words, there is only one `activities` element in a `schedule` element.

```
<activities dateFormat=”d/M/yy”>
  <activity id="a1" name="n1" start=”1/1/99” end=”1/9/99”>
    <activity id="a2" name="n2" start=”1/1/99” end=”1/3/99”/>
    <activity id="a3" name="n3" start=”1/3/99” end=”1/5/99”/>
  </activity>
  <activity id="a4" name="n4" start=”1/5/99” end=”1/9/99”/>
</activities>
```

The `dateformat` attribute defines the date format used for activities.

Resource Elements

There are 3 types of resource elements:

- *Resource Element* - for simple resources.
- *Resource Group Element* - for parent resources.
- *Resources Element* - for the root of the resources.

Resource Element

```
<!ELEMENT resource (resource|property)*>
<!ATTLIST resource
    id       ID     #REQUIRED
    name     %Text; #REQUIRED
    quantity %Text; #IMPLIED >
```

Example:

```
<resource
     id="MS"
     name="Michael Smith"
     quantity="1.0"
/>
```

The `resource` element is used to serialize `IlvResource` objects. It is defined by 3 attributes:

- `id` - The ID of a given resource must be unique within the set of all activities and resources in the schedule.
- `name` - The name of the resource.
- `quantity` - The quantity of the resource.

and can have any number of `property` elements.

Resource Group Element

A resource group is defined in the same way as a simple resource except that it contains simple resource elements:

```
<resource id="MKT" name="Marketing" quantity="1.0" >
    <resource id="BM" name="Bill McDonald" quantity="1.0" />
    <resource id="SJ" name="Steve Knoll" quantity="1.0" />
    <resource id="MD" name="Michael Smith" quantity="1.0" />
    <resource id="LG" name="Luc Dupont" quantity="1.0" />
</resource>
```

Resources Element

```
<!ELEMENT resources (resource)+>
<!ATTLIST resources >
```

A `resources` element groups all resource elements in a `schedule` element. In other words, there is only one `resources` element in a `schedule` element.

```
<resources>
    <resource id="r1" name="name1" quantity="1.0" >
        <resource id="r2" name="name2" quantity="1.0" />
        <resource id="r3" name="name3" quantity="1.0" />
        <resource id="r4" name="name4" quantity="1.0" />
    </resource>
    <resource id="r5" name="name5" quantity="1.0" />
    <resource id="r6" name="name6" quantity="1.0" />
</resources>
```

Constraint Elements

- *Constraint Element* - for simple constraints.
- *Constraints Element* - for the root of the constraints.

Constraint Element

```
<!ELEMENT constraint (property)*>
<!ATTLIST constraint
    from %ActivityID;     #REQUIRED
    to   %ActivityID;     #REQUIRED
    type %ConstraintType; #REQUIRED >
```

Example:

```
<constraint
    from="activity1"
    to="activity2"
    type="endstart"
/>
```

The `constraint` element is used to serialize `IlvConstraint` objects. It is defined by 3 attributes:

- `from` - The ID of the *from* activity. The activity must be a valid activity in the same SDXL file.
- `to` - The ID of the *to* activity. The activity must be a valid activity in the same SDXL file.
- `type` - The type of the constraint. It must be one the following: “Start-Start”, “Start-End”, “End-Start”, “End-End”.

and can have any number of `property` elements.

Constraints Element

```
<!ELEMENT constraints (constraint)+>
<!ATTLIST constraints >
```

A `constraints` element groups all constraint elements in a `schedule` element. In other words, there is only one `constraints` element in a `schedule` element.

```
<constraints>
    <constraint from="a2" to="a3" type="End-Start" />
    <constraint from="a3" to="a4" type="End-Start" />
    <constraint from="a2" to="a4" type="End-Start" />
</constraints>
```

Reservation Elements

- *Reservation Element* - for simple reservations.
- *Reservations Element* - for the root of the reservations.

Reservation Element

```
<!ELEMENT reservation (property)*>
<!ATTLIST reservation
    activity %ActivityID; #REQUIRED
    resource %ResourceID; #REQUIRED >
```

Example:

```
<reservation
    activity="id-of-activity1"
    resource="id-of-resource1"
/>
```

The `reservation` element is used to describe an `IlvReservation` object. A `reservation` element is defined by 2 attributes:

- `activity` - The ID of the activity. The activity must be a valid activity in the same file.
- `resource` - The ID of the resource. The resource must be a valid resource in the same file.

and can have any number of `property` elements.

Reservations Element

```
<!ELEMENT reservations (reservation)+>
<!ATTLIST reservations >
```

A `reservations` element groups all reservation elements in a `schedule` element. In other words, there is only one `reservations` element in a `schedule` element.

```
<reservations>
    <reservation activity="a1" resource="r1" />
    <reservation activity="a2" resource="r2" />
    <reservation activity="a3" resource="r3" />
</reservations>
```

Schedule Element

```
<!ELEMENT schedule (title?, desc?, resources?, activities?,
    constraints?, reservations?) >
<!ATTLIST schedule
    version %Text; #REQUIRED >
```

Example:

```
<schedule version=”1.0”>
    <activities> … </activities>
    <resources> … </resources>
    <constraints> … </constraints>
    <reservations> … </reservations>
</schedule>
```

The `schedule` element is the root element that contains other elements defined in these document. It is designed to describe the contents of a given `IlvGanttModel`. A `schedule` element contains the following:

- An activities element.
- A resources element.
- A constraints element.
- A reservations element.

The `version` attribute indicates the version of this SDXL.

Title and Desc Elements

```
<!ELEMENT title (#PCDATA)>
<!ELEMENT desc (#PCDATA)>
```

The `title` element and the `desc` element are used to specify extra information on the `schedule` element.

Property Element

```
<!ELEMENT property (#PCDATA)>
<!ATTLIST property
    name       %Text;     #REQUIRED
    javaClass  %Text;     #IMPLIED >
```

The `property` element is used to specify user defined properties added to the `IlvGeneral*` objects of the Gantt data model. It is defined by the 2 following attributes:

- `name` - The property name, which is required.
- `javaClass` - The property class name, which is not required.

The property value is the data section of this element.

C H A P T E R

7

Styling

The Gantt module allows you to customize the appearance of a chart by applying cascading style sheets (CSS). The example in the `<installdir>/demos/gantt/css` directory illustrates how styling works and contains several style sheet samples.

This chapter describes the styling capabilities of the Gantt module and covers the following topics:

- *Introduction to Styling*
- *Styling the Gantt and Schedule Chart Components*
- *Styling the Gantt Data*

Introduction to Styling

The appearance of a Gantt or Schedule chart can be controlled with CSS. CSS are introduced in the ILOG JViews *Stylable Data Mapper (SDM) User's Manual*. The following sections introduce the use of style sheets within the Gantt module, as follows:

- *Applying Styles*
- *Disabling Styling*
- *The Gantt and Schedule CSS Examples*

Applying Styles

The `IlvHierarchyChart` class, the common superclass of `IlvGanttChart` and `IlvScheduleChart`, implements the `IlvStylable` interface. This interface defines several methods that can be used to control styling. Here is an example of 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());
}
```

Here is a list of the `IlvHierarchyChart` methods that can be used to control styling:

Table 7.1 *Methods for Controlling Styling*

Where Used	Methods

Style Sheets	`String getStyleSheet()`
	`void setStyleSheet(String)`
	`String getStyleSheets(int)`
	`void setStyleSheets(int, String)`
	`String[] getStyleSheets()`
	`void setStyleSheets(String[])`

Debugging	`int getStyleSheetDebugMask`
	`void 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 or Schedule 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

You can globally disable styling by passing `null` to the `setStyleSheets(String[])` method of `IlvHierarchyChart`. This tells the chart 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.

The Gantt and Schedule CSS Examples

The Gantt and Schedule CSS examples are provided with the Gantt module to show how you can use CSS to customize the appearance of your charts.

Running the CSS Examples

The files and source code of the Gantt and Schedule CSS examples can be found in the directory:

```
<installdir>/demos/gantt/css
```

To run the examples, ensure that the `Ant` utility is properly configured. If not, see the instructions on how to configure `Ant` for ILOG JViews in:

```
<installdir>/html/installation.html
```

Then, go to the directory where the example is installed and type:

```
ant rungantt
```

to run the Gantt CSS example as an application

or type:

```
ant runschedule
```

to run the Schedule CSS example as an application.

Scheduling Data for the CSS Examples

The Gantt and Schedule CSS examples display scheduling data that is initially loaded from the XML file:

```
<installdir>/demos/gantt/css/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. For example, here is 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 examples 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 you use the *general* implementations that are provided in the `ilog.views.gantt.model.general` package. To read the XML file and create the data model objects, you need to customize an `IlvGanttDocumentReader` like this:

```
IlvGanttModel model = new IlvDefaultGanttModel();
IlvGanttDocumentReader docReader = new IlvGanttDocumentReader();
docReader.setActivityReader(new IlvGeneralActivityReader());
docReader.setConstraintReader(new IlvGeneralConstraintReader());
docReader.setReservationReader(new IlvGeneralReservationReader());
docReader.setResourceReader(new IlvGeneralResourceReader());
Document document = ...use JAXP to read XML file into document...
docReader.readGanttModel(document, model);
```

> ***Note:*** *For more details on how to read an* `IlvGanttModel` *from an XML data file, see Chapter 6, Schedule Data Serialization and Exchange with SDXL.*

A First Style Sheet

This section describes the contents of a simple style sheet and how it customizes a Gantt chart. You can follow and test this by performing the following steps:

1. Create an empty CSS file in the `data` directory of the Gantt and Schedule CSS examples. The file must have a `.css` extension:

   `<installdir>/demos/gantt/css/data/gantt/my-first-stylesheet.css`

2. Run the Gantt CSS example (see *Running the CSS Examples* on page 101) and your new style sheet will appear in the list of available style sheets. The Gantt CSS example makes available all the style sheets in the `data/gantt` directory. Similarly, the Schedule CSS example makes available all the style sheets it finds in the `data/schedule` directory.

3. While the Gantt CSS example 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 Gantt CSS example and reapplying the style sheet to the chart. Reselect your style sheet from the list of available style sheets.

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.

Next, 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;
}
```

Figure 7.1 shows how the Gantt chart looks with this style sheet.

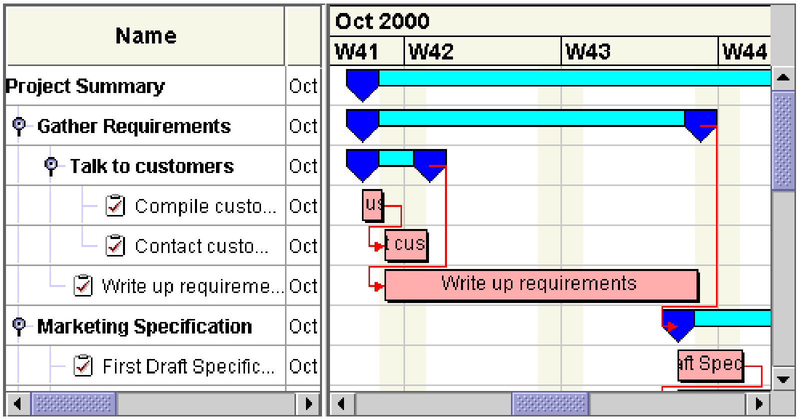
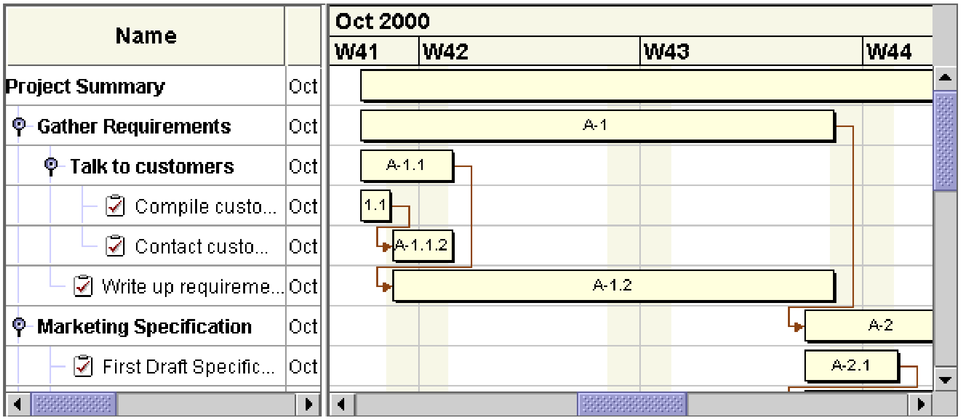


Figure 7.1 *Gantt Chart After Application of the Initial Style Sheet*

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;
}

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

Figure 7.2 shows how the Gantt chart looks now with the completed style sheet.



Name	
Project Summary	Oct
Gather Requirements	Oct
Talk to customers	Oct
Compile custo...	Oct
Contact custo...	Oct
Write up requireme...	Oct
Marketing Specification	Oct
First Draft Specific...	Oct

Figure 7.2 Gantt Chart After Application of the Completed Style Sheet

Two Kinds of Rules

From this example of a style sheet, 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, such as the table, the time scale, and the Gantt sheet. These rules are described in section *Styling the Gantt and Schedule Chart Components*.
- Rules that control how Gantt data model entities, such as activities, constraints, and reservations, are rendered in the Gantt sheet. These rules are described in *Styling the Gantt Data*.

Styling the Gantt and Schedule Chart Components

This section describes how style sheets can be used to customize the appearance of the Gantt and Schedule chart components and their sub-elements. The following table lists the CSS elements that are defined to reference the different parts of the chart components:

Table 7.2 *The Gantt and Schedule Chart CSS Elements*

CSS Element Type and ID	Description	Target Object Class	Bean Properties	Type

chart	The Gantt or Schedule chart component	`IlvGanttChart`
		`IlvScheduleChart`
			`constraintLayerVisible`
			`displayingConstraints`
			`dividerBorder`
			`dividerLocation`
			`dividerOpaqueMove`
			`dividerSize`
			`ganttSheetBackground`
			`ganttSheetToolTipsEnabled`
			`ganttSheetVisible`
			`horizontalScrollBarVisible`
			`maxVisibleTime`
			`minVisibleTime`
			`rowHeight`
			`tableBackground`
			`tableFont`
			`tableForeground`
			`tableGridColor`
			`tableHeaderBackground`
			`tableHeaderFont`
			`tableHeaderForeground`
			`tableVisible`
			`timeScaleBackground`
			`timeScaleFont`
			`timeScaleForeground`
			`verticalPosition`
			`visibleDuration`
			`visibleIntervalAnimationSteps`
			`visibleTime`
				`boolean`
				`boolean`
				`Border`
				`int`
				`boolean`
				`int`
				`Color`
				`boolean`
				`boolean`
				`boolean`
				`Date`
				`Date`
				`int`
				`Color`
				`Font`
				`Color`
				`Color`
				`Color`
				`Font`
				`Color`
				`boolean`
				`Color`
				`Font`
				`Color`
				`int`
				`IlvDuration`
				`int`
				`Date`

		`IlvScheduleChart`	`activityLayout`
			`reservationCacheLoadFactor`
			`reservationCacheLoadThreshold`
			`reservationCachingEnabled`
				`enum`
				`float`
				`float`
				`boolean`

Table 7.2 *The Gantt and Schedule Chart CSS Elements*

CSS Element Type and ID	Description	Target Object Class	Bean Properties	Type

sheet	The Gantt sheet	`IlvGanttSheet`	`antialiasing`
			`background`
			`backgroundPatternLocation`
			`defaultGhostColor`
			`defaultXORColor`
			`horizontalGrid`
			`verticalGrid`
				`boolean`
				`Color`
				`URL`
				`Color`
				`Color`
				`IlvGanttGridRenderer`
				`IlvGanttGridRenderer`

horizontalGrid	The horizontal grid of the Gantt sheet	`IlvHorizontalGanttGrid`	`evenRowsBackground`
			`filled`
			`foreground`
			`oddRowsBackground`
				`Color`
				`boolean`
				`Color`
				`Color`

verticalGrid	The vertical grid of the Gantt sheet	`IlvVerticalGanttGrid`	`foreground`	`Color`

timeScale	The time scale	`IlvTimeScale`	`background`
			`font`
			`foreground`
				`Color`
				`Font`
				`Color`

table	The Gantt table	`IlvJTable`	`background`
			`columnMargin`
			`columns`
			`font`
			`foreground`
			`gridColor`
			`headerBackground`
			`headerFont`
			`headerForeground`
			`showsRootHandles`
				`Color`
				`int`
				`String`
				`Font`
				`Color`
				`Color`
				`Color`
				`Font`
				`Color`
				`boolean`

> ***Note:*** *Many of the Gantt examples provided in the distribution, including the Gantt and Schedule CSS Examples, override the default* `IlvVerticalGanttGrid` *with an instance of* `WeekendGrid`*. This class is provided in the* `demos/gantt/shared/src/shared` *directory. You can review the source code to determine the additional Bean properties that this class provides.*

These elements can be used to modify the Bean properties of the corresponding target object. For example, here is 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 element type for each of the chart components. Therefore, the following CSS rules are equivalent to the ones above. Here, we specify the ID of each chart component, instead of its type, 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 pseudoclasses.*

Styling the Gantt Data

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 the activities or constraints in the Gantt data model that are being rendered. 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. This is explained in more detail in the following sections.

Styling the Gantt data works best when you use the general data model implementation classes provided in the `ilog.views.gantt.model.general` package. 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. See the sections “Selector” and “Model Indirection” in the SDM User’s Manual for more details.

> ***Note:*** *If you do not use the* `IlvGeneralActivity` *and* `IlvGeneralConstraint` *data model implementations, you will not be able to use attribute selectors or perform model indirection in your style sheets.*

Styling Activities

The activity element 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 the “Class Property Name” section of the SDM User’s Manual. Here is an extremely simple CSS rule that will display all activities using an `IlvBasicActivityBar` renderer:

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

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

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

The following table summarizes the CSS elements, tokens, and functions that are applicable when styling activities. Additional detail about each item in the table is discussed in the subsequent sections.

Table 7.3 *Styling Activities With CSS*

Model Object	An instance of `IlvActivity`. An `IlvGeneralActivity` provides the most flexibility.
Model Indirection	Properties of `IlvGeneralActivity`. Not supported for other `IlvActivity` implementations.
Target Object Class	`IlvActivityRenderer` or `IlvActivityRendererFactory` or `IlvGraphic.`
CSS Element Type	activity

Table 7.3 Styling Activities With CSS

CSS ID	The ID property of the activity: `IlvActivity.getID()` or `IlvGeneralActivity.getProperty(“id”).`
CSS Declaration Properties	Bean properties of the target object.
CSS Classes	The tags property of `IlvGeneralActivity`: `IlvGeneralActivity.getProperty(“tags”)` Not supported for other `IlvActivity` implementations.
CSS Pseudoclasses	parent leaf milestone selected
CSS Attribute Selectors	Properties of `IlvGeneralActivity`. Not supported for other `IlvActivity` implementations.

CSS Custom Functions	`formatDate()`
	`formatDuration()`

Activity Renderer Target Objects

As shown in Table 7.3, *Styling Activities With CSS*, the target object to which the CSS declarations will be applied can be an instance of `IlvActivityRenderer`, `IlvActivityRendererFactory`, or `IlvGraphic`. If you specify an `IlvGraphic` class, it will be instantiated and then wrappered in an `IlvActivityGraphicRenderer` by the Gantt CSS engine. Most `IlvGraphic` implementations provided in the JViews distribution do not have zero argument constructors. Therefore, you will need to specify the required constructor arguments in the CSS declaration. Here is an example that shows how to specify a filled `IlvRectangle` graphic as an activity renderer:

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

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

Notice how we provide a dummy `IlvRect` object 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` as your target object, the Gantt CSS engine will ask the factory to create the activity renderer. However, we do not recommend that you 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 customizations. 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:

Table 7.4 *Renderers For CSS Styling of Activities*

Renderer	Bean Properties	Type

`IlvBasicActivityBar`	background
	bottomMargin
	font
	foreground
	label
	style
	thickness
	toolTipText
	topMargin
		`Color`
		`float`
		`Font`
		`Color`
		`String`
		`enum`
		`int`
		`String`
		`float`

`IlvBasicActivityLabel`	background
	bottomMargin
	font
	foreground
	horizontalAlignment
	label
	offset
	toolTipText
	topMargin
	verticalAlignment
		`Color`
		`float`
		`Font`
		`Color`
		`enum`
		`String`
		`float`
		`String`
		`float`
		`enum`

`IlvBasicActivitySymbol`	alignment
	background
	bottomMargin
	foreground
	shape
	toolTipText
	topMargin
		`enum`
		`Color`
		`float`
		`Color`
		`enum`
		`String`
		`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 have written yourself. The following example shows how to use an `IlvActivityCompositeRenderer` to create a more complex renderer from simpler ones:

```
activity {
  class   : 'ilog.views.gantt.graphic.renderer.IlvActivityCompositeRenderer';
  renderer[0] : @#bar;
  renderer[1] : @#startSymbol;
  renderer[2] : @#endSymbol;
}

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

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

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

Activity ID Selectors

As shown in Table 7.3 on page 109, 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 elements 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 for the CSS Examples* on page 101 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 element 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 we do not recommend that you create a data model implementation where the ID of data elements 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, 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 provided examples, described in *The Gantt and Schedule CSS Examples* on page 101, populate their data model with `IlvGeneralActivity` instances.You can therefore use these examples to test and experiment with the styling features described in this section.

IlvGeneralActivity Model Indirection

An introduction to CSS model indirection is provided in the ILOG JViews Stylable Data Mapper (SDM) User's Manual. If your Gantt data model uses instances of `IlvGeneralActivity`, you can use the @ construct on the right side of your declarations to reference properties of the activity. As an example, here is a rule that labels the `IlvBasicActivityBar` renderer with the ID of the activity. The tooltip is rendered using HTML formatting and displays the activity name and ID on separate lines, bold, and centered:

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

IlvGeneralActivity Attribute Selectors

An introduction to CSS attribute selectors is provided in the ILOG JViews Stylable Data Mapper (SDM) User's Manual. Properties of the `IlvGeneralActivity` instances in your data model can be used to match against attribute values in your CSS rule selectors. For example, here are some cascaded rules from the provided `activity-completion.css` style sheet that show a typical use of attribute selectors:

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

activity[completion > '0'] {
        label       : '@|@id+", "+@completion';
}

activity[completion >= '0.25'] {
        background  : coral;
        foreground  : black;
}

activity[completion >= '0.5'] {
        background  : lightsalmon;
}

activity[completion >= '0.75'] {
        background  : khaki;
}

activity[completion >= '0.9'] {
        background  : lemonchiffon;
}

activity[completion >= '0.95'] {
        background  : honeydew;
}

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

The first rule establishes the default rendering for all activities to be an `IlvBasicActivityBar` that is labeled with the ID of the activity. As explained in *Scheduling Data for the CSS Examples* on page 101, the default data model of the provided examples defines a numeric “completion” property for some of the activities. The style sheet uses the value of the completion property to select which of the subsequent rules will be cascaded and will further refine the rendering of the activity. If the activity’s completion value is greater than zero, the label will contain the activity ID and the completion value. The remaining rules set the background color of the activity bar based upon how the activity matches against several completion threshold levels.

IlvGeneralActivity CSS Classes

The Gantt CSS engine interprets the “tags” property of an `IlvGeneralActivity` as the space-separated list of the CSS classes it belongs to. For example, the default data model of the provided examples 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 Table 7.3, *Styling Activities With CSS*, the Gantt CSS engine defines several activity pseudoclasses that you can use in your rule selectors. These 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 we have given 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 we can provide a symbol renderer for these activities by using the `milestone` pseudoclass in our 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:*** *We have used the special @ value to ignore the label property declaration that the milestone rule has inherited.*

The formatDate and formatDuration Functions

As shown in Table 7.3 on page 109, the Gantt CSS engine provides 2 predefined functions you can use as part of an expression in your style sheet: `formatDate` and `formatDuration`. 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>)
```

Here is an example declaration used to set the tooltip to the formatted start time of the activity:

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

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

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

Here is an example 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 the `demos/gantt/css/data` directory.

Styling Constraints

The `constraint` element 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. Here is an extremely simple CSS rule that will render all constraints using the standard `IlvConstraintGraphic` class:

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

We can then add additional declarations to the CSS rule that specify bean properties of the `IlvConstraintGraphic` target object that we want to customize:

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

The following table summarizes the CSS elements, tokens, and functions that are applicable when styling constraints. Additional detail about each item in the table is discussed in the subsequent sections.

Table 7.5 *Styling Constraints With CSS*

Model Object	An instance of `IlvConstraint`. An `IlvGeneralConstraint` provides the most flexibility.
Model Indirection	Properties of `IlvGeneralConstraint`. Not supported for other `IlvConstraint` implementations.
Target Object Class	`IlvConstraintGraphic` or `IlvConstraintGraphicFactory`
CSS Element Type	constraint
CSS ID	The ID property of `IlvGeneralConstraint`: `IlvGeneralConstraint.getProperty(“id”)`. Not supported for other `IlvConstraint` implementations.
CSS Declaration Properties	Bean properties of the target object.
CSS Classes	The tags property of `IlvGeneralConstraint`: `IlvGeneralConstraint.getProperty(“tags”)`. Not supported for other `IlvConstraint` implementations.
CSS Pseudo Classes	selected
CSS Attribute Selectors	Properties of `IlvGeneralConstraint`. Not supported for other `IlvConstraint` implementations.

CSS Custom Functions	`activityProperty()`
	`formatDate()`
	`formatDuration()`

Constraint Graphic Target Objects

As shown in Table 7.5, *Styling Constraints With CSS*, 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:

***Table 7.6** Constraint Graphic Bean Properties*

Bean Properties	Bean Properties	Allowed Values
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 Selectors

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, 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 section Activity ID Selectors on page 112 apply to constraint ID selectors.*

IlvGeneralConstraint Properties

If your Gantt data model uses the `IlvGeneralConstraint` implementation, 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 provided examples, described in *The Gantt and Schedule CSS Examples* on page 101, populate their data model with `IlvGeneralConstraint` instances. You can therefore use these examples to test and experiment with the styling features described in this section.

IlvGeneralConstraint Model Indirection

As shown in Table 7.5, *Styling Constraints With CSS*, if your Gantt data model uses instances of `IlvGeneralConstraint`, you can use the @ construct on the right side of your declarations to reference properties of the constraint. As an example, here is a rule that sets the tooltip of the constraint graphic to be its type, such as End-Start, Start-Start, and so on. The tooltip is rendered using HTML formatting and displays the constraint type in bold:

```
constraint {
  class      : 'ilog.views.gantt.graphic.IlvConstraintGraphic';
  foreground : green;
  lineWidth  : 2;
  toolTipText : '@|"<html><b>"+@constraintType+"</b></html>"';
}
```

IlvGeneralConstraint Attribute Selectors

Properties of the `IlvGeneralConstraint` instances in your data model can also be used to match against attribute values in your CSS rule selectors. For example, you could add the following cascaded constraint rules to the previous example in order to display different constraint types in different colors:

```
constraint {
  class      : 'ilog.views.gantt.graphic.IlvConstraintGraphic';
  foreground : green;
  lineWidth  : 2;
  toolTipText : '@|"<html><b>"+@constraintType+"</b></html>"';
}

constraint[constraintType="Start-Start"] {
  foreground : magenta;
}

constraint[constraintType="Start-End"] {
  foreground : blue;
}

constraint[constraintType="End-End"] {
  foreground : orange;
}
```

IlvGeneralConstraint CSS Classes

The Gantt CSS engine interprets the “tags” property of an `IlvGeneralConstraint` as the space-separated list of the CSS classes it belongs to. This is identical in concept to *IlvGeneralActivity CSS Classes* on page 114. 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”);
```

We 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 Table 7.5, *Styling Constraints With CSS*, the Gantt CSS engine defines the `selected` pseudoclass that you can use in your rule selectors. Here is an example of 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 Table 7.5, *Styling Constraints With CSS*, the Gantt CSS engine provides 3 predefined functions that you can use as part of an expression in your style sheet. We already discussed two of the functions in *The formatDate and formatDuration Functions* on page 115. The `activityProperty` function lets you refer to a property of the constraint’s `from` activity or `to` activity. The syntax of this function is:

```
activityProperty(<IlvGeneralActivity>, <property name>)
```

Here is an example 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 `demos/gantt/css/data` directory.

CHAPTER

8

The Gantt Printing Framework

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.

The Gantt package 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 Gantt printing framework

> ***Note:*** *Before reading this chapter you should familiarize yourself with the Printing Framework User's Manual.*

Introduction

The Gantt printing framework extends the basic JViews 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:

- A Gantt printing controller, instance of the `IlvGanttPrintingController` class, that controls the printing process.
- A Gantt printable document, instance of the `IlvGanttPrintableDocument` class, that defines the printing configuration and contains the Gantt data you want to print in a set of pages.
- A Gantt sheet printable object, instance of the `IlvPrintableGanttSheet` class, that is used to print a portion of an `IlvGanttSheet`.
- A time scale printable object, instance of the `IlvPrintableTimeScale` class, that is used to print a portion of an `IlvTimeScale`.

Simple Example

Run the example given in the `print` demo. The source code of this example can be found in the `<installdir>/demos/gantt/print/src/print` directory. For clarity reasons, only the source code related to the Gantt printing is detailed here.

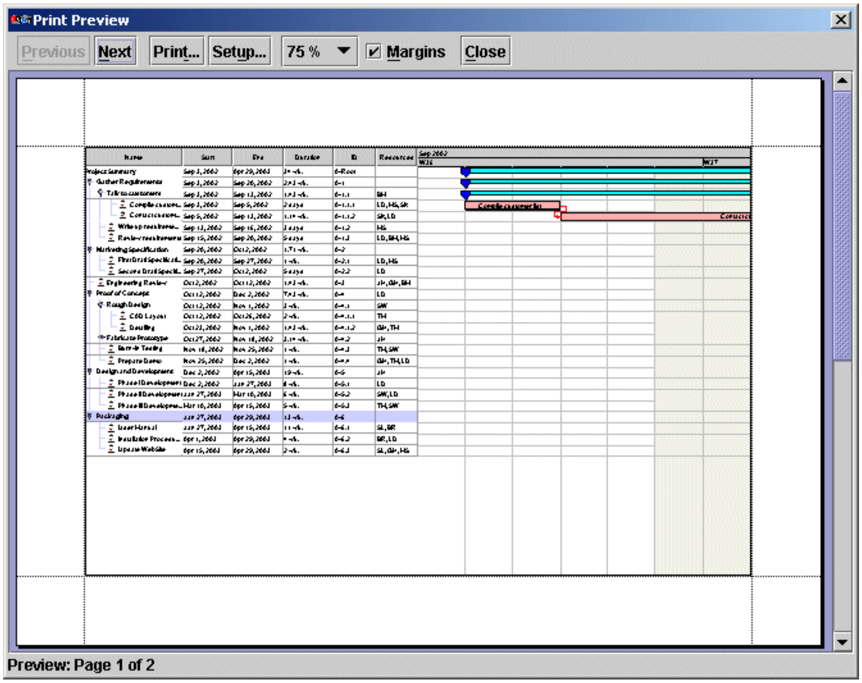
To create an instance of an `IlvGanttPrintingController`, proceed as follows:

```
IlvGanttChart gant = …;
IlvGanttPrintingController printController =
            new IlvGanttPrintingController(gantt);
```

Then, invoke on that instance the action you want to see performed, such as `print()`, `setupDialog()`, or `printPreview()` as shown here:

```
printController.printPreview((java.awt.Frame)gantt.getTopLevelAncestor());
```

The following picture shows the result of the demo:



Preview: Page 1 of 2

Figure 8.1 The Print Preview Dialog Box

Classes Involved

The following classes are involved in the 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.

Here are the different properties you can customize for printing:

- **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 Gantt printing framework provides support for multipage printing through the pages per band property. This represents the number of 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.

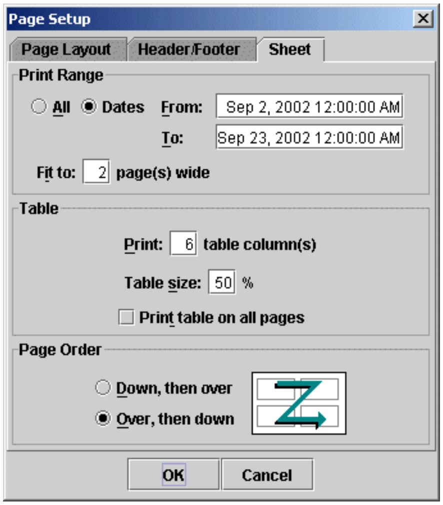
> ***Note:*** *All these properties are also accessible from the Gantt Print Setup dialog box, which you can invoke by calling setupDialog() on the IlvGanttPrintingController.*

The following table summarized the `IlvGanttPrintableDocument` properties:

Property	**Methods**	**Default Value when Automatically Created**
Divider position.	`getDividerPosition` `setDividerPosition`	0.5
End date of the last page in a band.	`getEnd` `setEnd`	The chart visible time + the chart visible duration
Number of pages per band.	`getPagesPerBand` `setPagesPerBand`	2
Whether the table is printed repeatedly on every page.	`getRepeatTable` `setRepeatTable`	False

Property	Methods	Default Value when Automatically Created
Start date of the first page.	`getStart` `setStart`	The chart visible time
Number of columns of the table.	`getTableColumnCount` `setTableColumnCount`	The chart table column count

The default settings are shown here:



Print Range

○ All ● Dates From: Sep 2, 2002 12:00:00 AM

To: Sep 23, 2002 12:00:00 AM

Fit to: 2 page(s) wide

Table

Print: 6 table column(s)

Table size: 50 %

□ Print table on all pages

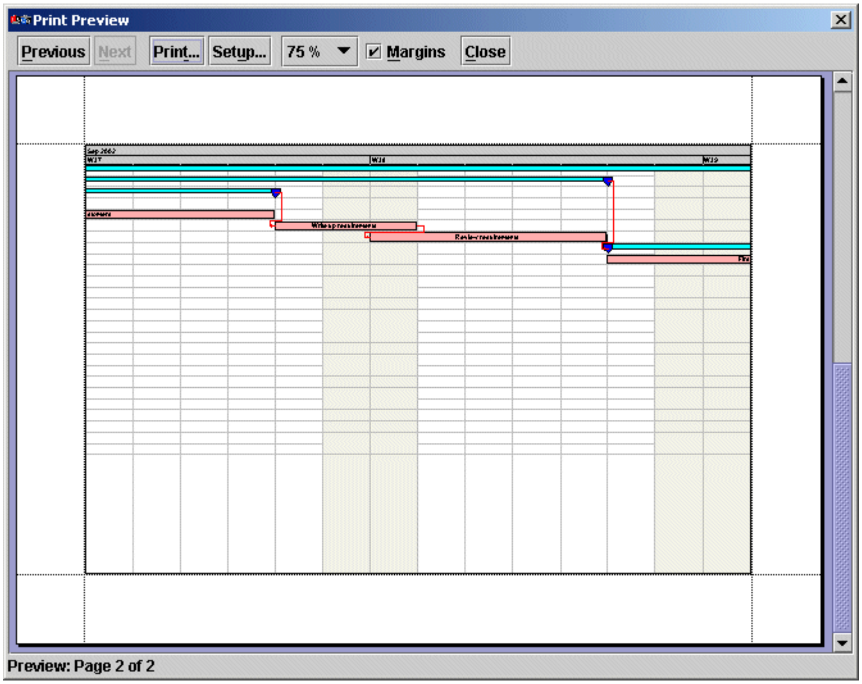
Page Order

○ Down, then over

● Over, then down

Figure 8.2 *Default Print Settings*

With these settings you get 2 pages in the Print Preview window. You can see that the table, with its 6 columns, is only displayed on the first page occupying 50% of the page size. Figure 8.1 shows the first page while Figure 8.3 shows the second page:



75 %

☑ Margins

Preview: Page 2 of 2

Figure 8.3   The 2nd Page of the Print Preview Dialog Box

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 Figure 8.1.

- Through code, by creating a Gantt printable document and setting the parameters as described in section *IlvGanttPrintableDocument*.

IlvPrintableGanttSheet

The IlvPrintableGanttSheet represents the concrete Gantt sheet object that can be printed. It implements the IlvPrintable interface (for more information, see the Printing Framework User's Manual), and 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, users do not need to use this object if they want to print a Gantt using the parameters provided by the Setup dialog box.

Users need to use this class if they want to control their IlvPrintableDocument by creating pages and adding their IlvPrintableObject (see the Printing Framework User's Manual).

IlvPrintableTimeScale

The IlvPrintableTimeScale represents the concrete time scale object that can be printed. It implements the IlvPrintable interface, and lets you print a portion of the IlvTimeScale.

The way the Gantt fills the region is determined by the different document properties. Consequently users do not need to use this object if they want to print a Gantt using the parameters provided by the Setup dialog box.

Users need to use this class if they want to control their IlvPrintableDocument by creating pages and adding their IlvPrintableObject (see the Printing Framework User's Manual).

How it Works

A printing task is initiated and processed by an IlvGanttPrintingController instance. It can be done either by code (using the IlvPrintingController.print(boolean) method) or from a GUI request using the Setup or Preview dialog (by means of the IlvPrintingController.printPreview(Window) and IlvPrintingController.setupDialog(Window, boolean, boolean) methods).

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.

Handling Pages

Pages of a Gantt document are instances of the `ilog.views.util.print.IlvPage` class. They handle a collection of printable objects, instances of `ilog.views.util.print.IlvPrintableObject`.

Populating a Page

Pages created by an `IlvGanttPrintableDocument` are populated in the `IlvGanttPrintableDocument.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`

Example

For a more complex example, you should look in the chart `tablemodel` demo in the `<installdir>/demos/chart/tablemodel` directory, and read the Printing chapter in the Charts User’s Manual.

CHAPTER

9

Thin-Client Support for Web Applications

In the chapters so far, we have discussed how the Gantt Chart module can be used on the client side where you develop Java applets or applications. The Gantt module can also be used on the server side. Some Web applications require that the client stay very light, with most of the functionality residing in the server. The thin-client support in the Gantt module allows you to create such types of applications easily. You can use the power of the ILOG JViews Gantt module to build Gantt or Schedule charts on the Web server. You can then use the Gantt thin-client support on your Web browser to display and interact with those images created by the server.

In this chapter we will see how to use the Gantt packages and classes on both the server side and the client side. The topics are:

- Gantt Thin-Client Web Architecture
- Getting Started With the Gantt Thin Client: An Example
- Developing the Server Side
- Developing the Client Side
- Adding Client/Server Interactions
- The IlvGanttServlet and IlvGanttServletSupport Classes

Gantt Thin-Client Web Architecture

The Gantt thin-client web application support is based on the *Java servlet* technology. Servlets are Java programs that run on a Web server. They act as a middle layer between HTTP requests coming from a Web browser or other HTTP clients (such as applets or applications) and the application or databases on the Web server. The job of the servlet is to read and interpret HTTP requests coming from an HTTP client program and to generate a resulting document that in most cases is an HTML page. For more information about servlet technology, you can visit the JavaSoft site `http://java.sun.com/products/servlet`. This site also provides information about the Web servers that support Java servlets.

For the predefined Gantt thin client, the content created by the servlet is primarily a JPEG or PNG image. The servlet generates the images from a Gantt Chart server side application that is almost identical to the client side Gantt Chart applications we have discussed in previous chapters. The servlet acts as an intermediate layer. It interprets the HTTP requests from the thin client running in the user's browser, generates images of the Gantt Chart server side application, and delivers the images in HTTP responses back to the client. In turn, the Gantt Chart server side application may obtain the scheduling information that it displays from XML files, databases, or other application-specific data. This basic architecture is illustrated in Figure 9.1:

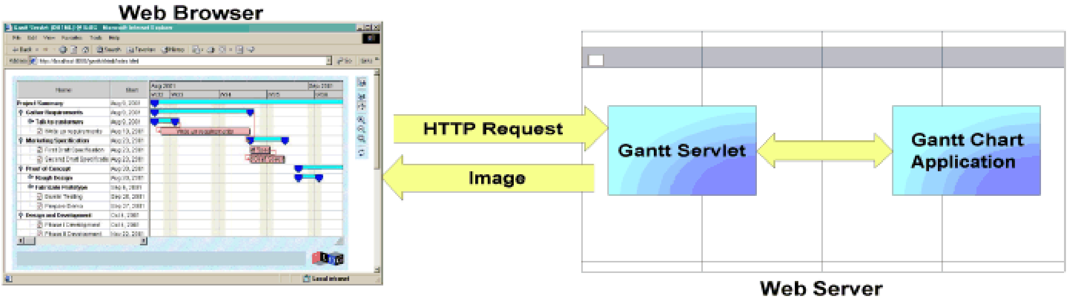


Figure 9.1   Gantt Thin-Client Web Application Architecture

The JViews Gantt thin-client support contains the following:

- An abstract servlet class that can generate images from a Gantt Chart display.
- A set of browser-independent Dynamic HTML scripts written in JavaScript that can be used on the client side to display and interact with the images created on the server side.

Getting Started With the Gantt Thin Client: An Example

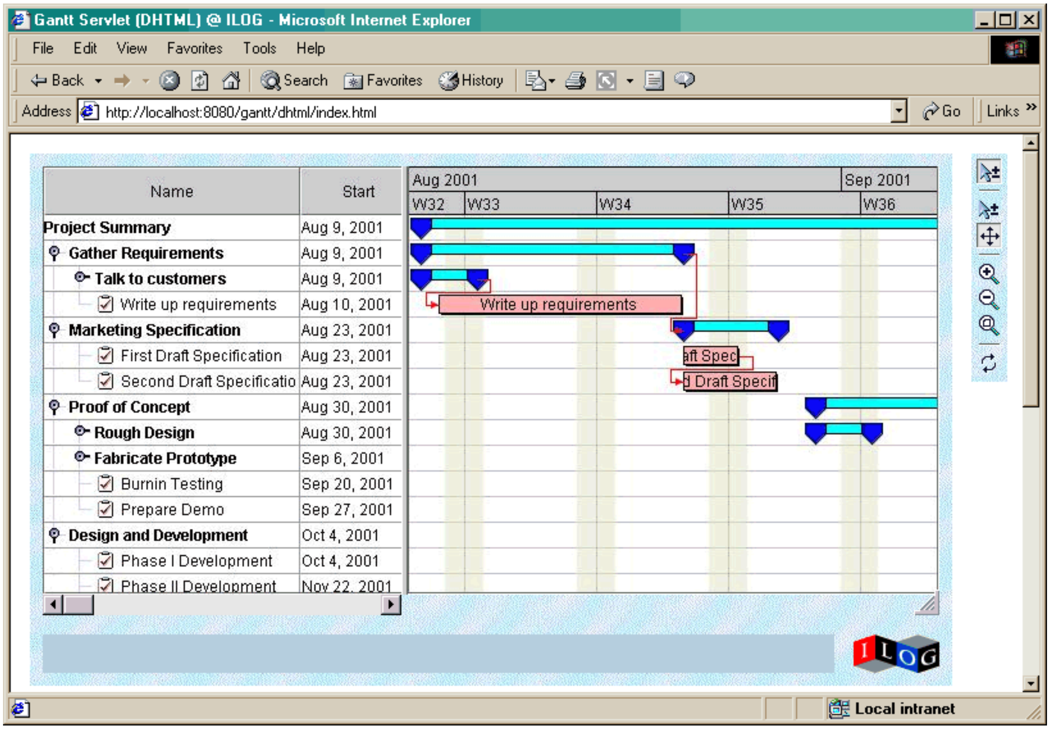
Creating a Gantt thin-client application consists of two steps: developing the server side and developing the client side. The Gantt Servlet example, provided with the distribution, illustrates these steps.

The Gantt Servlet Example

The Gantt Servlet example can be found in the following directory:

```
<installdir>/demos/gantt/servlet
```

This example allows you to show a standard Gantt Chart of scheduling information in a thin-client context.


Gantt Servlet (DHTML) @ ILOG - Microsoft Internet Explorer
File Edit View Favorites Tools Help
Back Search Favorites History
Address http://localhost:8080/gantt/dhtml/index.html
Go Links
Aug 2001 Sep 2001
W32 W33 W34 W35 W36
Write up requirements
aft Spec
d Draft Specif
Local intranet

Name	Start
Project Summary	Aug 9, 2001
Gather Requirements	Aug 9, 2001
Talk to customers	Aug 9, 2001
Write up requirements	Aug 10, 2001
Marketing Specification	Aug 23, 2001
First Draft Specification	Aug 23, 2001
Second Draft Specificatio	Aug 23, 2001
Proof of Concept	Aug 30, 2001
Rough Design	Aug 30, 2001
Fabricate Prototype	Sep 6, 2001
Burnin Testing	Sep 20, 2001
Prepare Demo	Sep 27, 2001
Design and Development	Oct 4, 2001
Phase I Development	Oct 4, 2001
Phase II Development	Nov 22, 2001

***Figure 9.2** The Gantt Servlet Example*

The Gantt Servlet example is composed of the following:

- The server side, which consists of 3 Java files located in the directory:

  ```
  <installdir>/demos/gantt/servlet/src
  ```

These files are:

File	Description
`GanttChartServlet.java`	A servlet that produces JPEG images from a standard `IlvGanttChart` component.
`SimpleProjectDataModel.java`	A Gantt data model that contains the project scheduling information.
`WeekendGrid.java`	A customized vertical grid that is used with the `IlvGanttChart` to highlight weekend periods.

- The DHTML client, which consists of:
  - The HTML starting page:

    `<installdir>/demos/gantt/servlet/web/index.html`
  - The set of JViews common JavaScript DHTML components, located in:

    `<installdir>/classes/thinclient/javascript`

    and the images needed for these components in:

    `<installdir>/classes/thinclient/javascript/images`
  - The set of Gantt JavaScript DHTML components, located in:

    `<installdir>/classes/thinclient/javascript/gantt`

    and the images needed for these components in:

    `<installdir>/classes/thinclient/javascript/gantt/images`

Installing and Running the Gantt Servlet Example

To run the Gantt Servlet example requires a Web server and a Web browser that support Dynamic HTML. The Web server must support the Servlet API 2.1 or later.

The Gantt Servlet example contains a WAR file (Web Archive):

`<installdir>/demos/gantt/servlet/gantt.war`

that allows you to easily install the example on the Web server of your choice. You can check the latest list of servers that support servlets at

`http://java.sun.com/products/servlet/industry.html`

For your convenience, we have supplied the Tomcat web server with the ILOG JViews distribution. The Gantt Servlet Example is pre-installed in Tomcat and is ready to run. TOMCAT is the official reference implementation of the Servlet and JSP specifications. To get more information on Tomcat go to `http://jakarta.apache.org/tomcat/`.

Browsers that support DHTML are Netscape Communicator 4.x (on Windows and UNIX platforms), and Internet Explorer 5.0 and higher (on Windows platforms).

Here are the steps for running the example on the Tomcat web server supplied with the ILOG JViews installation:

1. Set the JAVA_HOME environment variable to point to your Java Development Kit installation.
2. Go to the Tomcat `bin` directory located in `<installdir>/tools/tomcat/bin`, where `<installdir>` is the directory in which ILOG JViews is installed.
3. Depending on your system, run the `startup.bat` or `startup.sh` script to run the Tomcat server.
4. Launch a Web browser and open the page: `http://localhost:8080/gantt`
5. For additional details, consult the ILOG JViews installation and setup instructions in:

   `<installdir>/html/installation.html`

Your browser then shows the Gantt Servlet example.

Developing the Server Side

The server side of a Gantt thin-client application is composed of two main parts: the Gantt application itself, which can be any type of Gantt Chart or Schedule Chart built upon the Gantt module components and APIs, and a Servlet that interprets requests from the client to generate images of the chart. Here is an overview of the key classes and their relationships:

9. Thin-Client Support

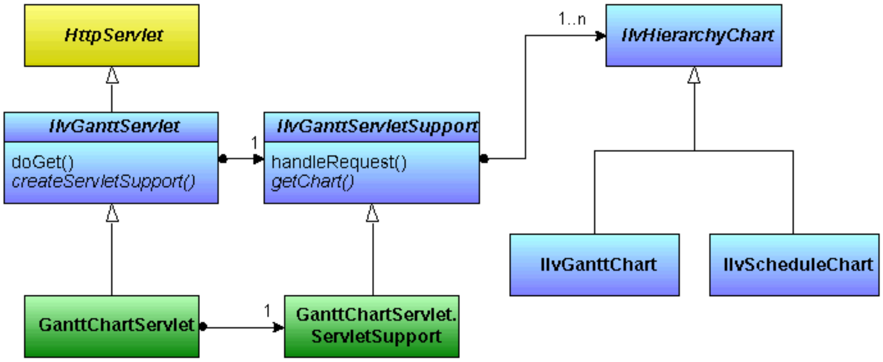


Figure 9.3 Overview of Key Server-Side Classes

The server-side classes are colored to indicate their packaging:

- The yellow class, `HTTPServlet`, is part of the standard Java Servlet API. It is located in the `javax.servlet.http` package and is the abstract base class for all HTTP servlet implementations.
- The blue classes are members of the Gantt module API. The abstract classes `IlvGanttServlet` and `IlvGanttServletSupport` belong to the `ilog.views.gantt.servlet` package. `IlvGanttServlet` is an abstract servlet that responds to HTTP requests to generate images of a Gantt Chart or a Schedule Chart. `IlvGanttServlet` is a very simple class that delegates all its real work to an instance of `IlvGanttServletSupport`. This allows you to easily integrate the full capabilities of the Gantt server-side classes into your own servlet implementations.
- The green classes belong to the Gantt Servlet Example and are located in the file:

  ```
  <installdir>/demos/gantt/servlet/src/GanttChartServlet.java
  ```

  `GanttChartServlet` is the concrete servlet implementation for the server side of the example. Its concrete inner support class, `GanttChartServlet.ServletSupport`, generates images of a standard Gantt Chart in response to HTTP requests.

To analyze these parts, we will see how the server side is built in the Gantt Servlet example.

The Servlet Support Class

The Gantt Servlet example displays a standard Gantt Chart containing project scheduling information. The `IlvGanttServletSupport` class does all the work on the server side to generate images of the chart in response to HTTP requests. The concrete implementation for this example is the `IlvGanttChartServlet.ServletSupport` inner class, located in the file:

```
<installdir>/demos/gantt/servlet/src/GanttChartServlet.java
```

The getChart Method

The method:

```
public IlvHierarchyChart getChart(HttpServletRequest,
                                  IlvServletRequestParameters)
    throws ServletException
```

is the only abstract method of the `IlvGanttServletSupport` class. It should return the `IlvGanttChart` or `IlvScheduleChart` that will be used to satisfy an HTTP request. The request is given as a parameter to the `getChart` method, so it is possible to provide charts to the client that are session-specific. The servlet support class of the example has been simplified to use a single Gantt Chart instance to satisfy all HTTP requests. This means that every client will see the same data.

The Example Code

First, we include the `import` statements that are required to use the Java Servlet API:

```
import javax.servlet.*;
import javax.servlet.http.*;
```

Then we include the `import` statements that are required for the Gantt module and the Gantt server side classes:

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

The servlet support class of the example is very simple, and consists of only 2 methods:

```
class ServletSupport extends IlvGanttServletSupport
{
  private IlvHierarchyChart _chart;

  /**
   * Creates the Gantt chart that will be used by the servlet to satisfy HTTP
   * requests.
   */
  private IlvHierarchyChart createChart()
  {
    IlvHierarchyChart chart = new IlvGanttChart();
    IlvGanttModel ganttModel = new SimpleProjectDataModel();
    chart.setGanttModel(ganttModel);
    chart.getGanttSheet().setVerticalGrid(new WeekendGrid());
    ... more chart customizations ...
    return chart;
  }

  /**
   * Returns the chart used for the specified request. This implementation
   * always returns the same chart.
   * @param request The current HTTP request.
   * @param params The parameters parsed from the request.
   */
  public IlvHierarchyChart getChart(HttpServletRequest request,
                                    IlvServletRequestParameters params)
    throws ServletException
  {
    synchronized(this) {
      if (_chart == null) {
        _chart = createChart();
      }
    }
    return _chart;
  }
}
```

As you can see, the steps necessary to create a chart on the server side are almost identical to those we have discussed in earlier chapters for developing client-side Java applications and applets. In summary, you will need to:

1. Create a concrete subclass of `IlvGanttServletSupport`.

2. Implement the `getChart` method to return an instance of `IlvGanttChart` or `IlvScheduleChart`.

3. Connect the chart to your application data model and customize the appearance of the chart as you desire.

Multi-Threading Issues on the Server Side

We now need to briefly discuss the threading issues involved when using GUI components, such as the `IlvGanttChart` and `IlvScheduleChart`, on the server side. The Web server run-time environment is inherently multi-threaded. However, the Gantt chart components, like all Swing GUI components, are not multi-thread safe. There is also a further design constraint of Swing GUI components. After the Web server has sent an image of a chart to the client for the first time, all modifications to the visual properties of the chart must be performed on the AWT event dispatch thread. The AWT event dispatch thread will never be the same thread that the HTTP request is being serviced on.

In general, the `IlvGanttServletSupport` base class handles all these threading issues for you. It ensures that all requests to modify a chart and generate its image are moved from the HTTP request thread onto the AWT event dispatch thread as necessary. In the `createChart` method of the servlet support class, we are able to customize the visual properties of the chart on the HTTP request thread because the chart has not been sent to the client yet. However, note the use of the `synchronized` block in the `getChart` method. This is necessary to ensure that only a single chart instance is ever created in the multi-threaded Web server environment.

The Servlet Class

The `IlvGanttServlet` class is a simple HTTP servlet implementation that delegates all its work to its associated support class. It contains a single abstract method:

```
protected IlvGanttServletSupport createServletSupport()
```

This method must return the single support instance that will service the HTTP requests sent to the servlet. The implementation of this class for the example is located in the file:

```
<installdir>/demos/gantt/servlet/src/GanttChartServlet.java
```

It consists of only the `createServletSupport` method:

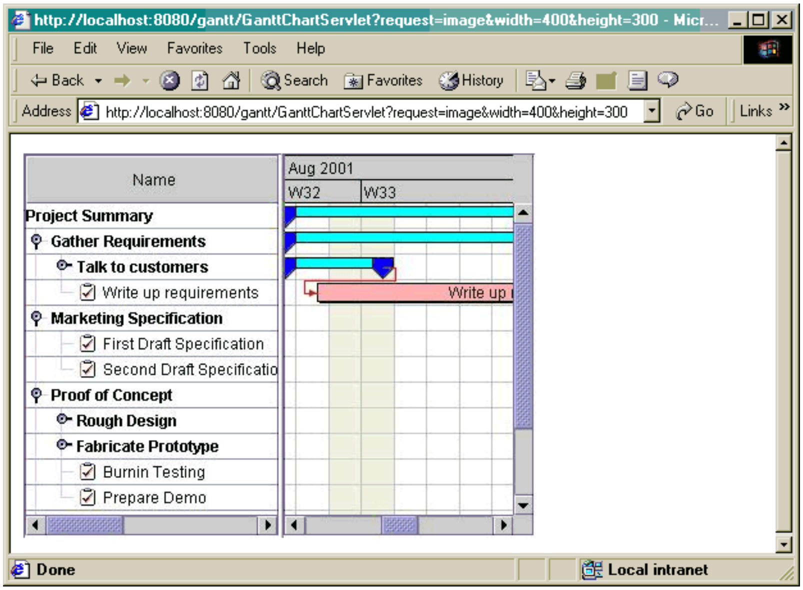
```
public class GanttChartServlet extends IlvGanttServlet
{
  /**
   * Creates the servlet support object to which this servlet delegates HTTP
   * request handling.
   */
  protected IlvGanttServletSupport createServletSupport()
  {
    IlvGanttServletSupport support = new ServletSupport();
    ... customize the support class ...
    return support;
  }
}
```

Summary

As you have seen, creating the server side of the Gantt Chart web application is very simple. The servlet can now answer HTTP requests from a client by sending JPEG images of the chart. If you have the Tomcat server running that is supplied with the ILOG JViews distribution, you can try typing the following HTTP request in your Web browser:

```
http://localhost:8080/gantt/
GanttChartServlet?request=image&width=400&height=300
```

This produces the following image:



In this request, we ask the servlet named `GanttChartServlet` to produce an image of size 400 x 300 showing the entire `IlvGanttChart` component. In most cases you do not have to know the servlet parameters because the client side Dynamic HTML objects provided by the Gantt module will take care of the HTTP requests for you.

Developing the Client Side

After creating the server side, you can create the client side of your Gantt Web application. The Gantt thin-client support allows you to easily build a client based on Dynamic HTML that will run on Web browsers that support DHTML. You build an HTML web page for the DHTML client using predefined JavaScript components.

Developing a Dynamic HTML Client

The static nature of HTML limits the interactivity of Web pages. Dynamic HTML allows you to create Web pages that are more interactive and engaging. It gives content providers new controls and allows them to manipulate the contents of HTML pages through scripting. To learn more about Dynamic HTML, you can visit the following Web sites:

- The Microsoft Web Workshop:

  `http://msdn.microsoft.com/workshop/`

- The Netscape DevEdge pages on DHTML:

  `http://developer.netscape.com/tech/dynhtml/index.html`

The ILOG JViews Gantt module provides a set of browser-independent Dynamic HTML components written in JavaScript that allow you to build your DHTML pages very easily. The JavaScript files are located in the 2 directories:

- `<installdir>/classes/thinclient/javascript`

  Contains the JViews common JavaScript DHTML components.

- `<installdir>/classes/thinclient/javascript/gantt`

  Contains the Gantt JavaScript DHTML components

> ***Warning:*** *Keep in mind that not all versions of Web browsers support DHTML. The ILOG JViews DHTML scripts have been tested on Internet Explorer 5.x and higher (on Windows platforms, and Netscape Communicator 4.x (on Windows and UNIX platforms).*

Common DHTML Components

The common JavaScript DHTML components are located in the directory:

`<installdir>/classes/thinclient/javascript`

Here is an overview of the common DHTML component classes and their relationships:

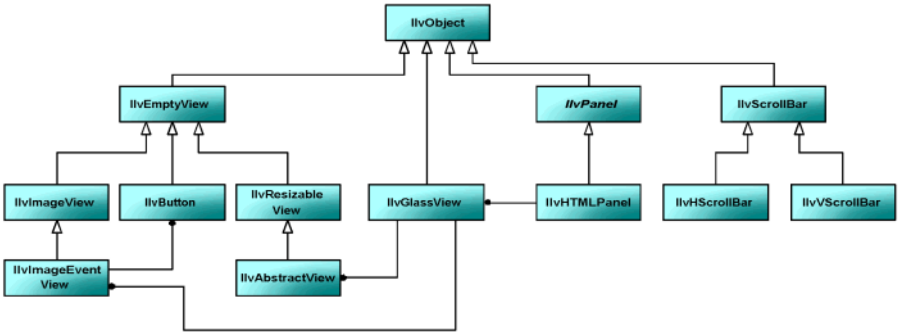


Figure 9.4 *Common DHTML Components*

Here is the list of common JavaScript files and a brief description of each:

Table 9.1 *Common DHTML Script Files*

Script File	Description
`IlvUtil.js`	Dynamic HTML tools and functions used by other scripts. This file must always be included. This file defines the `IlvObject` and `IlvPanel` classes.
`IlvEmptyView.js`	Defines the class `IlvEmptyView`, the base class for all view components that have a size and position on the HTML page.
`IlvImageView.js`	Defines the `IlvImageView` and `IlvImageEventView` classes.
`IlvGlassView.js`	Defines the `IlvGlassView` class.
`IlvResizableView.js`	Defines the `IlvResizableView` class, the base class for all view components that can be interactively resized. This is the base class for `IlvGanttView`.
`IlvAbstractView.js`	Defines the `IlvAbstractView` class, the base class for `IlvGanttComponentView`.
`IlvInteractor.js`	Defines the `IlvInteractor` class, the base class for all view interactors.
`IlvButton.js`	Defines the `IlvButton` class, a simple DHTML button.

Table 9.1 Common DHTML Script Files (Continued)

Script File	Description
`IlvToolBar.js`	Defines the `IlvToolBar` class, a DHTML toolbar that can contain `IlvButtons`.
`IlvInteractorButton.js`	Defines the `IlvInteractorButton` class, a subclass of `IlvButton` that can set an interactor on a view.
`IlvScrollbar.js`	Defines the DHTML scroll bar classes `IlvScrollBar`, `IlvVScrollBar`, and `IlvHScrollBar`.

The full reference documentation of each component can be found in the Dynamic HTML Component Reference located in:

```
<installdir>/doc/jscript/index.html
```

Gantt DHTML Components

The Gantt JavaScript DHTML components are located in the directory:

```
<installdir>/classes/thinclient/javascript/gantt
```

Here is an overview of the Gantt DHTML component classes and their relationships:

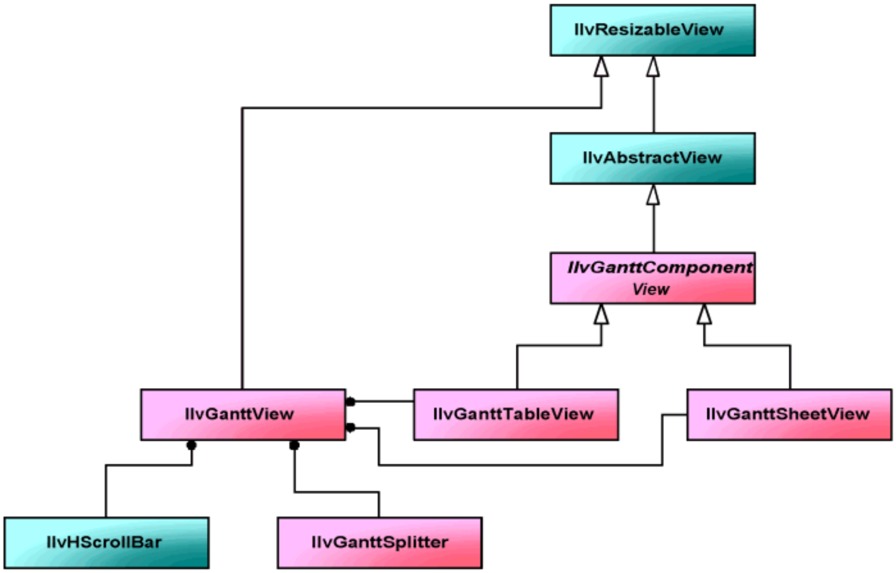


Figure 9.5 Gantt DHTML Components

Here is the list of Gantt JavaScript files and a brief description of each:

***Table 9.2** Gantt DHTML Script Files*

Script File	Description
`gantt.js`	A file containing all the scripts for all the Gantt and Common components. You import this file into your Web page instead of importing all the individual components.
`IlvGanttView.js`	Defines the main Gantt view classes `IlvGanttView`, `IlvGanttComponentView`, `IlvGanttTableView`, and `IlvGanttSheetView`.
`IlvGanttTableScrollInteractor.js`	Defines the class `IlvGanttTableScrollInteractor`, an interactor that lets you pan and scroll an `IlvGanttTableView`.
`IlvGanttSheetScrollInteractor.js`	Defines the class `IlvGanttSheetScrollInteractor`, an interactor that lets you pan and scroll an `IlvGanttSheetView`.
`IlvRowExpandCollapseInteractor.js`	Defines the class `IlvRowExpandCollapseInteractor`, an interactor that lets you expand and collapse rows in an `IlvGanttTableView` or an `IlvGanttSheetView`.

The full reference documentation of each component can be found in the Dynamic HTML Component Reference located in:

`<installdir>/doc/jscript/index.html`

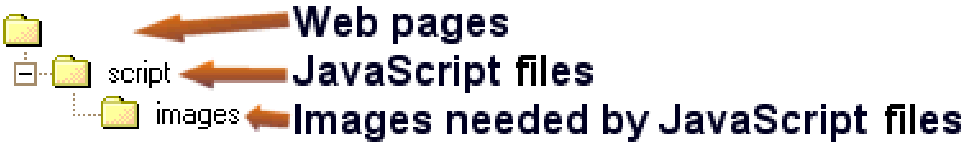
The DHTML Client for the Gantt Servlet Example

We will now create a Dynamic HTML client for the Gantt Servlet example, starting with a very simple example and including most of the DHTML components. The full HTML file for the Gantt Servlet example is located in:

`<installdir>/demos/gantt/servlet/web/index.html`

Web Application Directory Structure

Before we start using the Gantt DHTML components to build our client, we must first decide on the directory structure that the users will see when they visit our Web Application with their browser. This does not, and should not, match the location of the example and JavaScript files in the ILOG JViews distribution. The Gantt Servlet example is deployed to use the following directory structure:



The Ant build file for the Gantt Servlet example:

```
<installdir>/demos/gantt/servlet/build.xml
```

creates this directory structure in the `gantt.war` Web Archive.

The IlvGanttView DHTML Component

The `IlvGanttView` component (located in the `IlvGanttView.js` file) is the main Gantt DHTML component. This component queries the servlet and displays the resulting image of the chart.

First, we must include the JavaScript files that are required to use the `IlvGanttView` component:

- `IlvUtil.js`
- `IlvEmptyView.js`
- `IlvImageView.js`
- `IlvGlassView.js`
- `IlvResizableView.js`
- `IlvAbstractView.js`
- `IlvScrollbar.js`
- `IlvGanttView.js`

Instead of including the individual `.js` files of each component, you can add the file:

```
<installdir>/classes/thinclient/gantt/gantt.js
```

This file is a concatenation of all the `.js` files required for developing DHML thin clients in the Gantt module.

Here is a simple HTML page that creates an `IlvGanttView`:

```
<HTML>
<HEAD>
<META HTTP-EQUIV="Expires" CONTENT="Mon, 01 Jan 1990 00:00:01 GMT">
<META HTTP-EQUIV="Pragma" CONTENT="no-cache">
</HEAD>

<script TYPE="text/javascript"  src="script/IlvUtil.js" ></script>
<script TYPE="text/javascript"  src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript"  src="script/IlvImageView.js"></script>
<script TYPE="text/javascript"  src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript"  src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript"  src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvScrollbar.js"></script>
<script TYPE="text/javascript"  src="script/IlvGanttView.js"></script>

<script TYPE="text/javascript">
function init()
{
  chartView.init();
}

function handleResize()
{
  if (document.layers)
    window.location.reload()
}
</script>
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet.
  var servletName = "/gantt/GanttChartServlet";

  // Position of the Gantt Chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.toHTML();
</script>
</body>
</html>
```

In this example we start by importing the needed JavaScript files:

```
<script TYPE="text/javascript"  src="script/IlvUtil.js" ></script>
<script TYPE="text/javascript"  src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript"  src="script/IlvImageView.js"></script>
```

```
<script TYPE="text/javascript"  src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript"  src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript"  src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvScrollbar.js"></script>
<script TYPE="text/javascript"  src="script/IlvGanttView.js"></script>
```

The JavaScript files must be placed in the head of the page. Note that the scripts are included from the relative `script` subdirectory. Remember that when we build the Web application, the HTML Web pages will be placed in the upper directory and the scripts will be in the `script` directory.

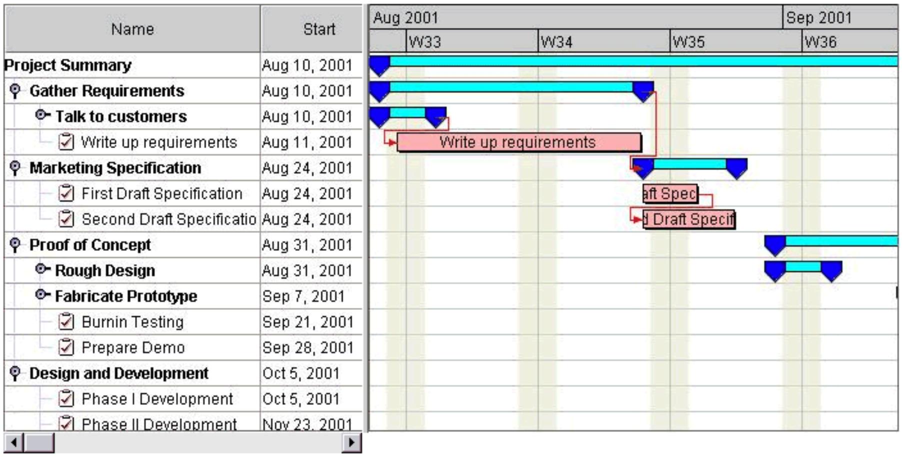
In the body of the page, we create an `IlvGanttView` located in (25, 25) on the HTML page. The size is 350 x 700. This view displays images produced by the servlet `GanttChartServlet`. Note the `toHTML` method that creates the HTML necessary for the component.

This example also defines two JavaScript functions:

- The `init` function, called on the `onload` event of the page, initializes the `IlvGanttView` by calling its `init` method.
- The `handleResize` function, called on the `onresize` event of the page, will reload the page if the browser is Netscape Communicator 4 or higher. This is necessary for a correct resizing of Dynamic HTML content on Communicator.

> ***Note:*** *The global* `ilvDispose` *function must be called in the* `onunload` *event of the HTML page. This function disposes of all the resources acquired by the JViews DHTML components.*

Once the image is loaded from the server, the page looks like this:



Name	Start
Project Summary	Aug 10, 2001
Gather Requirements	Aug 10, 2001
Talk to customers	Aug 10, 2001
Write up requirements	Aug 11, 2001
Marketing Specification	Aug 24, 2001
First Draft Specification	Aug 24, 2001
Second Draft Specificatio	Aug 24, 2001
Proof of Concept	Aug 31, 2001
Rough Design	Aug 31, 2001
Fabricate Prototype	Sep 7, 2001
Burnin Testing	Sep 21, 2001
Prepare Demo	Sep 28, 2001
Design and Development	Oct 5, 2001
Phase I Development	Oct 5, 2001
Phase II Development	Nov 23, 2001

The Message Panel

We will now add a Dynamic HTML panel below our main view. A DHTML panel is an area of the page that can contain some HTML content. We will use the DHTML panel to display status messages as the user interacts with the Gantt view. We create the message panel using the class `IlvHTMLPanel`, defined in the `IlvUtil.js` file.

The body of the page is now:

> ***Note:*** *The lines added are in bold.*

```
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet.
  var servletName = "/gantt/GanttChartServlet";

  // Position of the Gantt Chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.toHTML();

  var messagePanel = new IlvHTMLPanel('');
  messagePanel.setBackgroundColor('#B6D5DA');
  messagePanel.setVisible(true);
  chartView.setMessagePanel(messagePanel);
  var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop() + chart.getHeight() + 15,
                           chart.getWidth(),
                           45);
  }
  layoutPage(chartView);
  chartView.addSizeListener(layoutPage);

</script>
</body>
```

Note that the `IlvHTMLPanel` does not have a `toHTML` method, it generates its HTML content immediately from within its constructor. Also, the `IlvHTMLPanel` is initially hidden. You must explicitly call its `setVisible` method to show it on the page. These are the main differences between the DHTML "view" components and the DHTML "panel" components.

We anticipate that we will make the main Gantt view interactively resizable. Therefore, we have created a `layoutPage` function that positions the message panel relative to the current size and position of the main Gantt view:

```
var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop() + chart.getHeight() + 15,
                           chart.getWidth(),
                           45);
  }
```

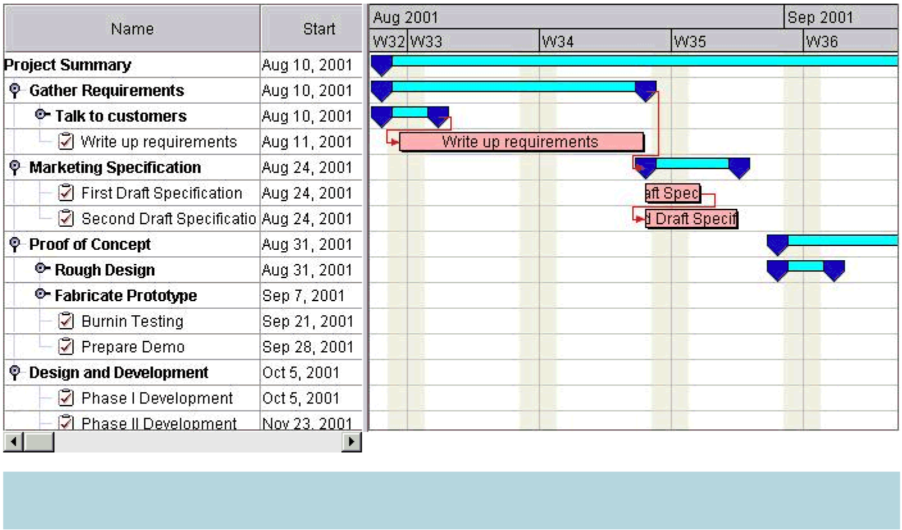
We then call `layoutPage` to perform the initial arrangement of the components:

```
layoutPage(chartView);
```

And we add the `layoutPage` to listen to resize events from the `IlvGanttView`:

```
chartView.addSizeListener(layoutPage);
```

Our web page now looks like this:



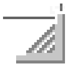
Name	Start
Project Summary	Aug 10, 2001
Gather Requirements	Aug 10, 2001
Talk to customers	Aug 10, 2001
Write up requirements	Aug 11, 2001
Marketing Specification	Aug 24, 2001
First Draft Specification	Aug 24, 2001
Second Draft Specificatio	Aug 24, 2001
Proof of Concept	Aug 31, 2001
Rough Design	Aug 31, 2001
Fabricate Prototype	Sep 7, 2001
Burnin Testing	Sep 21, 2001
Prepare Demo	Sep 28, 2001
Design and Development	Oct 5, 2001
Phase I Development	Oct 5, 2001
Phase II Development	Nov 23, 2001

Interactively Resizing the IlvGanttView

We will now make the main Gantt view interactively resizable by calling the `setResizable` method:

```
var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
chartView.setServletURL(servletName);
chartView.setResizable(true);
chartView.toHTML();
```

Our `IlvGanttView` now displays a resize tool at its lower right corner:



We can now click and drag on the tool to interactively resize the Gantt view. When the resize operation completes, the `layoutPage` method is invoked, and the position of the message panel is updated to match that of the main view.

Decorative Panels

Next, we will add some decorative panels around our main Gantt view to improve the appearance of the web page. We use an `IlvHTMLPanel` to display a tiled image pattern as a background frame:

```
var backgroundPanel = new IlvHTMLPanel('');
backgroundPanel.setBackgroundImage(ilvImagePath + 'skybg.jpg');
backgroundPanel.setBackgroundColor('#909090');
backgroundPanel.setVisible(true);
```

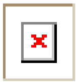
The background panel must be created before the `IlvGanttView.toHTML` method is invoked. DHTML components have an implied z-order in the browser that is determined by the order that their HTML code is created in the page body. The `IlvHTMLPanel` component creates its HTML code in its constructor and the `IlvGanttView` component creates its HTML code in its `toHTML` method. By placing the background panel before the Gantt view in the page body, we ensure that the panel will appear behind the Gantt view.

The `ilvImagePath` variable, used to define the tiled image for the background panel, is a global variable defined in `IlvUtil.js`. It contains the path to the images used by the script files. Its default value is `script/images`, which is the location of the image files relative to the web pages in our Web application.

We also add a small company logo to display on the right side of the message panel. We use an `IlvImageView` component instead of an `IlvHTMLPanel` because we do not want to tile the logo image:

```
var logoPanel = new IlvImageView(0, 0, 67, 30,
                                 ilvImagePath+'ilog-small.gif');
logoPanel.toHTML();
```

The `IlvImageView` component has the additional advantage of remaining hidden until its image is loaded. This is important for a nice appearance when the images take some time to download from the server due to image size or network latencies. Normally, if an image has not been loaded from the server yet, the browser will display a box with a red “X” in it:



The `IlvImageView` component avoids this effect and remains invisible until the image is available to display.

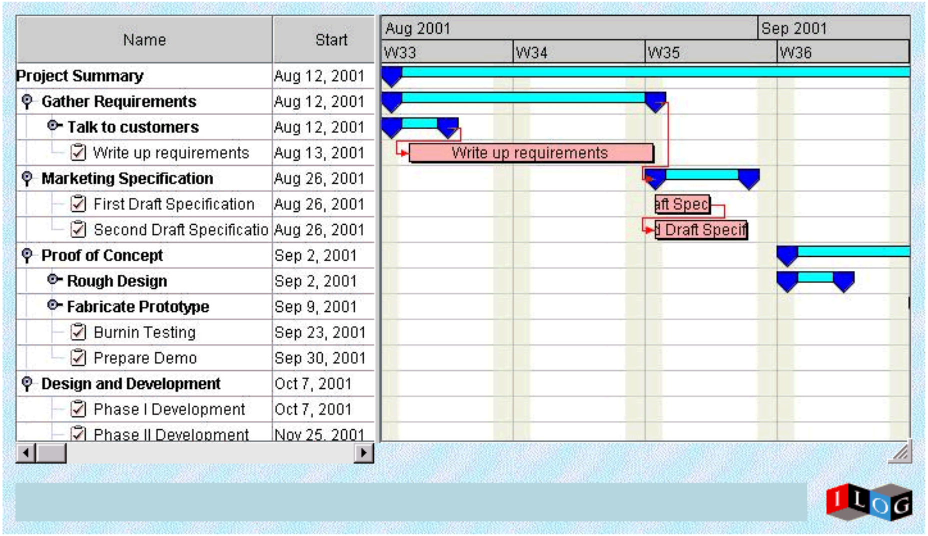
Finally, we must update the `layoutPage` method to properly arrange the new panels:

```
var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop()+chart.getHeight()+15,
                           chart.getWidth()-logoPanel.getWidth()-15,
                           logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop()  - 10,
                              chart.getWidth()+ 20,
                              messagePanel.getTop() +
                                  messagePanel.getHeight() + 20 -
                                      chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft() +
                              messagePanel.getWidth() + 15,
                          messagePanel.getTop());
  }
```

The body of the HTML file now looks like:

```
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet
  var servletName = "/gantt/GanttChartServlet";

  // Position of the Gantt Chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var backgroundPanel = new IlvHTMLPanel('');
  backgroundPanel.setBackgroundImage(ilvImagePath + 'skybg.jpg');
  backgroundPanel.setBackgroundColor('#909090');
  backgroundPanel.setVisible(true);

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.setResizable(true);
  chartView.toHTML();

  var messagePanel = new IlvHTMLPanel('');
  messagePanel.setBackgroundColor('#B6D5DA');
  messagePanel.setVisible(true);
  chartView.setMessagePanel(messagePanel);

  var logoPanel = new IlvImageView(0, 0, 67, 30, ilvImagePath+'ilog-small.gif');
  logoPanel.toHTML();

  var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop() + chart.getHeight() + 15,
                            chart.getWidth() - logoPanel.getWidth() - 15,
                        logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop()  - 10,
                              chart.getWidth()+ 20,
                                   messagePanel.getTop() +
                                            messagePanel.getHeight() + 20 -
                                        chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()+messagePanel.getWidth()+15,
                      messagePanel.getTop());
  }
  layoutPage(chartView);
  chartView.addSizeListener(layoutPage);

</script>
</body>
```

You should now see the following Web page:



Name	Start
Project Summary	Aug 12, 2001
Gather Requirements	Aug 12, 2001
Talk to customers	Aug 12, 2001
Write up requirements	Aug 13, 2001
Marketing Specification	Aug 26, 2001
First Draft Specification	Aug 26, 2001
Second Draft Specificatio	Aug 26, 2001
Proof of Concept	Sep 2, 2001
Rough Design	Sep 2, 2001
Fabricate Prototype	Sep 9, 2001
Burnin Testing	Sep 23, 2001
Prepare Demo	Sep 30, 2001
Design and Development	Oct 7, 2001
Phase I Development	Oct 7, 2001
Phase II Development	Nov 25, 2001

IlvToolBar and IlvButton

The `IlvButton` class is a simple DHTML button component that allows you to call some JavaScript code when the user clicks on it. The `IlvToolBar` class is a component that can be used to arrange `IlvButtons` vertically or horizontally. We will now add some buttons to our page to zoom in and out on the chart.

First, we must include the JavaScript files that define the `IlvButton` and `IlvToolBar` classes. Our JavaScript import statements now look like this:

```
<script TYPE="text/javascript"  src="script/IlvUtil.js" ></script>
<script TYPE="text/javascript"  src="script/IlvEmptyView.js"></script>
<script TYPE="text/javascript"  src="script/IlvImageView.js"></script>
<script TYPE="text/javascript"  src="script/IlvGlassView.js"></script>
<script TYPE="text/javascript"  src="script/IlvResizableView.js"></script>
<script TYPE="text/javascript"  src="script/IlvAbstractView.js"></script>
<script TYPE="text/javascript"  src="script/IlvScrollbar.js"></script>
<script TYPE="text/javascript"  src="script/IlvGanttView.js"></script>
<script TYPE="text/javascript"  src="script/IlvButton.js"></script>
<script TYPE="text/javascript"  src="script/IlvToolBar.js"></script>
```

In the page body, we first create the vertical toolbar and give it the same background image as the main background panel:

```
var backgroundPattern = ilvImagePath + 'skybg.jpg';
var toolbar = new IlvToolBar(0, 0);
toolbar.setOrientation(IlvToolBar.VERTICAL);
toolbar.setBackgroundColor('#53537A');
toolbar.setBackgroundImage(backgroundPattern);
```

Notice that we set the initial position of the toolbar to (0, 0) and do not call its `toHTML` function immediately. We will wait until the `layoutPage` function calculates the correct position of the toolbar and then generate its HTML code. This approach makes the page more maintainable by encapsulating all the component positioning into the `layoutPage` function and eliminates complex initial position calculations when we create each component.

Next, we create 3 buttons and add them to the toolbar. Each button is defined by its position, size, three images, and a piece of JavaScript to be executed when the button is clicked. The actual positioning of each button will be controlled by the toolbar it is contained in, so we simply set each button's initial position to (0, 0). The three images used by the button are:

- The main image specified in the `IlvButton` constructor. This is the image used when the mouse is not over the button and the button is not selected.
- The rollover image is used when the mouse is over the button, but the button is not selected.
- The selected image is used when the mouse button is pressed on the button.

The code to create the 3 zoom buttons is:

```
// Create the Zoom-In toolbar button.
var zoomInAction = function() {
  chartView.zoomIn();
};
var zoomInButton = new IlvButton(0, 0, 20, 20,
                                 ilvImagePath+'zoomin-up.gif',
                                 zoomInAction);
zoomInButton.setRolloverImage(ilvImagePath+'zoomin-sel.gif');
zoomInButton.setSelectedImage(ilvImagePath+'zoomin-dn.gif');
zoomInButton.setToolTipText("Zoom In");
zoomInButton.setMessage("Press to zoom in");
zoomInButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomInButton);

// Create the Zoom-Out toolbar button.
var zoomOutAction = function() {
  chartView.zoomOut();
};
var zoomOutButton = new IlvButton(0, 0, 20, 20,
                                  ilvImagePath+'zoomout-up.gif',
                                  zoomOutAction);
zoomOutButton.setRolloverImage(ilvImagePath+'zoomout-sel.gif');
zoomOutButton.setSelectedImage(ilvImagePath+'zoomout-dn.gif');
zoomOutButton.setToolTipText("Zoom Out");
zoomOutButton.setMessage("Press to zoom out");
zoomOutButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomOutButton);

// Create the Zoom-To-Fit toolbar button.
```

```
  var zoomFitAction = function() {
    chartView.zoomToFit();
  };
  var zoomFitButton = new IlvButton(0, 0, 20, 20,
                                    ilvImagePath+'zoomfit-up.gif',
                                    zoomFitAction);
  zoomFitButton.setRolloverImage(ilvImagePath+'zoomfit-sel.gif');
  zoomFitButton.setSelectedImage(ilvImagePath+'zoomfit-dn.gif');
  zoomFitButton.setToolTipText("Zoom To Fit");
  zoomFitButton.setMessage("Press to zoom to fit");
  zoomFitButton.setMessagePanel(messagePanel);
  toolbar.addButton(zoomFitButton);
```

The `zoomInButton` simply calls the `zoomIn` method of the `IlvGanttView`, the `zoomOutButton` calls the `zoomOut` method, and the `zoomFitButton` calls the `zoomToFit` method. Each button also has a `message` property. The message will be automatically displayed in the status window of the browser when the mouse is over the button. The message can also be displayed in an `IlvHTMLPanel` positioned on the page. This is accomplished by setting the `messagePanel` property of the buttons.

Finally, we must update the `layoutPage` function to arrange the toolbar on the page and generate the HTML code for the toolbar:

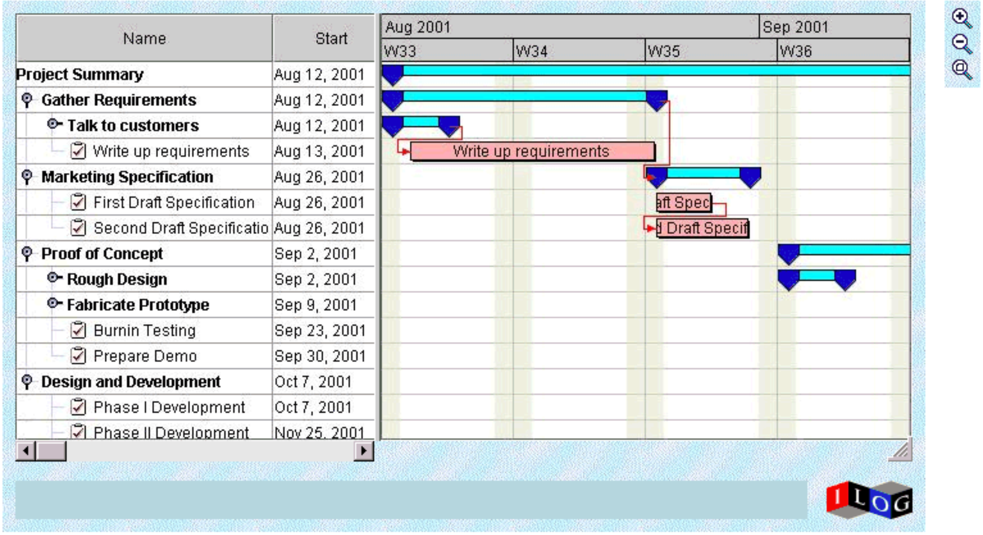
```
  var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop()+chart.getHeight()+15,
                           chart.getWidth()- logoPanel.getWidth()-15,
                           logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop()  - 10,
                              chart.getWidth()+ 20,
                              messagePanel.getTop()
                                + messagePanel.getHeight()
                                + 20
                                - chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()
                           + messagePanel.getWidth()
                           + 15,
                          messagePanel.getTop());
    toolbar.setLocation(backgroundPanel.getLeft()
                          + backgroundPanel.getWidth()
                          + 15,
                        backgroundPanel.getTop());
  }
  layoutPage(chartView);
  chartView.addSizeListener(layoutPage);
  toolbar.toHTML();
```

The complete body of the page is now:

```
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
```

```
// The Gantt chart servlet.
var servletName = "/gantt/GanttChartServlet";
// The background image.
var backgroundPattern = ilvImagePath + 'skybg.jpg';

// Position of the Gantt Chart.
var chartX = 25;
var chartY = 25;
var chartH = 350;
var chartW = 700;

var backgroundPanel = new IlvHTMLPanel('');
backgroundPanel.setBackgroundImage(backgroundPattern);
backgroundPanel.setBackgroundColor('#909090');
backgroundPanel.setVisible(true);

var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
chartView.setServletURL(servletName);
chartView.setResizable(true);
chartView.toHTML();

var messagePanel = new IlvHTMLPanel('');
messagePanel.setBackgroundColor('#B6D5DA');
messagePanel.setVisible(true);
chartView.setMessagePanel(messagePanel);

var logoPanel = new IlvImageView(0, 0, 67, 30,
                                 ilvImagePath+'ilog-small.gif');
logoPanel.toHTML();

var toolbar = new IlvToolBar(0, 0);
toolbar.setOrientation(IlvToolBar.VERTICAL);
toolbar.setBackgroundColor('#53537A');
toolbar.setBackgroundImage(backgroundPattern);

// Create the Zoom-In toolbar button.
var zoomInAction = function() {
  chartView.zoomIn();
};
var zoomInButton = new IlvButton(0, 0, 20, 20,
                                 ilvImagePath+'zoomin-up.gif',
                                 zoomInAction);
zoomInButton.setRolloverImage(ilvImagePath+'zoomin-sel.gif');
zoomInButton.setSelectedImage(ilvImagePath+'zoomin-dn.gif');
zoomInButton.setToolTipText("Zoom In");
zoomInButton.setMessage("Press to zoom in");
zoomInButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomInButton);

// Create the Zoom-Out toolbar button.
var zoomOutAction = function() {
  chartView.zoomOut();
  };
  var zoomOutButton = new IlvButton(0, 0, 20, 20,
                                    ilvImagePath+'zoomout-up.gif',
                                    zoomOutAction);
  zoomOutButton.setRolloverImage(ilvImagePath+'zoomout-sel.gif');
  zoomOutButton.setSelectedImage(ilvImagePath+'zoomout-dn.gif');
  zoomOutButton.setToolTipText("Zoom Out");
  zoomOutButton.setMessage("Press to zoom out");
  zoomOutButton.setMessagePanel(messagePanel);
  toolbar.addButton(zoomOutButton);

  // Create the Zoom-To-Fit toolbar button.
  var zoomFitAction = function() {
    chartView.zoomToFit();
  };
  var zoomFitButton = new IlvButton(0, 0, 20, 20,
                                    ilvImagePath+'zoomfit-up.gif',
                                    zoomFitAction);
  zoomFitButton.setRolloverImage(ilvImagePath+'zoomfit-sel.gif');
  zoomFitButton.setSelectedImage(ilvImagePath+'zoomfit-dn.gif');
  zoomFitButton.setToolTipText("Zoom To Fit");
  zoomFitButton.setMessage("Press to zoom to fit");
  zoomFitButton.setMessagePanel(messagePanel);
  toolbar.addButton(zoomFitButton);

  var layoutPage = function(chart) {
    messagePanel.setBounds(chart.getLeft(),
                           chart.getTop()+chart.getHeight()+15,
                           chart.getWidth()- logoPanel.getWidth()-15,
                           logoPanel.getHeight());
    backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop()  - 10,
                              chart.getWidth()+ 20,
                              messagePanel.getTop()
                                + messagePanel.getHeight()
                                + 20
                                - chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()
                           + messagePanel.getWidth()
                           + 15,
                          messagePanel.getTop());
    toolbar.setLocation(backgroundPanel.getLeft()
                          + backgroundPanel.getWidth()
                          + 15,
                        backgroundPanel.getTop());
  }
  layoutPage(chartView);
  chartView.addSizeListener(layoutPage);
  toolbar.toHTML();
</script>
</body>
```

The page now looks like this with our new vertical toolbar:



Name	Start
Project Summary	Aug 12, 2001
Gather Requirements	Aug 12, 2001
Talk to customers	Aug 12, 2001
☑ Write up requirements	Aug 13, 2001
Marketing Specification	Aug 26, 2001
☑ First Draft Specification	Aug 26, 2001
☑ Second Draft Specificatio	Aug 26, 2001
Proof of Concept	Sep 2, 2001
Rough Design	Sep 2, 2001
Fabricate Prototype	Sep 9, 2001
☑ Burnin Testing	Sep 23, 2001
☑ Prepare Demo	Sep 30, 2001
Design and Development	Oct 7, 2001
☑ Phase I Development	Oct 7, 2001
☑ Phase II Development	Nov 25, 2001

IlvGanttSheetScrollInteractor

Until now, we have added components and buttons to our page. We will now add an interactor that allows direct interaction with the image. The `IlvGanttSheetScrollInteractor` allows the user to interactively scroll and pan the image of the Gantt sheet. The `IlvGanttView` component is composed of 2 child views: an `IlvGanttTableView` that displays the image of the table on the left side of the splitter, and an `IlvGanttSheetView` that displays the image of the Gantt sheet on the right side of the splitter. This is shown in Figure 9.5 on page 142. You set interactors separately on the Gantt table and sheet views. The code to create an `IlvGanttSheetScrollInteractor` and set it on the `IlvGanttSheetView` is very simple:

```
var sheetView = chartView.getSheetView();
var sheetScrollInteractor = new IlvGanttSheetScrollInteractor();
sheetView.setInteractor(sheetScrollInteractor);
```

In order to use the `IlvGanttSheetScrollInteractor` class, we must include the JavaScript files that define the class and its base class, `IlvInteractor`:

```
<script TYPE="text/javascript"  src="script/IlvInteractor.js"></script>
<script TYPE="text/javascript"
src="script/IlvGanttSheetScrollInteractor.js"></script>
```

IlvRowExpandCollapseInteractor

The `IlvRowExpandCollapseInteractor` can be set on both an `IlvGanttTableView` and an `IlvGanttSheetView`. It allows the user to click on rows in the image to expand and collapse them. We will set the interactor on the Gantt table view. First, we must include the JavaScript file that defines the class:

```
<script TYPE="text/javascript"
src="script/IlvRowExpandCollapseInteractor.js"></script>
```

Then we set the interactor on the `IlvGanttTableView`:

```
var tableView = chartView.getTableView();
var tableToggleRowInteractor =
  new IlvRowExpandCollapseInteractor('toggleRow');
tableView.setInteractor(tableToggleRowInteractor);
```

The string `‘toggleRow’` refers to a named action on the server side that this interactor invokes. On the server side, this action is registered with the servlet support object in the file:

`<installdir>/demos/gantt/servlet/src/GanttChartServlet.java`

in the `createServletSupport` method:

```
protected IlvGanttServletSupport createServletSupport()
{
  IlvGanttServletSupport support = new ServletSupport();
  support.addServerAction("toggleRow", new IlvRowExpandCollapseAction());
  return support;
}
```

These types of interactors and how to use them are described in detail in *Adding Client/ Server Interactions* on page 163.

IlvInteractorButton

The `IlvInteractorButton` class is a subclass of `IlvButton` that installs an interactor on a view. If multiple interactor buttons are defined for the same view, they will behave like radio buttons. When an interactor button is pressed, it installs its interactor and remains pressed until another button installs a different interactor.

In our example, we will define an `IlvRowExpandCollapseInteractor` for the Gantt sheet view. We will then add 2 interactor buttons to the toolbar that toggle between the scroll interactor and the new row interactor.

First, we include the JavaScript file that defines the `IlvInteractorButton` class:

```
<script TYPE="text/javascript"  src="script/IlvInteractorButton.js"></script>
```

Then, we create the new interactor for the Gantt sheet view:

```
var sheetView = chartView.getSheetView();
var sheetScrollInteractor = new IlvGanttSheetScrollInteractor();
sheetView.setInteractor(sheetScrollInteractor);
var sheetToggleRowInteractor =
  new IlvRowExpandCollapseInteractor('toggleRow');
```

Finally, we create the interactor buttons and add them to the toolbar:

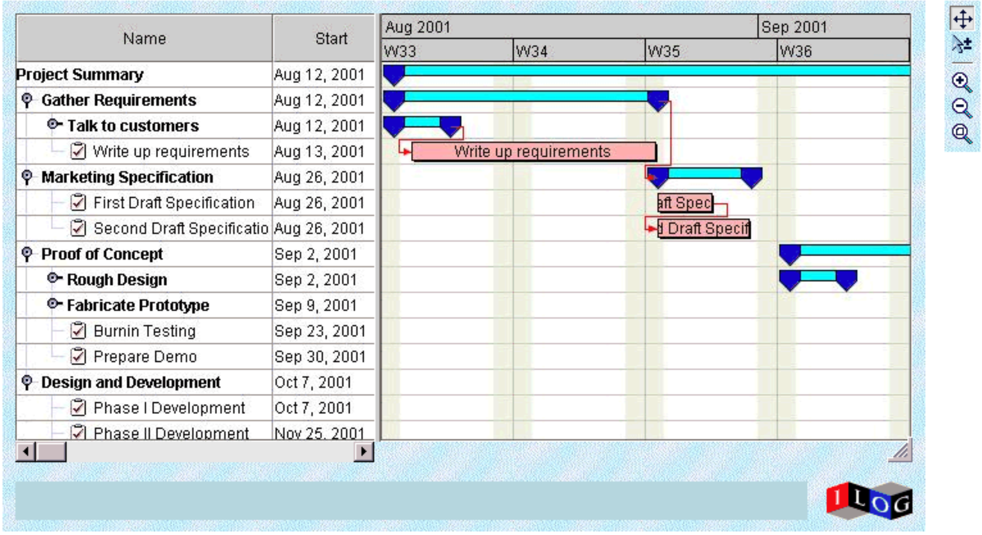
```
// Create the scroll toolbar button for the Gantt sheet.
var sheetPanButton =
  new IlvInteractorButton(0, 0, 20, 20,
                          ilvImagePath+'move-up.gif',
                          sheetScrollInteractor,
                          sheetView);
sheetPanButton.setRolloverImage(ilvImagePath+'move-sel.gif');
sheetPanButton.setSelectedImage(ilvImagePath+'move-dn.gif');
sheetPanButton.setToolTipText("Pan Gantt Sheet");
sheetPanButton.setMessage("Scroll and pan the Gantt sheet");
sheetPanButton.setMessagePanel(messagePanel);
toolbar.addButton(sheetPanButton);

// Create the toggle row toolbar button for the Gantt sheet.
var sheetToggleRowButton =
  new IlvInteractorButton(0, 0, 20, 20,
                          ilvImagePath+'bluearrow-plusminus-up.gif',
                          sheetToggleRowInteractor,
                          sheetView);
sheetToggleRowButton.setRolloverImage(ilvImagePath+'bluearrow-
                                                      plusminus-sel.gif');
sheetToggleRowButton.setSelectedImage(ilvImagePath+'bluearrow-
                                                      plusminus-dn.gif');
sheetToggleRowButton.setToolTipText("Expand/Collapse Gantt Sheet Rows");
sheetToggleRowButton.setMessage("Expand/collapse rows in the Gantt sheet");
sheetToggleRowButton.setMessagePanel(messagePanel);
toolbar.addButton(sheetToggleRowButton);
```

The body of the page is now:

```
<body onload="init()" onunload=”ilvDispose()”
      onresize="handleResize()" bgcolor="#ffffff">
<script TYPE="text/javascript">
  // The Gantt chart servlet.
  var servletName = "/gantt/GanttChartServlet";
  // The background image.
  var backgroundPattern = ilvImagePath + 'skybg.jpg';

  // Position of the Gantt Chart.
  var chartX = 25;
  var chartY = 25;
  var chartH = 350;
  var chartW = 700;

  var backgroundPanel = new IlvHTMLPanel('');
  backgroundPanel.setBackgroundImage(backgroundPattern);
  backgroundPanel.setBackgroundColor('#909090');
  backgroundPanel.setVisible(true);

  var chartView = new IlvGanttView(chartX, chartY, chartW, chartH);
  chartView.setServletURL(servletName);
  chartView.setResizable(true);
  chartView.toHTML();

  var sheetView = chartView.getSheetView();
  var sheetScrollInteractor = new IlvGanttSheetScrollInteractor();
  sheetView.setInteractor(sheetScrollInteractor);
  var sheetToggleRowInteractor =
    new IlvRowExpandCollapseInteractor('toggleRow');

  var tableView = chartView.getTableView();
  var tableToggleRowInteractor =
    new IlvRowExpandCollapseInteractor('toggleRow');
  tableView.setInteractor(tableToggleRowInteractor);

  var messagePanel = new IlvHTMLPanel('');
  messagePanel.setBackgroundColor('#B6D5DA');
  messagePanel.setVisible(true);
  chartView.setMessagePanel(messagePanel);

  var logoPanel = new IlvImageView(0, 0, 67, 30,
                                   ilvImagePath+'ilog-small.gif');
  logoPanel.toHTML();

  var toolbar = new IlvToolBar(0, 0);
  toolbar.setOrientation(IlvToolBar.VERTICAL);
  toolbar.setBackgroundColor('#53537A');
  toolbar.setBackgroundImage(backgroundPattern);

  // Create the scroll toolbar button for the Gantt sheet.
  var sheetPanButton =
    new IlvInteractorButton(0, 0, 20, 20,
                            ilvImagePath+'move-up.gif',
                            sheetScrollInteractor,
                            sheetView);
  sheetPanButton.setRolloverImage(ilvImagePath+'move-sel.gif');
  sheetPanButton.setSelectedImage(ilvImagePath+'move-dn.gif');
  sheetPanButton.setToolTipText("Pan Gantt Sheet");
  sheetPanButton.setMessage("Scroll and pan the Gantt sheet");
  sheetPanButton.setMessagePanel(messagePanel);
  toolbar.addButton(sheetPanButton);

  // Create the toggle row toolbar button for the Gantt sheet.
  var sheetToggleRowButton =
    new IlvInteractorButton(0, 0, 20, 20,
                            ilvImagePath+'bluearrow-plusminus-up.gif',
                            sheetToggleRowInteractor,
                            sheetView);
  sheetToggleRowButton.setRolloverImage(ilvImagePath+'bluearrow-
                                              plusminus-sel.gif');
  sheetToggleRowButton.setSelectedImage(ilvImagePath+'bluearrow-
                                              plusminus-dn.gif');
  sheetToggleRowButton.setToolTipText("Expand/Collapse Gantt Sheet Rows");
  sheetToggleRowButton.setMessage("Expand/collapse rows in the Gantt sheet");
  sheetToggleRowButton.setMessagePanel(messagePanel);
  toolbar.addButton(sheetToggleRowButton);

  // Create a toolbar separator.
```

```
var separator = new IlvButton(0, 0, 20, 10,
                              ilvImagePath+'horzsep.gif');
toolbar.addButton(separator);

// Create the Zoom-In toolbar button.
var zoomInAction = function() {
  chartView.zoomIn();
};
var zoomInButton = new IlvButton(0, 0, 20, 20,
                                 ilvImagePath+'zoomin-up.gif',
                                 zoomInAction);
zoomInButton.setRolloverImage(ilvImagePath+'zoomin-sel.gif');
zoomInButton.setSelectedImage(ilvImagePath+'zoomin-dn.gif');
zoomInButton.setToolTipText("Zoom In");
zoomInButton.setMessage("Press to zoom in");
zoomInButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomInButton);

// Create the Zoom-Out toolbar button.
var zoomOutAction = function() {
  chartView.zoomOut();
};
var zoomOutButton = new IlvButton(0, 0, 20, 20,
                                  ilvImagePath+'zoomout-up.gif',
                                  zoomOutAction);
zoomOutButton.setRolloverImage(ilvImagePath+'zoomout-sel.gif');
zoomOutButton.setSelectedImage(ilvImagePath+'zoomout-dn.gif');
zoomOutButton.setToolTipText("Zoom Out");
zoomOutButton.setMessage("Press to zoom out");
zoomOutButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomOutButton);

// Create the Zoom-To-Fit toolbar button.
var zoomFitAction = function() {
  chartView.zoomToFit();
};
var zoomFitButton = new IlvButton(0, 0, 20, 20,
                                  ilvImagePath+'zoomfit-up.gif',
                                  zoomFitAction);
zoomFitButton.setRolloverImage(ilvImagePath+'zoomfit-sel.gif');
zoomFitButton.setSelectedImage(ilvImagePath+'zoomfit-dn.gif');
zoomFitButton.setToolTipText("Zoom To Fit");
zoomFitButton.setMessage("Press to zoom to fit");
zoomFitButton.setMessagePanel(messagePanel);
toolbar.addButton(zoomFitButton);

var layoutPage = function(chart) {
  messagePanel.setBounds(chart.getLeft(),
                         chart.getTop()+chart.getHeight()+15,
                         chart.getWidth()- logoPanel.getWidth()-15,
                         logoPanel.getHeight());
  backgroundPanel.setBounds(chart.getLeft() - 10,
                              chart.getTop()  - 10,
                              chart.getWidth()+ 20,
                              messagePanel.getTop()
                                + messagePanel.getHeight()
                                + 20
                                - chart.getTop());
    logoPanel.setLocation(messagePanel.getLeft()
                           + messagePanel.getWidth()
                           + 15,
                          messagePanel.getTop());
    toolbar.setLocation(backgroundPanel.getLeft()
                          + backgroundPanel.getWidth()
                          + 15,
                        backgroundPanel.getTop());
  }
  layoutPage(chartView);
  chartView.addSizeListener(layoutPage);

  toolbar.toHTML();

</script>
</body>
```

This results in the following page:



Name	Start
Project Summary	Aug 12, 2001
Gather Requirements	Aug 12, 2001
Talk to customers	Aug 12, 2001
Write up requirements	Aug 13, 2001
Marketing Specification	Aug 26, 2001
First Draft Specification	Aug 26, 2001
Second Draft Specificatio	Aug 26, 2001
Proof of Concept	Sep 2, 2001
Rough Design	Sep 2, 2001
Fabricate Prototype	Sep 9, 2001
Burnin Testing	Sep 23, 2001
Prepare Demo	Sep 30, 2001
Design and Development	Oct 7, 2001
Phase I Development	Oct 7, 2001
Phase II Development	Nov 25, 2001

You can now click the first two toolbar buttons to select which interactor is installed on the Gantt sheet view.

Adding Client/Server Interactions

The Gantt thin-client support gives you a simplified way to define new actions that should take place on the server side. For example, suppose you want to allow the user to change the name of an activity that appears on the generated image. Part of this action, clicking the image to select the activity, must be done on the client side. Changing the name of the activity in the Gantt data model must be done on the server side before a new image is generated. The notion of a “server-side action” exists to perform such behavior. An action is defined by a name and a set of string parameters.

The Client Side

In a dynamic HTML client, you tell the server to perform an action using the `performAction` method of the `IlvGanttTableView` or `IlvGanttSheetView` JavaScript component. Here is an example that asks the server side to execute the action “setName” with coordinate and string parameters, assuming that `view` is an `IlvGanttSheetView`:

```
var x = 100;
var y = 50;
var params = new Array();
params[0]=x;
params[1]=y;
params[2]=”New Activity Name”;
view.performAction(“setName”, params);
```

The `performAction` method will ask the server for a new image. In the image request, additional parameters are added so that the server side can execute the action. Thus, the `performAction` call results in only one client/server round-trip.

> ***Note:*** *In section Actions that Modify Chart Capabilities, we will discuss server actions that require 2 client/server round trips.*

Creating a Custom Interactor

Now let us explore how to create a custom client-side interactor that will allow the user to click on an activity graphic in the `IlvGanttSheetView` and ask the server side to execute the “setName” action. We start by defining our new `ActivityNameInteractor` class as a subclass of `IlvInteractor`. `IlvInteractor` is the base class for all client-side interactors that operate on JavaScript view components:

```
function ActivityNameInteractor() {
  this.superConstructor();
}

ActivityNameInteractor.prototype = new IlvInteractor();
ActivityNameInteractor.prototype.setClassName("ActivityNameInteractor");
```

Then we override the mouseDown method to convert the mouse coordinates to be relative to the Gantt sheet and request the server side to perform the “setName” action:

```
function ActivityNameInteractor() {
  this.superConstructor();
}

ActivityNameInteractor.prototype = new IlvInteractor();
ActivityNameInteractor.prototype.setClassName("ActivityNameInteractor");

ActivityNameInteractor.prototype.mouseDown = function(e) {
  // The JavaScript view component is always stored in the view
  // instance variable of the interactor.
  var view = this.view;
  // The Y position of the mouse event is relative to the top of the DHTML
  // IlvGanttSheetView. The action needs a Y position relative to the top of
  // the Gantt sheet, ignoring the time scale.
  var actionYPos = e.mouseY - view.cap_tableHeaderHeight;
  if (actionYPos < 0) // Mouse is in timescale, so ignore
    return;
  // Create parameters for the setName action and send it to the server side.
  var params = new Array();
  params[0]=e.mouseX;
  params[1]=actionYPos;
  params[2]="New Activity Name";
  view.performAction("setName", params);
}
```

Notice how we have used the cap_tableHeaderHeight instance variable of the IlvGanttSheetView to subtract out the height of the time scale. This variable is one of several that are initialized when the server side sends capabilities information to the view. The full details of the capabilities request are described in the section *The Capabilities Request* on page 169. You can find details on the other instance variables that the view initializes from the capabilities information in the reference manual for the IlvGanttComponentView.getCapabilities method. IlvGanttComponentView is the superclass of IlvGanttSheetView.

We can make our new interactor a little bit safer to use by overriding the setView method. This method is inherited from IlvInteractor and is invoked automatically when an interactor is set on or removed from a view. By overriding this method, we can verify that the view is indeed an instance of IlvGanttSheetView:

```
function ActivityNameInteractor() {
  this.superConstructor();
}

ActivityNameInteractor.prototype = new IlvInteractor();
ActivityNameInteractor.prototype.setClassName("ActivityNameInteractor");

ActivityNameInteractor.prototype.mouseDown = function(e) {
  // The JavaScript view component is always stored in the view
  // instance variable of the interactor.
  var view = this.view;
  // The Y position of the mouse event is relative to the top of the DHTML
  // IlvGanttSheetView. The action needs a Y position relative to the top of
  // the Gantt sheet, ignoring the time scale.
  var actionYPos = e.mouseY - view.cap_tableHeaderHeight;
  if (actionYPos < 0) // Mouse is in timescale
    return;
  // Create parameters for the setName action and send it to the server side.
  var params = new Array();
  params[0]=e.mouseX;
  params[1]=actionYPos;
  params[2]="New Activity Name";
  view.performAction("setName", params);
}

ActivityNameInteractor.prototype.setView = function(view) {
  if (view != null && !view.instanceOf(IlvGanttSheetView)) {
    alert("ActivityNameInteractor can only be set on an IlvGanttSheetView");
  }
}
```

The Server Side

On the server side, we need to detect that an action was requested and execute the action before the image is generated and sent back to the client. This is done by implementing the `ilog.views.gantt.servlet.IlvServerAction` interface. To listen for an action request from the client and execute the action on the server side, you register the action with your instance of `IlvGanttServletSupport` using the `addServerAction` method.

For the “setName” action, we would add the following lines of code in the `createServletSupport` method of the example `GanttChartServlet`:

```
  protected IlvGanttServletSupport createServletSupport()
  {
    IlvGanttServletSupport support = new ServletSupport();
    support.addServerAction("setName", new IlvServerAction()
    {
      public void actionPerformed(ServerActionEvent event)
        throws ServletException;
      {
        int x = event.getIntParameter(0);
        int y = event.getIntParameter(1);
        String name = event.getStringParameter(2);
        IlvHierarchyChart chart = event.getChart();
        IlvGraphic graphic = chart.getGanttSheet().getGraphic(new Point(x, y));
        if (graphic instanceof IlvActivityGraphic) {
          IlvActivity activity = ((IlvActivityGraphic)graphic).getActivity();
          activity.setName(name);
        }
      }
    });
    return support;
  }
}});
```

Actions that Modify Chart Capabilities

The DHTML client maintains certain state information about the server-side `IlvHierarchyChart` that it is displaying. We call this basic set of state information “capabilities”. The capabilities information is sent by the server to the client when the client side requests it. When the client side requests an updated set of capabilities data, the server also sends an updated image to the client. The capabilities data includes the number of rows currently visible, the height of the rows, and other basic state information that allows the client to intelligently scroll and manipulate the chart images. The full details of the capabilities request and chart data are described in section *The Capabilities Request* on page 169.

Some server actions requested by the client may modify the capabilities state information for the chart. In this case, the client must be able to request that the server perform the action, send updated capabilities information to the client, and then send an updated image to the client. For example, suppose you want to allow the user to click on a row in the table and toggle the row’s expand/collapse state. As in the previous example, toggling the row must be performed on the server side. However, expanding and collapsing rows in the server side chart modifies the capabilities data on how many rows are visible. The client side must be updated with the new capabilities so that client-side scrolling and row hit testing can be performed correctly.

We again use the `performAction` method of the `IlvGanttTableView` or `IlvGanttSheetView` DHTML components to request this type of server action. This time however, we set the optional third parameter, `updateAll`, to `true`. This requests the server to send updated capabilities to the client, in addition to performing the action and sending an updated image. Here is some example JavaScript code that asks the server side to execute the action “toggleRow” with a y-coordinate relative to the first row in the table, assuming that `view` is an `IlvGanttTableView`:

```
var mouseY = .. mouse pos relative to top of IlvGanttTableView ..
// Take table header into account.
mouseY = mouseY - view.cap_tableHeaderHeight;
// If mouse is in table header, nothing to do
if (mouseY < 0)
  return;
// Take vertical scroll position into account.
mouseY = mouseY + view.getVerticalScrollPosition();
var params = new Array();
params[0] = tableYPos;
view.performAction(“toggleRow”, params, true);
```

As in the previous example, you register the action on the server side using the `IlvGanttServletSupport.addServerAction` method. For the “toggleRow” action, we would add the following lines of code in the `createServletSupport` method of the example `GanttChartServlet`:

```
  protected IlvGanttServletSupport createServletSupport()
  {
    IlvGanttServletSupport support = new ServletSupport();
    support.addServerAction("toggleRow", new IlvServerAction()
    {
      public void actionPerformed(ServerActionEvent event)
        throws ServletException;
      {
        int yPos = event.getIntParameter(0);
        IlvHierarchyChart chart = event.getChart();
        IlvHierarchyNode row = chart.getVisibleRowAtPosition(yPos);
        if (row == null)
          return;
        if (chart.isRowExpanded(row))
          chart.collapseRow(row);
        else
          chart.expandRow(row);
      }
    });
    return support;
  }
}});
```

The IlvGanttServlet and IlvGanttServletSupport Classes

The server side of a thin-client Gantt web application consists of creating a servlet that can produce an image and send it to the client. The Gantt thin-client support provides a predefined servlet to achieve this task. The predefined servlet class is named `IlvGanttServlet`. This class, which can be found in the package `ilog.views.gantt.servlet`, is an abstract subclass of the `HTTPServlet` class from the Java servlet API.

Using the `IlvGanttServlet` class is an easy way to create a servlet, but it has one main drawback. You cannot use it to add support for the Gantt thin-client protocol to an existing servlet. This is the purpose of the `IlvGanttServletSupport` class. The `IlvGanttServletSupport` class implements all the Gantt thin-client server-side functionality. In fact, the `IlvGanttServlet` class is just a basic wrapper around an instance of `IlvGanttServletSupport`. The `doGet` method of `IlvGanttServlet` simply calls the `handleRequest` method of its `IlvGanttServletSupport` instance.

In the same way, you can integrate an instance of `IlvGanttServletSupport` into your own servlet to handle the requests coming from the Gantt client side. In our Gantt Servlet example, the code of the servlet can be rewritten using the `IlvGanttServletSupport` class as follows:

```
import ilog.views.gantt.*;
import ilog.views.gantt.servlet.*;
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class GanttChartServlet extends HttpServlet
{
  private IlvGanttServletSupport support;

  public void init(ServletConfig config)
    throws ServletException
  {
    super.init(config);
    support = new ServletSupport();
  }

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
    throws IOException, ServletException
  {
    if (!support.handleRequest(request, response))
      throw new ServletException("Unrecognized request");
  }

  public void doPost(HttpServletRequest request,
                     HttpServletResponse response)
       throws IOException, ServletException
  {
    doGet(request, response);
  }

}

class ServletSupport extends IlvGanttServletSupport
{
  private IlvHierarchyChart _chart;

  public ServletSupport()
  {
    _chart = createChart();
  }

  private IlvHierarchyChart createChart()
  {
    IlvHierarchyChart chart = new IlvGanttChart();
    IlvGanttModel ganttModel = new SimpleProjectDataModel();
    return chart;
  }

  public IlvHierarchyChart getChart(HttpServletRequest request,
                                    IlvServletRequestParameters params)
    throws ServletException
  {
    return _chart;
  }

}
```

In this code we have created a new servlet class, `GanttChartServlet`, that is derived directly from the `HttpServlet` class. The `doGet` method passes the requests to an instance of the `IlvGanttServletSupport` class for handling.

The Servlet Parameters

The Gantt servlet support can respond to two different types of HTTP requests, the “image” request and the “capabilities” request. The image request will return an image from the Gantt or Schedule chart. The capabilities request will return information to the client, such as the number of rows visible in the chart, the height of the rows, and the minimum and maximum times for horizontally scrolling the Gantt sheet. This information allows the client to know the capabilities of the chart in order to intelligently scroll and manipulate the chart images. When developing the client side of your application, you will use the DHTML scripts provided by the Gantt thin-client support. The scripts will create the HTTP request for you, so you do not really need to write the HTTP request yourself.

The Image Request

The image request produces an image from the chart. Here is an example of a request for the image of the table portion of a chart, assuming that `myservlet` is the name of the servlet:

```
http://host/servlets/myservlet?request=image
  &comp=table
  &width=300
  &height=250
```

And here is an example for the image of the Gantt sheet portion of a chart:

```
http://host/servlets/myservlet?request=image
  &comp=sheet
  &width=500
  &height=250
  &startTime=2001,0,1
  &endTime=2001,5,30
```

The Gantt sheet will display the time period from January 1, 2001 to June 30, 2001. A detailed listing of the image request parameters can be found in the reference manual for the `IlvGanttServletSupport` class.

The Capabilities Request

The capabilities request tells the servlet to return the capabilities information about the chart to the client. The capabilities request has the following syntax:

```
http://host/servlets/myservlet?request=capabilities
```

```
  &format=(html|octet-stream)
  [ &onload= <a string> ]
```

The `format` parameter tells the servlet which format should be used to return the capabilities information. Two formats are supported, HTML or Octet stream.

The HTML format is used when the client is a Dynamic HTML client. In this case, the result is an empty HTML page that contains some JavaScript code. The JavaScript code is executed on the client side, and some information variables are then available.

The octet-stream format is used when the client is a Java applet. In this case, the result is a stream of octets. The data is produced using a `java.io.DataOutput` and can be read using a `java.io.DataInput`.

> ***Note:*** *The Gantt thin-client support does not provide a predefined Java applet thin client.*

Full details on the capabilities request and the information returned by the server can be found in the reference manual for the `IlvGanttServletSupport` class. Details on how the client-side DHTML components save and use the capabilities information can be found in the reference manual for the `IlvGanttComponentView.getCapabilities` method.

Multiple Sessions

The Gantt Servlet example presented a very simple example that creates a single Gantt chart for the servlet. This means that all calls to the servlet (that is, all clients) are looking at the same chart and the same data model. In some applications, you may want to have a chart and/or a separate data model for each client. In this case, you might use the notion of *HTTP sessions*. You can then create a chart and/or a data model and store them as parameters of the session.

Here we take our Gantt Servlet example and modify it slightly so that each client has its own chart that is viewing a common data model. This way, each user can toggle rows to expand and collapse without affecting the charts viewed by the other clients. We use an instance of the `IlvGanttSessionAttribute` class to store the chart as an attribute of the HTTP session. This class handles the details of properly disposing server-side GUI components when the user session expires. Our updated `IlvGanttServletSupport` implementation now looks like this:

```
class ServletSupport extends IlvGanttServletSupport
{
  private IlvGanttModel _model;

  private IlvHierarchyChart createChart()
  {
    synchronized(this) {
      if (_model == null)
        _model = new SimpleProjectDataModel();
    }
    IlvHierarchyChart chart = new IlvGanttChart();
    chart.setGanttModel(_model);
    chart.getGanttSheet().setVerticalGrid(new WeekendGrid());
    ... more chart customizations ...
    return chart;
  }

  /**
   * Returns the chart used for the specified request.
   * @param request The current HTTP request.
   * @param params The parameters parsed from the request.
   */
  public IlvHierarchyChart getChart(HttpServletRequest request,
                                    IlvServletRequestParameters params)
    throws ServletException
  {
    IlvHierarchyChart chart = null;
    HttpSession session = request.getSession();
    if (session.isNew()) {
      chart = createChart();
      IlvGanttSessionAttribute chartProxy =
        new IlvGanttSessionAttribute(chart);
      session.setAttribute("IlvHierarchyChart", chartProxy);
    } else {
      IlvGanttSessionAttribute chartProxy =
        (IlvGanttSessionAttribute)session.getAttribute("IlvHierarchyChart");
      if (chartProxy != null)
        chart = chartProxy.getChart();
    }
    if (chart == null)
      throw new ServletException("session problem");
    return chart;
  }
}
```

Note: *If you store the chart directly as an attribute of the HTTP session, it will not be properly garbage collected when the session expires. You must wrapper the chart in an instance of* `IlvGanttSessionAttribute` *to ensure that the chart is properly disposed.*

A

Document Type Definition for SDXL

```
<!--
    This is the DTD for 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 >

<!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 >
```

Glossary

about-to-change event

When a property value is about to be notified to the model, it is advisable to notify interested listeners as well so that they have an opportunity to constrain or even to veto the proposed new property value.

activity

A task or occupation that is planned to be completed. A parent activity can be broken down into several child activities. *See also* child activity, From activity, leaf activity, parent activity, root activity, To activity.

activity graphic

In a Gantt chart, an instance of the class `IlvActivityGraphic` used to represent the associated activity on a row of the Gantt sheet.

activity renderer

In a Gantt chart, an object that implements the `IlvActivityRenderer` interface to draw activity graphics to represent activities in the Gantt sheet.

bounded mode

In the Gantt sheet, an operation mode of the time scale whereby the scroll bar is limited to the specified time interval. *See also* unbounded mode.

constraint

A condition set between two activities whereby one activity depends on the other. Constraints are represented by an arrowed polyline object. *See also* end to end, end to start, From activity, start to end, start to start, To activity.

constraint graphic

In a Gantt chart, an instance of the class `IlvConstraintGraphic` used to represent a constraint between two activities.

data model

The scheduling information you want to represent as a Gantt chart. One major feature of the ILOG JViews Gantt Chart module is that the data model is separated from the visualization part.

data nodes

Another way of referring to activities in a Gantt chart or to resources in a Schedule chart.

end time

An activity property that determines the date and time at which the activity is planned to finish.

end to end

A type of constraint whereby the end of the To activity depends on the end of the From activity. *See also* From activity, To activity.

end to start

A type of constraint whereby the end of the To activity depends on the start of the From activity. *See also* From activity, To activity.

From activity

The source activity of a constraint, that is, the activity whose start or end controls the start or end of another activity as the result of the constraint. *See also* activity, To activity.

Gantt chart

A type of schedule diagram where data from a table is displayed as horizontal bars along a time scale.

Gantt sheet

The right-hand part of a Gantt chart or Schedule chart, where activities and constraints (on Gantt charts) or reservations (on Schedule charts) are represented graphically. `IlvGanttSheet` objects are instances of a subclass of `IlvManagerView`.

leaf activity

An activity with no child activity. *See also* activity, child activity.

leaf resource

A resource with no child resource. *See also* resource, child resource.

load-on-demand

A mechanism whereby data is loaded based on whether it is visible.

milestone

An activity whose duration is null, that is, whose start time and end time are simultaneous.

parent activity

An activity with at least one child activity. *See also* activity, child activity.

parent resource

A resource with at least one child resource. *See also* resource, child resource.

print framework

A package of classes helping you to print and preview data.

resource

Means that enable an activity to be completed: persons, premises, equipment, and so forth. *See also* child resource, leaf resource, parent resource.

reservation

Booking usage of a resource for the duration of an activity. In the terminology of the Gantt Chart module, a reservation represents the allocation of *one* resource to *one* activity.

reservation graphic

In a Schedule chart, an instance of the class `IlvReservationGraphic` (itself a subclass of `IlvActivityGraphic`) to represent reservation on the rows of the Gantt sheet. *See also* activity graphic.

root activity The top level of the hierarchical tree of activities.

root resource The top level of the hierarchical tree of resources.

separable model architecture A Swing variant of the traditional MVC design of user-interface objects, where the view and controller parts are bundled together.

start time An activity property that determines the date and time at which the activity is planned to begin.

start to end A type of constraint whereby the start of the To activity depends on the end of the From activity. *See also* From activity, To activity.

start to start A type of constraint whereby the start of the To activity depends on the start of the From activity. *See also* From activity, To activity.

time scale A zoomable range of rows at the top of the Gantt sheet where the time divisions are represented using various time unit values.

To activity The target activity of a constraint, that is, the activity whose start or end depends on the start or end of another activity as the result of the constraint. *See also* activity, From activity.

unbounded mode In the Gantt sheet, the default operation mode of the time scale whereby there is no upper or lower limit to the horizontal scrolling. *See also* bounded mode.

Index