JClass HiGrid™

Programmer's Guide

Version 6.3
for Java 2 (JDK 1.3.1 and higher)

The Unique RAD Outline-grid
for Hierarchical Dynamic Data

8001 Irvine Center Drive
Irvine, CA 92618
949-754-8000
www.quest.com

April 2004

RefNo: PRGDE-JCHGD/630-04/2004

Warranty

The information contained in this document is subject to change without notice. Quest Software makes no warranty of any kind with respect to this information. QUEST SOFTWARE SPECIFICALLY DISCLAIMS THE IMPLIED WARRANTY OF THE MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Quest Software shall not be liable for any direct, indirect, incidental, consequential, or other damage alleged in connection with the furnishing or use of this information.

Trademarks

JClass, JClass Chart, JClass Chart 3D, JClass DataSource, JClass Elements, JClass Field, JClass HiGrid, JClass JarMaster, JClass LiveTable, JClass PageLayout, JClass ServerChart, JClass ServerReport, JClass DesktopViews, and JClass ServerViews are trademarks of Quest Software, Inc. Other trademarks and registered trademarks used in this guide are property of their respective owners.

World Headquarters
8001 Irvine Center Drive
Irvine, CA 92618
www.quest.com
e-mail: info@quest.com
U.S. and Canada: 949.754.8000

Please refer to our Web site for regional and international office information.

This product includes software developed by the Apache Software Foundation http://www.apache.org/.

The JPEG Encoder and its associated classes are Copyright © 1998, James R. Weeks and BioElectroMech. This product is based in part on the work of the Independent JPEG Group.

> Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:
>
> 1. Redistributions of source code must retain the above copyright notice, this list of conditions, all files included with the source code, and the following disclaimer.
> 2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
>
> THIS SOFTWARE IS PROVIDED BY THE AUTHOR AND CONTRIBUTORS “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE AUTHOR OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL,

EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

This product includes software developed by the JDOM Project (http://www.jdom.org/). Copyright © 2000-2002 Brett McLaughlin & Jason Hunter, all rights reserved. Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions, and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions, and the disclaimer that follows these conditions in the documentation and/or other materials provided with the distribution.
3. The name "JDOM" must not be used to endorse or promote products derived from this software without prior written permission. For written permission, please contact license@jdom.org.
4. Products derived from this software may not be called "JDOM", nor may "JDOM" appear in their name, without prior written permission from the JDOM Project Management (pm@jdom.org).

THIS SOFTWARE IS PROVIDED “AS IS” AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE JDOM AUTHORS OR THE PROJECT CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

Table of Contents

Preface

Introducing JClass HiGrid

Database applications that require an expandable grid to display the graphic representation of master-detail relationships have a new Java tool: JClass HiGrid. Its collapsible grid is an excellent tool for allowing top-down exploration and navigation to detail levels of arbitrary depth. A versatile data binding tool, called JClass DataSource, is the engine that drives all data retrieval for the grid. You can use it for more generalized data binding tasks spanning the whole set of JClass products. In addition, JClass DataSource provides a set of data bound AWT and Swing components.

Because its design is based on the Model-View-Controller (MVC) paradigm, JClass HiGrid is conceptually straightforward, being both the view and the controller, while JClass DataSource functions as the model. Ultimately, a grid is a collection of cells. JClass HiGrid's cell classes extend the capabilities in `com.klg.jclass.cell` to support rendering, editing, and some elementary validating operations. As far as cells are concerned, JClass DataSource remains the model, while a cell renderer is the view and a cell editor is the controller. This design decouples the way that a cell is edited from the way that it is displayed, allowing such flexible scenarios as having a cell containing a Boolean quantity display it as an icon, while its editor may employ a checkbox, a String value, or combo box with true/false choices. Extended editors are available for more specialized items. For instance, a date field is edited with a drop-down calendar.

JClass DesktopViews makes the full range of data bound components available, where JClass DataSource is used as the data binding layer for tables, charts, and fields requiring specialized validation. It may be used with any database for which a JDBC (Java Database Connectivity) driver exists or, through a JDBC-ODBC bridge, for which an ODBC driver exists. Its primary purpose is to bind to databases that have JDBC drivers, but it can also be used with non-database sources such as text files, or it can act as a supplier of information extracted from an "in-memory database" whose values are created dynamically at run-time. If used within an Integrated Development Environment (IDE), JClass HiGrid's Bean makes it particularly easy to bind to a data source, issue SQL statements, and retrieve and display the resulting data tables.

JClass DataSource manages circumstances in which the underlying structure of the data design is a tree. Because Java does not define a `Tree` data structure, a completely general tree model is defined within JClass DataSource itself[1].

1. The `TreeModel` interface is used for this purpose. The more specific interface is `DataModel`, which encapsulates two types of trees, the `MetaDataTree` and the `DataModelTree`. Each of these has its own interface.

Feature Overview

JClass HiGrid and JClass DataSource are JavaBeans that facilitate the presentation of data extracted from a database or elsewhere in a hierarchical, or master-detail, form. Their full-featured customizers can be used in IDEs to quickly develop a data retrieval application. JClass HiGrid's custom property editor exhibits a highly interactive interface that allows end-users to perform all the common data operations without extensive coding. Moreover, for those whose application may demand more in-depth programming, the products' APIs contain a number of helper methods designed to make common tasks easy to accomplish.

You can set the properties of JClass HiGrid components to determine how your data entry elements will look and behave. You can:

- Modify the number and arrangement of hierarchical levels. Customizers allow you to add or remove tables, fields, and joins as your project matures and your needs change.
- Include columns whose contents are computed from existing fields and, if necessary, other generated fields.
- Include header and footer columns which can contain aggregate information. For instance, a footer column may display the total amount of a number of purchase orders where each row in a table has a field containing the individual amount for that order.
- Present fields that contain various database and non-database types, including pictures.
- Use JClass Field components in cells to validate data entry operations.

JClass HiGrid also provides several methods which:

- Simplify connecting to a database, and allow you to build database applications more quickly using JDBC-ODBC bridge drivers or native-protocol all-Java drivers.
- Support transaction management.
- Permit you to control the appearance of the graphical user interface components as well as controlling the type of operation the end-user is permitted to perform on the records.

Other highlights of JClass HiGrid**:**

- JClass HiGrid is easy to configure (subclassable) or replicate.
- Events in JClass HiGrid derive from `HiGridEvent`, a base class for about a dozen specific grid events. The delegation event model means that you need only listen for the events in which you are interested.
- Listeners for these events all have Adapter classes that you can subclass.
- Many interfaces have `Default` versions.

- Formatting blocks of cells in the grid has been simplified by introducing `CellStyles` classes. There are `Default...CellStyle` classes for all five types of rows in JClass HiGrid.
- The header row can be made to temporarily replace the row above the pointer, ensuring that a copy of the header row is always visible even when scrolling a large table.
- Rows can be copied and pasted.
- Scrollbars can be placed on either edge of the grid, right or left, top or bottom.
- A "cursor tracking" mechanism makes it possible to change the pointer's icon depending on its location.
- The edit status column, which is used to mark edited rows, is an optional item.
- Java 2-style printing is supported.
- There is a `dispose()` method to ensure unused grids don't remain in memory.

JClass HiGrid may be used in conjunction with Quest's JClass Field, JClass LiveTable, and JClass Chart. These products permit data binding as well as providing you with additional Java components that complement or replace their equivalent AWT and Swing components.

All JClass HiGrid components are written entirely in Java; so as long as the Java implementation for a particular platform works, JClass HiGrid will work.

You can freely distribute Java applets and applications containing JClass components according to the terms of the License Agreement that appears at install time.

Assumptions

This manual assumes that you have some experience with the Java programming language. You should have a basic understanding of object-oriented programming and Java programming concepts such as classes, methods, and packages before proceeding with this manual. See Related Documents later in this section of the manual for additional sources of Java-related information.

Typographical Conventions in this Manual

`Typewriter Font`	■ Java language source code and examples of file contents. ■ JClass HiGrid and Java classes, objects, methods, properties, constants, and events. ■ HTML documents, tags, and attributes. ■ Commands that you enter on the screen.
Italic Text	■ Pathnames, filenames, URLs, programs, and method parameters. ■ New terms as they are introduced, and to emphasize important words. ■ Figure and table titles. ■ The names of other documents referenced in this manual, such as *Java in a Nutshell*.
Bold	■ Keyboard key names and menu references.

Overview of the Manual

For general instructions on installing JClass products, including JClass HiGrid, please see the *JClass DesktopViews Installation Guide*. It provides help with common configuration problems, including setting your CLASSPATH, establishing a database connection for running JClass HiGrid's examples, and IDE setup.

Part I – Using JClass HiGrid – describes how to use the JClass HiGrid programming components.

Chapter 1, JClass HiGrid Overview, presents an overview of JClass HiGrid's general structure and use.

Chapter 2, Properties of JClass HiGrid, provides additional information on using JClass HiGrid.

Chapter 3, JClass HiGrid Beans, discusses JClass HiGrid's Bean properties and shows how to use the custom property editor.

Chapter 4, Displaying and Editing Cells, discusses cell renderers and editors and presents an indication of how to write your own renderers and editors.

Chapter 5, JClass DataSource Overview, introduces the data access mechanism for JClass HiGrid.

Chapter 6, The Data Model, describes how a connection to a database is established.

Chapter 7, JClass DataSource Beans, discusses JClass DataSource's Bean properties and shows how to use the custom property editor.

Chapter 8, DataSource's Data Bound Components, presents the suite of data bound components that accompany the product.

Chapter 9, Sample Programs, illustrates some selected techniques that are useful in programming the grid.

Part II – Reference Appendices – contains detailed technical reference information.

Appendix A, Bean Properties Reference, contains tables listing the property names, return types, and default values for JClass HiGrid's JavaBeans.

Appendix B, Distributing Applets and Applications, illustrates how to use JClass JarMaster to help you combine only those JClass JARs that you really need for deploying your application.

Appendix C, Colors and Fonts, lists the common color names and their values. You may find this list useful when you are deciding which colors to use for various elements in your grid.

API Reference

The API reference documentation (Javadoc) is installed automatically when you install JClass HiGrid and is found in the *JCLASS_HOME/docs/api/* directory.

Licensing

In order to use JClass HiGrid, you need a valid license. Complete details about licensing are outlined in the *JClass DesktopViews Installation Guide*, which is automatically installed when you install JClass HiGrid.

Related Documents

The following is a sample of useful references to Java and JavaBeans programming:

- "*Java Platform Documentation*" at *http://java.sun.com/docs/index.html* and the "*Java Tutorial*" at *http://www.java.sun.com/docs/books/tutorial/index.html* from Sun Microsystems
- For an introduction to creating enhanced user interfaces, see "*Creating a GUI with JFC/Swing*" at *http://java.sun.com/docs/books/tutorial/uiswing/index.html*
- *Java in a Nutshell, 2nd Edition* from O'Reilly & Associates Inc. See the O'Reilly Java Resource Center at *http://java.oreilly.com*.

- Resources for using Java Beans are at *http://www.java.sun.com/beans/resources.html*

These documents are not required to develop applications using JClass HiGrid, but they can provide useful background information on various aspects of the Java programming language.

About Quest

Quest Software, Inc. (NASDAQ: QSFT) is a leading provider of application management solutions. Quest provides customers with Application Confidence[sm] by delivering reliable software products to develop, deploy, manage and maintain enterprise applications without expensive downtime or business interruption. Targeting high availability, monitoring, database management and Microsoft infrastructure management, Quest products increase the performance and uptime of business-critical applications and enable IT professionals to achieve more with fewer resources. Headquartered in Irvine, Calif., Quest Software has offices around the globe and more than 18,000 global customers, including 75% of the Fortune 500. For more information on Quest Software, visit *www.quest.com*.

Contacting Quest Software

E-mail	*sales@quest.com*
Address	Quest Software, Inc. World Headquarters 8001 Irvine Center Drive Irvine, CA 92618 USA
Web site	*www.quest.com*
Phone	949.754.8000 (United States and Canada)

Please refer to our Web site for regional and international office information.

Customer Support

Quest Software's world-class support team is dedicated to ensuring successful product installation and use for all Quest Software solutions.

SupportLink	*www.quest.com/support*
E-mail	*support@quest.com*

You can use SupportLink to do the following:

- Create, update, or view support requests
- Search the knowledge base, a searchable collection of information including program samples and problem/resolution documents
- Access FAQs
- Download patches
- Access product documentation, API reference, and demos and examples

Please note that many of the initial questions you may have will concern basic installation or configuration issues. Consult this product's *readme* file and the *JClass DesktopViews Installation Guide* (available in HTML and PDF formats) for help with these types of problems.

To Contact JClass Support

Any request for support *must* include your JClass product serial number. Supplying the following information will help us serve you better:

- Your name, email address, telephone number, company name, and country
- The product name, version and serial number
- The JDK (and IDE, if applicable) that you are using
- The type and version of the operating system you are using
- Your development environment and its version
- A full description of the problem, including any error messages and the steps required to duplicate it

JClass Direct Technical Support	
JClass Support Email	*support@quest.com*
Telephone	949-754-8000
Fax	949-754-8999
European Customers Contact Information	Telephone: +31 (0)20 510-6700 Fax: +31 (0)20 470-0326

Product Feedback and Announcements

We are interested in hearing about how you use JClass HiGrid, any problems you encounter, or any additional features you would find helpful. The majority of enhancements to JClass products are the result of customer requests.

Please send your comments to:
Quest Software
8001 Irvine Center Drive
Irvine, CA 92618

Telephone: 949-754-8000
Fax: 949-754-8999

Part I

Using JClass HiGrid

1

JClass HiGrid Overview

1.1 Introduction

JClass HiGrid provides your users with an intuitive structured visual representation of information retrieved from one or more databases. While a form with data bound fields is a one-record-at-a-time approach, JClass HiGrid lets you present your users with a hierarchically organized grid. The visibility of sub-levels is under the user's control, but it can be under programmatic control as well. Your design can include many sub-levels, resulting in an efficient concentration of information for tasks that require users to have many levels of detail available. Since you choose the detail records to be made available at every level, your design can be quite flexible and it can encompass many differing data retrieval needs. Not only can your users view the information, they can update it, add and delete rows–whatever you choose to permit.

Transaction processing is supported. If your application demands that some operations must be treated as a logical unit of work, the data source part of JClass HiGrid can (with your assistance, of course) execute the sequence as one atomic operation. If any one of the internal units fails to commit, all updates are rolled back and the database is left in its original state. Note that this is not an automatic recovery mechanism using log files, such as would be triggered by some type of system failure.

JClass HiGrid works with JDK 1.3.1 or later. It can contain hierarchical levels of information whose sub-tables the end-user can choose to show or hide dynamically at run time. The HiGrid can be contained in a resizable window, so the user has control over the size of the main level, and whether or not an individual detail level is showing.

1.1.1 The Relationship between JClass HiGrid and JClass DataSource

We'll take the simplest possible master-detail scenario to illustrate the way that the two products cooperate to present information extracted from a data source. We assume the existence of a database that contains a table called *Orders*. It contains the most frequently needed information about all the outstanding orders placed by every customer. Another table, called *OrderDetails*, contains additional information about these orders.

The entity-relationship diagram for this situation is shown below.

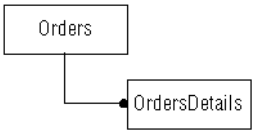


Figure 1 Entity-relationship diagram for a simple master-detail relationship.

In a master-detail scenario, one or more records of detailed information are displayed for each master record. In our example, the master records comes from the *Orders* table, while the detail records comes from the *OrderDetails* table. The database designer has already determined what type of information each of the tables contains, so you simply extract the pieces that you need in your application.

Meta Data

At this stage, you are working with the database's *meta data*. The meta data contains descriptive information about the tables, including the names of the data fields within each table and the SQL data types of each field. Each node in the entity-relationship diagram stores the `MetaData` of a given table in a simple tree-like structure called the `MetaDataTree`. The diagram shows that an *OrderDetails* sub-table will be created for each row of the *Orders* table, in order to show more detailed information about the specified order.

SQL Query

The exact contents of both the *Orders* and *OrderDetails* tables as displayed are determined by the SQL query used to retrieve the rows. A simple SQL query names the data fields of interest for a given table. A more complex SQL query adds a "WHERE" clause to restrict the number of records returned by the database. The SQL statement needed to construct an *OrderDetails* sub-table requires a WHERE clause that is able to select only the subset of records from the *OrderDetails* table within the database that is related to the master *Orders* record. This selection process is called a database "join." The join is usually accomplished by matching common fields within each table to each other. In this example, a field called *OrderID* will be used to construct an *OrderDetails* sub-table by selecting rows from the database's *OrderDetails* table where its *OrderID* field is the same as the `OrderID` field in the master *Orders* record.

At startup, the grid by default displays a single table called the root `DataTable`. The root `DataTable` is created by executing the SQL query associated with the `MetaData` node at the root of the `MetaDataTree`. In our example, the root `DataTable` will display the contents of the *Orders* table as the master records. Detail records, in the form of *OrderDetails* `DataTable`s are exposed at run time by clicking on an *expander icon*, also called a *folder icon*, used for that purpose. As the user clicks on more expander icons, more `DataTable`s are created. Thus, at run time, another tree-like structure called the `DataTableTree` is constructed, and it is composed of `DataTable`s.

While there is a 1:1 relationship between *Orders* and *OrderDetails* in the meta data diagram, there is a 1:N relationship between the *Orders* `DataTable` and the *OrderDetails* `DataTable`s associated with each row of the parent *Orders*. This occurs because any single order may consist of a number of items, each being described by an *OrderDetails* record. Both the grid and its underlying data source must be able to deal with structures of this type, and the more general type where the master-detail relationship forms an N-way tree.

Result Sets

Once the structure has been defined, and the SQL query has been formulated, the database can return data, called *result sets*, for the root-level table and for every one of the sub-tables. The queries that return sub-table result sets are performed as needed, when the grid needs to display them. It is the DataTable's job to store the result sets for later retrieval.

Visual Aspects of the Grid

Now that you can retrieve data in a hierarchical fashion, you may wish to control the visual aspects of the fields within the grid. For instance, you may want to adjust the size, color, or other visual property of the data fields. The following list illustrates those items that present information about the state of the grid and those that are under a user's control.

These are:

Clip Indicators	These small icons are present when the size of a cell is too small to display all of the text in it.
Column Moving	The initial position of a column is the same as the order in which they are mentioned in the database SELECT statement. End users can rearrange the order to suit themselves by dragging a column to a new position on the row.
Column Resizing	Columns may be resized by dragging on the right-hand border. The resizing operation affects all columns of the same type. If you place a grid in an Applet, or flush in a frame, the right-hand column lies very close to the frame's border. This might make it difficult for end-users to position the mouse pointer on the column's resizing border. The `setExtraWidth()` method in `HiGrid` lets you set more space between the rightmost column's border and the frame's border.
Dynamic Headers	The header row disappears off the top of the viewing area when an end-user scrolls down a large table. A method in `HiGrid`, called `setHeaderTipVisible()`, lets you control the visibility of a special "tool tip" that exactly replicates the header row and places it in the row above the mouse pointer.
Edit Status Column	The Edit Status Column presents a visual indication that marks the current row, and a pencil icon marks rows that end users have changed by editing one or more cells. You can control whether the edit status column appears. See the API documentation for `EditStatusCellFormat.setShowing()`. `EditStatus` is now a potentially replaceable public class. So is `NodeRenderer`, the class that draws the cell. To replace it, you need to extend `JCImageCellRenderer` and implement `HiGridNodeRenderer`. See Displaying and Editing Cells, in Chapter 4 for more details on JClass HiGrid's cell renderers.
Folder Icons	These icons present visual clues about the hierarchy of the data.
Indenting Subtables	Levels are indented by a fixed amount, if at all. Indentation is turned on or off using the `setLevelIndent()` method in `HiGrid`.
Popup Menu	A configurable popup menu gives end users access to frequently used commands. Popup commands are exposed, allowing you to invoke menu commands from your code.
Row-based copy and paste	Row-based copy and paste operations are available.

Row Resizing	Rows are resized by dragging on the horizontal boundaries of any Edit Status cell. The resizing operation affects all the rows in the corresponding table, not just the one that was dragged.
Sort Indicators	These indicators appear when an end user clicks on a column header. Since bidirectional sorting is implemented, the icon shows whether the sort is in ascending or descending order.

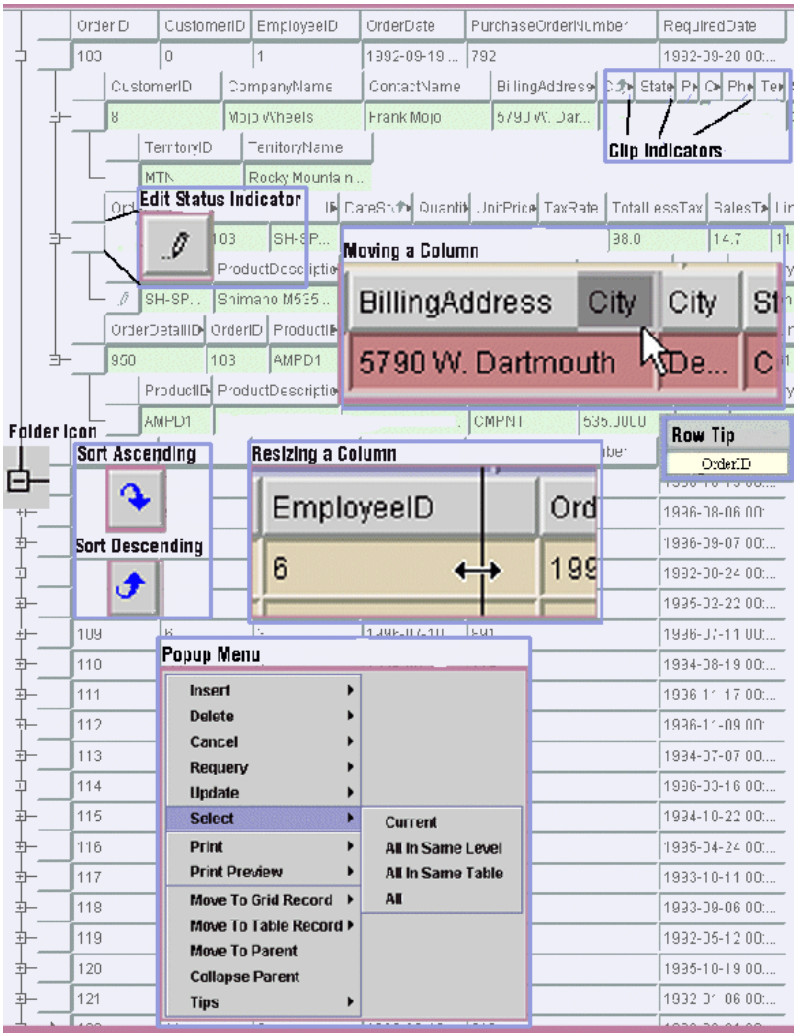


Figure 2 Active elements on HiGrid's graphical user interface.

Information about how a given DataTable is to be rendered is stored within a FormatNode. For each field in a MetaData node, there exists a corresponding field within a FormatNode that describes all visual aspects of that data field. FormatNodes are created at design-time and are stored within a FormatTree. There is one FormatNode stored in the FormatTree for each MetaData node, so the FormatTree bears a 1:1 relationship to the MetaDataTree on a node-by-node basis.

A RowTree is created to visually depict the contents of the DataTableTree at run-time. It is able to selectively display the contents of the entire DataTableTree. However, the RowTree is a logical superset of the DataTableTree because it contains rows that are not retrieved from the data source. These are the header, footer, before details, and after details rows which provide places for such things as column headings and summary information. The RowTree contains only the essential amount of information needed to reference the data in the DataTableTree and the formatting information in the FormatTree. It uses this information to manage the display of the grid.

Cells

The rows of the RowTree are comprised of cells. This is the basic unit of information in the grid. The source of a cell's value is a database field. Thus, within the context of this manual, the words field, cell, and column are used almost interchangeably[1].
JClass DataSource maintains the values for all these cells within DataTables – the grid does not maintain separate copies. Thus, for any row, the DataTableTree contains the actual data while the formatting information is drawn from the corresponding node in the FormatTree.

Besides specifying simple visual information such as color and font, each FormatNode also defines objects to display and edit a given cell. Each cell is drawn by an object called a Cell Renderer and edited by an object called a Cell Editor. This allows tremendous flexibility in terms of displaying and modifying data, since the modification of a cell's value is not tied to its display. For ease of use, a Cell Renderer and Cell Editor are automatically chosen for each field based on the cell's data type (although these choices can be overridden).

1.1.2 JClass HiGrid and the Model-View-Controller Paradigm

The design of JClass HiGrid conforms to the *Model View Controller* (MVC) paradigm, a technique for managing graphical user interfaces. MVC is a popular object oriented pattern that separates the application object (Model) from the way it is represented to the user (View), and from the way in which the user controls it (Controller). In the case of the JClass data binding products, the separation between the model and the view is achieved by providing two distinct packages. The data model is contained in JClass DataSource (the jclass.datasource package), and functions as the Model, and JClass HiGrid (the com.klg.jclass.higrid package) comprises the View and the Controller. Further

1. *Field* is used to emphasize the origin of the data (in a database field), *cell* is used when the emphasis is on the display, and *column* is used when referring to the group of cells in a DataTable that have the same field name.

separation of function is achieved by having a separate `Controller` class within JClass HiGrid to manage user interactions.

The `Controller` class is subclassable and replaceable.

The Model part of MVC keeps all the information about the organization and the state of the data. It also implements all the operations that can be used to manipulate the data. It has no responsibility for displaying the data, nor for the GUI actions that are used to manipulate the data. The Model's methods know how to access and modify data; the View methods manage the display of that data. The View object communicates with the Model. It uses the query methods of the Model to obtain data from the underlying database and then displays the information.

Because the View is separate from the Model, multiple views, even different kinds of views, can all draw their data from the same model. This makes it possible to have a form containing JClass Field components, a JClass LiveTable, and a JClass Chart all presenting data from a connection managed by JClass DataSource. Selecting a different cell in any one of the views and modifying its contents there causes all the corresponding cells in the other views to update themselves, thereby maintaining a consistent view of the data everywhere.

In the MVC paradigm, the Model is the object that manipulates the data. The View code relies on a public interface to the data. It does not need to know anything about implementation details. When the data model changes due to an update, it fires an event that is passed on to the View so that the information on the screen can be updated. The View has no memory, except for the layout structure. It refers to the data source whenever it needs to redisplay the data in cells.

The Controller object receives mouse events and keyboard inputs and translates them into commands for the model or the view. For example, a mouse click on a cell selects it and launches its editor, or a mouse click on the folder (expander) icon exposes dependent rows in a hierarchical grid. In some situations the Controller may interact directly with the View without needing to communicate with the Model. For example, the view may consist of a group of rows. Upon receiving a mouse-click on an editable cell in one of these rows, the Controller can request that the View should indicate that the selected cell is being edited, and launch the appropriate editor. The exchange of a cell renderer for a cell editor does not require that the data be updated in the Model. After a successful edit, the Model is informed to update the data source.

HiGrid's View and its Controller work together, but the Controller is also responsible for communicating data access operations back to the data source.

1.1.3 Types of Data Sources Supported

The grid may be bound to any of the common sources of data. Naturally, a JDBC data source is supported, as is any source that can be accessed through a JDBC-ODBC bridge. Unbound data can be presented as well, although you are responsible for adding the necessary code. In-memory data arrays or vectors, where the data has been generated as

the program runs, may be used as the data source. The example called `VectorData.java` sketches one approach that can be taken.

1.2 JClass HiGrid's Major Classes and Interfaces

The diagrams in this and the following two sections show a simplified inheritance hierarchy for the three major groups in the HiGrid family. The inheritance chain is not followed back to the ultimate ancestor; instead, the hierarchy within the packages is shown.

Among the many classes in the JClass HiGrid package, the main one is the `HiGrid` object itself. It delegates to another class, normally the `DefaultDataModelListener` class, in the JClass DataSource package, which is for objects that are interested in receiving `DataModelEvent`s. The `DataModelEvent`, through the parameter it passes, describes changes to the data source. Interested listeners can then query this data source to reflect the changes in their display. HiGrid implements three other interfaces, `ComponentListener`, and `JCValidateListener`. The first two are recognizable as standard

java.awt.event and java.awt interfaces respectively, and JCValidateListener, part of the com.klg.jclass.cell package, is for receiving value change events from a cell editor.

JClass HiGrid - Structure of the Major Classes and Interfaces

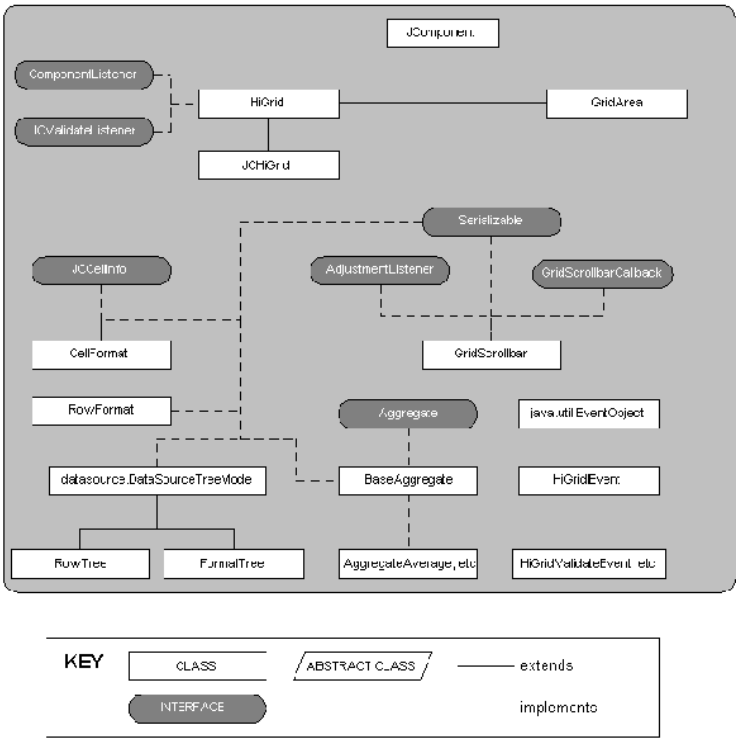


Figure 3 Major classes and interfaces for HiGrid.

GridArea passes mouse and keyboard events to the Controller class where they are handled. This class also positions the cell editors.

Class CellFormat implements the JCCellInfo interface, which lists the methods that must be defined to return information about the cell. CellFormat renders a given cell by implementing these methods. RowFormat sets the height of the row, its indent and grid style, and sets up the formats for the row.

GridScrollbar manages the interaction with the actual scrollbar. The position of the vertical scrollbar can be on either side of the grid, and the position of the horizontal

scrollbar can be at the top or the bottom of the grid. See `HiGrid`'s API for `setHorizontalScrollbarConstraints()` and `setVerticalScrollbarConstraints()`.

`RowTree` is based on `Tree`, which implements `TreeModel`, a generic interface for a Tree hierarchy. This tree interface is used for organizing the meta data and the actual data for the HiGrid.

For more information on the JClass HiGrid API, see the entries under `com.klg.jclass.higrid` in the Javadoc API for this product.

1.2.1 A Closer Look at JClass HiGrid

JClass HiGrid consists of one single GUI Component, with sub-tables rendered as necessary. The grid is bound to a database, although this is not strictly necessary, because the grid can be used in unbound mode. Either way, the grid is bound to some source of data. The grid is able to present any hierarchically organized (tree-like) data design. Normally, there will be a parent table (or joined parent tables) and one or more subsidiary tables. These subsidiary tables can themselves contain subsidiary tables, and so on. You can code your design or you can use JClass HiGrid's complete design-time customizer. See the discussion in JClass HiGrid Beans, in Chapter 3, for details on how to create a hierarchical data design using this design aid, the `HiGridBeanCustomizer`.

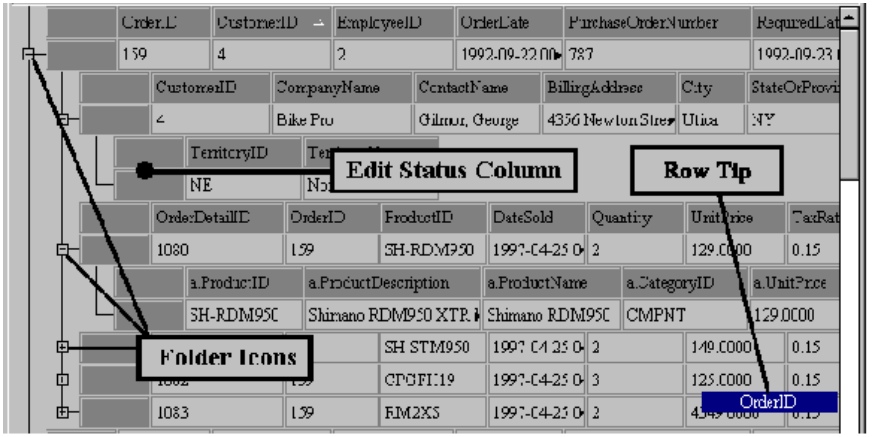


Figure 4 The general appearance of a grid, showing folder icon and edit status columns. A row tip appears while there is a mouse-down event on the scrollbar thumb.

The grid itself is the visual component for displaying the data model. It is expandable and collapsible through user interaction. The user can choose to expose dependent rows, and, if these rows also contain dependent rows, they too can be exposed. Changes to cells are effected by clicking on a cell to activate its editor. The changes are committed according to three selectable commit policies, ranging from manual to two types of deferred commit. Transaction processing is supported as a special case of the commit policy.

The appearance of a level in the structure can be individually specified. The Customizer can be employed to specify the appearance of a level.

One scrollbar controls the view of the top-level data. One way that end-users locate a row is by scrolling. They click on the scrollbar thumb and a "row tip" appears. The row tip's label is customizable, but by default it displays the contents of the first data column for the row that it is on. Note that there are two columns to the left of the first data column. The first is where the *Folder Icons* reside and the column to its right, containing initially blank squares, is the *Edit Status* column. The Folder Icons column shows the grid hierarchy in outline form, while the Edit Status column uses icons to indicate the various ways that a row may have been edited. The various icons are shown in the grid symbols table under Grid Symbols. Scrollbars are not available for sub-tables.

The grid area of the HiGrid is always double buffered. The result is faster updates and flicker-free operation.

1.2.2 Resizing the Grid

Since JClass HiGrid is subclassed from `java.awt.Component`, the grid's overall proportions are resized like any other window.

Users may resize any column horizontally by placing the mouse pointer at its right hand boundary where it becomes a double-ended arrow, then dragging to the new size. Resizing a column or row affects all columns/rows for that level of the table. Individual cells are selected by clicking on them, or by traversing to them using the arrow keys (unless the cell is in edit mode, in which case the right and left arrow keys position the I-beam in the field). Resize vertically by placing the mouse pointer at the bottom of one of the *Edit Status* column cells, then holding the left mouse button down and dragging to the new height. You can resize the width of a column no matter what row you are currently on. Drag the cell's left border to resize all cells in that column. By default, vertical resizing is invoked only on cells in the Edit Status Column, but there is a property that allows you to set vertical resizing on any cell if you wish. Once selected, a cell's contents may be modified, assuming that the proper update policy is in effect. Since cells may contain different data types, different cell editors will be invoked to effect the edits.

Jump scrolling is used in the vertical direction. A row is either visible or it is not; there are no half measures except for the last row, and only if the height of the view rectangle is incommensurable with an integral number of rows. On the other hand, horizontal scrolling by dragging the scrollbar is on a pixel-by-pixel basis, resulting in smoother scrolling in that direction. (Clicking on the scrollbar arrows results in jump scrolling by a few pixels at a time.)

1.2.3 The ActionInitiator Interface

JClass HiGrid uses the `ActionInitiator` interface as a way of interpreting user input. There is a default mapping between user actions (keystrokes and mouse clicks) and JClass HiGrid responses. Since not all user input is meaningful to JClass HiGrid, some

user input is ignored, while other input is associated with a specific command, such as navigating to some specific cell. This section gives an overview of the mechanism used to manage the mapping, starting with the `ActionInitiator` interface, whose job is to determine whether a match has been found between the `AWTEvent` resulting from a keystroke or mouse click and a defined mapping currently in effect for the grid.

Classes Related to the ActionInitiator Interface

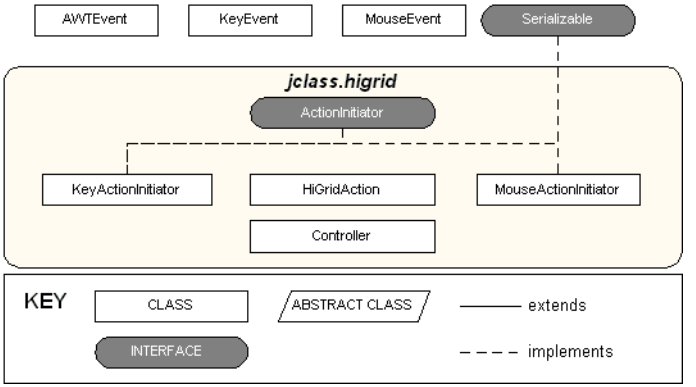


Figure 5 Classes and Interfaces relating to navigation through the grid.

The two classes that implement `ActionInitiator` are `KeyActionInitiator` and `MouseActionInitiator`. The two classes allow for distinguishing between keyboard actions and mouse actions. They have their own implementations of `isMatch()`, which determines whether a user input corresponds to one of the actions meaningful to the grid. Action handling code is contained in the `Controller`, which defines a list of default mappings using the constants found in `HiGridAction` that the grid uses to allow navigation through the grid. By default, keystrokes are interpreted as navigation commands, whereas a left-clicking on a cell launches the cell's editor, or right-clicking on a cell whose editor is inactive invokes a popup menu that contains database commands as well as navigation and print commands.

The various actions possible in JClass HiGrid are discussed in the following sections.

1.3 Operations on Cells

Once a cell is highlighted by left-clicking, an editor appropriate for the cell's data type is displayed. In this manual, it is referred to as the *current cell*.

There are the usual cell editors for String and numeric types, and a number of custom editors are employed as well, such as calendar popups for editing dates, and editors that perform data validation functions.

If you are a user of JClass LiveTable, it is useful for you to know that the cell editors for JClass HiGrid work in the same manner. They are single instance embeddable components.

Because only one editor at a time is allowed, there is no problem in basing the cell renderer on the data type. There are cell renderers for each different column in a table. Also, programmers can write their own cell renderers if they need to accommodate novel data types. See Displaying and Editing Cells, in Chapter 4, for details.

In a Microsoft Windows environment, the end-user modifies the cell's data with the help of an edit popup menu which is accessed by right-clicking on any cell while it is in edit mode.

The `EditPopupMenu` class defines a large number of enums for the possible items that can be chosen to appear in the edit popup menu. You can choose to place any of these in a customized version of the edit popup menu. A new set of popup menu items presents frames that contain instructions on using the mouse and keyboard shortcuts in JClass HiGrid.

Figure 6 The Edit Popup menu in Windows.

All the standard editing operations are available on the popup.

1.3.1 Cursor Tracking

A new method, called `setTrackCursor`, allows you to set a Boolean flag that controls whether you wish to be informed about the position of the cursor. You can then change the default cursor under program control.

1.3.2 Cell Traversal Via the Popup Menu

JClass HiGrid has its own popup menu, quite distinct from the edit popup menu in Windows. Right-clicking on any cell except the one with an active editor activates this HiGrid-specific object. The Popup Menu commands for moving from one cell to another are described in this section and shown in Figure 18, Section 2.5.6. Some of the Popup Menu items contain sub-choices, in which case they are shown in brackets following the main item.

- **Move To Grid Record** (*First, Previous, Next, Last*)

  Choosing *First* causes the first cell of the first row to be highlighted. Choosing *Previous* causes the first cell of the previous row to be highlighted. The action depends on

which sub-tables are open because the previous visible row will be chosen no matter what its level is. The same policy applies to *Move To Next*. Choosing *Last* causes the first cell on the last open row to be highlighted no matter what its level is. Thus, these operations potentially move the highlighted cell between levels in the master-detail hierarchy.

- **Move To Table Record** (*First, Previous, Next, Last*)

  This group maintains the selection within the same level (the same data table), but in all other respects functions similarly to *Move To Grid Record*.

- **Move To Parent**

  Focus moves to the leftmost cell in the parent row from the one on which the command was issued.

- **Collapse Parent**

  Collapse the table containing the highlighted cell. Other sub-tables at the same level as this one but belonging to different parents are not affected.

Edit Operation

- A highlighted cell may be edited unless you have flagged the column as read-only. When the cell has focus it is in edit mode, and right-clicking on the cell in a Windows environment produces a different popup with the usual edit options: *Undo*, *Cut*, *Copy*, *Paste*, *Delete*, and *Select All*.

Operations on Rows

The defined operations for rows are (bracketed items are the sub-choices):

- **Insert** (A list of names of tables–the names that are available in the current data source)

  Inserts a new row in the chosen table.

- **Delete** (*Current* or *Selected*)

  Marks the current or selected rows for deletion. When the rows are actually deleted depends on the commit policy.

- **Cancel** (*Current*, *Selected*, or *All*)

  Cancels the edits on the current row, on all selected rows, or all edits that have been done after the last commit.

- **Update** (*Current*, *Selected*, or *All*)

  Updates the edits on the current row, on all selected rows, or all edits that have been done after the last commit.

- **Requery** (*Record, Record and Details, Selected, Selected and Details*, or *All*)

Requeries the database. A requery can be done on an individual record, a record and its dependent tables, selected records with or without their dependent tables, or the whole database can be requeried. All modifications to the grid since the last commit will be lost.

- **Select** (*Current, All In Same Level, All In Same Table, All*)

  Causes the referenced cells to be selected.

- **Row height sizing operation**

  All rows of the same type must have the same height. Point the mouse at the lower border of the rectangle at the left of the first cell on any row (the **Edit Status** column). The mouse pointer turns into a double-tipped arrow. Click and drag to resize the height of all related rows to whatever height you selected for the row you are on.

1.3.3 Operations on Columns

- **Resizing columns horizontally**

  Resize the width of a column by placing the mouse pointer on the right hand border of the column header. The mouse pointer turns into a double-tipped arrow. Click and drag to resize the width. All the cells in the chosen column will be resized, even those that are underneath and separated by sub-table rows.

- **Resizing columns vertically**

  A column's height may not be resized individually. Only the row height is adjustable, as described in the previous section.

- **Sorting a column**

  Left-clicking on a column header at any level causes the rows at this level to be sorted in ascending order if the data type for the column is numeric, date, or String. A subsequent left-click reverses the sort order. A sorting indicator is displayed in the header column indicating whether the sort is ascending or descending.

- **Truncated fields**

  If the size of the cell is too small horizontally to display all the data, a small arrow icon appears on its right hand side if the text in the cell is right-justified, indicating that the cell should be resized to make its complete contents visible. Similarly, if the cell's height is too small the arrows appear at the top and bottom of the cell. The position of the arrows depends on which text justification policy (right, center, left) is in effect.

1.3.4 Displaying Images in Cells

Because all standard data types, including byte arrays, are supported, it is possible to display images in cells. In addition, it is possible to display a scaled image in a cell. See Displaying and Editing Cells, in Chapter 4, for details.

1.3.5 Cell Borders

All cells have a border around them. You can choose custom borders for cells, with variable styles and widths. There are ten different border styles to choose from, including a "no border" option.

JClass HiGrid's `CellFormat` class contains the methods that let you choose among the various styles of cell borders.

1.3.6 Keyboard Shortcuts for Grid Navigation

The `HiGridAction` class lets you define a mapping from a user event to an action performed on JClass HiGrid. For example, `HiGridAction` can be used to map **Control + Left Click** to a selection event.

These keystrokes control cell traversal:

- **Alt+Page Up, Alt+Page Down** – Scroll horizontally (Alt+Page Up = left) if the width of the grid is larger than the width of its window.
- **Ctrl+Home, Ctrl+End** – Move to the first or last cell in the table. If the last row of the main table has been expanded to show sub-tables, move to the last cell of the last visible sub-table.
- **Ctrl+Number Pad Plus(+), Ctrl+Number Pad Minus** (-) – Expand the current row to make the detail levels visible, or hide the detail levels by collapsing them.
- **Enter** – If pressed after an edit, the current row is marked as changed if the cell's value did indeed change.
- **Esc** (Escape key) – Cancel the current edit and return to the value the cell had before editing began, that is, return to the last updated value.
- **Home, End** – Move to the beginning or end of a row.
- **Page Up, Page Down** – Move to the corresponding cell on the previous or next page. A page is defined as the number of rows that are visible in a window, and thus it is dependent on the window height. If some rows have been expanded to show sub-tables, the Page Up and Page Down commands will show the rows of these sub-tables and ensure that none are bypassed.
- **Tab, Shift+Tab** – Move forward or backward one cell. Wraps from one row to the next.

1.3.7 Mouse Actions

HiGrid supports most of the standard mouse actions. The following list describes mouse actions in the Windows environment.

- **Left-Click** on a cell – Selects that cell for editing. If the cell contains a graphic, show its full size.

- **Left-Click** on an expander button (if there is one, it is the small square containing a + sign at the extreme left of a row) – Expands the table by making sub-tables associated with that row visible. If the sub-rows are visible, the button contains a - sign. Clicking on it collapses the table by hiding the sub-rows.
- **Right-Click** on a selected cell (Windows only) – When the button is released, shows a popup menu containing the editing choices **Undo, Cut, Copy, Paste, Delete, and Select All.**
- **Right-Click** anywhere else – Shows a popup menu containing the following choices. (The bracketed items are the sub-choices for each choice.)
  - **Insert** (*List of Table Names*) – Inserts a new row at the level specified by the table name.
  - **Delete** *(Current or Selected)* – Deletes the current or selected row(s).
  - **Cancel** *(Current, Selected, All)* – Cancels uncommitted changes to the specified row or rows.
  - **Requery** (*Record, Record and Details, Selected, Selected and Details, All*) – Refreshes the grid's values by requerying the database.
  - **Update** *(Current, Selected, All)* – Commits changes (deletes/inserts/updates).
  - **Select** *(Current, All In Same Level, All In Same Table, All)* – Selects rows in the grid.
  - **Print** (*As Displayed...*, *As Expanded...*) – Prints the exposed levels or all the levels.
  - **Print Preview** (*As Displayed...*, *As Expanded...*) – Invokes the printer driver window. After selecting a printer, show a print preview on the screen rather than actually printing.
  - **Move To Grid Record** (*First, Previous, Next, Last*) – If *First* (or *Last*) is chosen, focus is transferred to the leftmost cell of the first (or last) cell in the grid. A row in a collapsed table cannot be a target of a *Last* operation; instead, the last row of the last open table is the target. If *Previous* (or *Next*) is selected, the motion will, if possible, preserve the column that initially holds the current cell.
  - **Move To Table Record** (*First, Previous, Next, Last*) – Similar to the above, except that motion is restricted to the table that initially holds the current cell.
  - **Move To Parent** – Moves to the leftmost cell of the parent row for the table containing what was initially the current cell.
  - **Collapse Parent** – Hides the level containing the current cell. The new current cell is the leftmost one of the parent row.
- **Ctrl+Left-Click** – Selects the row. Multiple rows may be selected by repeating the operation on rows that need not be adjacent.
- **Shift+Left-Click** – After choosing **Select**, **Current** from the popup menu or selecting a row with **Ctrl+Left-Click**, this action selects all intervening rows between and including the first chosen and the one on which the shift-click occurred. The action can be used to select a parent and all its children, so long as these tables have been expanded.

1.3.8 Grid Symbols

Graphic	Meaning
Current Row Icon 	Icon Column: This is the selected row.
Row Edited Icon 	Icon Column: This row has been edited.
Marked for Deletion  Icon	Icon Column: This row has been marked for deletion.
Expand Icon 	Folder Column: The expander button indicates that there are sub-tables associated with this row. **Note:** Other icons are available. See Section 1.3.16, Folder Icon Styles.
Collapse Icon 	Folder Column: The appearance of the expander button when the row has actually been expanded
Truncated String Icon 	Anywhere on a cell's edge: This icon appears near the border of a cell when its size is too small to hold all the data.
Sort Icons 	Header Column: One of these icons appears after a left-click to indicate that the column has been sorted.

1.3.9 Bookmarks

The data source needs a mechanism for keeping track of all open rows. This is accomplished by assigning a row identifier, called a *bookmark*, to every row. Using this scheme, each cell is uniquely identified by its bookmark and by a column identifier which names the column within that row. The assignment of a bookmark to a row is dynamic. A row's bookmark may change as a result of requery operations. In fact, selecting the **Requery**, **All** option from the popup menu replaces all bookmarks with new ones. Other requery operations cause the replacement of the bookmarks of the affected rows.

1.3.10 A Note on Public Methods in HiGrid

There are a number of public methods in HiGrid that are not intended for use by the application programmer. They must be public so that they can be used by Bean editors and the like. As a general rule of thumb, consult the API documentation.

1.3.11 Editing Cells

Once a cell is highlighted, an editor appropriate for the cell's data type is instantiated. There are the usual cell editors for String and numeric types, and you can employ a number of custom editors, such as calendar popups for editing dates and editors that perform data validation functions.

1.3.12 Changing the Grid's Appearance

You can customize the grid's appearance by changing fonts, border styles, and colors. You can select from a set of predefined folder icons – the symbols that indicate whether a level is expanded – or you can design your own, and you can change the color and thickness of the connecting lines. These changes can be made programmatically, or they can be done in an IDE using the `HiGridBeanCustomizer`, which includes functionality for setting these properties one level at a time.

1.3.13 Adding Headers and Footers

Another way of customizing your grid's appearance is by adding header, footer, and detail rows. These rows can contain items that are not drawn directly from the data source yet are related to it, such as text fields that introduce summary columns, and computed results that the database itself does not supply.

1.3.14 Displaying More of the Grid

Imagine an idealized monitor so large that it is capable of accommodating a window of any size. It is useful to define the "visible" grid as that which would be seen in a virtual monitor's window spacious enough to hold all of its open tables. Any real monitor's window containing a grid can be thought of as one through which you can view a portion of the virtual screen that holds the entire grid. The concept of a visible grid is essential for understanding how the aggregate classes work. This topic is discussed in a later section. Also, the visible grid determines how much data must be retrieved from the data source. The grid requires all the data necessary to display the visible grid, not the collapsed layers. Thus, if the root-level table contains ten thousand rows, the data for all those rows must be retrieved because the root level is always visible.

The implication for displaying more of the grid is that simply resizing the view area causes the grid to be repainted with cached data. On the other hand, exposing sub-tables requires that a query be sent to the database, which is potentially more time-consuming.

1.3.15 The HiGrid Class

This central class in the package defines the overall look of the data grid. It sets up various parameters and controls such things as whether pop-up menus, row selection, and sorting are allowed. It sets colors, border sizes and styles, indents, spacing, and initializes print parameters. It manages the look of the GUI as levels are opened and closed, and as edits are made on cells, rows, or a group of rows. Instantiate this class to create a visible grid. Its signature is

```
public class HiGrid extends JComponent
    implements java.awt.event.ComponentListener,
        JCValidateListener
```

Because it is a subclass of `javax.swing.JComponent`, it inherits properties from the `Container` and `Component` classes as well. Naturally, it responds to window resizing and closing events. Among the methods contained in `HiGrid` are the following:

- `levelIndent` – a Boolean that controls whether a sub-table is left-indented.
- `width`, `height` – the width and height of the entire component.
- `verticalScrollbar`, `horizontalScrollbar` – in GridScrollbar, gets the scrollbars for the grid.
- `selectedObjects` – an array of references to `RowNode`s for the currently selected rows.
- `gridArea`– the double buffer for the grid area.
- `rowSelectionMode` – you may wish to prevent certain rows from being selected. Possible values are `ROW_SELECT_ANY` (the default), `ROW_SELECT_IN_SAME_LEVEL`, and `ROW_SELECT_IN_SAME_TABLE`.
- `dataModel` – sets the data source for the given level, which may be an instance of `HiGridData` or `TreeData`.
- `allowRowSelection` – sets whether or not row selection is allowed.
- `drawingConnections` – indicates whether or not the connector lines that join rows of sub-tables to their parents are to be drawn.
- `borderSize` – sets the size of the border to be drawn around a cell.
- `formatTree` – a format tree sets the visual characteristics for each level.

1.3.16 Folder Icon Styles

HiGrid has seven predefined folder icon styles. These are:

- `FolderIconStyle.FOLDER_ICON_STYLE_SHORTCUT`
- `FolderIconStyle.FOLDER_ICON_STYLE_FOLDER`
- `FolderIconStyle.FOLDER_ICON_STYLE_TRIANGLE`
- `FolderIconStyle.FOLDER_ICON_STYLE_SMALL_LINE_3D`
- `FolderIconStyle.FOLDER_ICON_STYLE_MEDIUM_LINE_3D`

- FolderIconStyle.FOLDER_ICON_STYLE_LARGE_LINE_3D
- FolderIconStyle.FOLDER_ICON_STYLE_TURNER

The icons are shown in Figure 7.

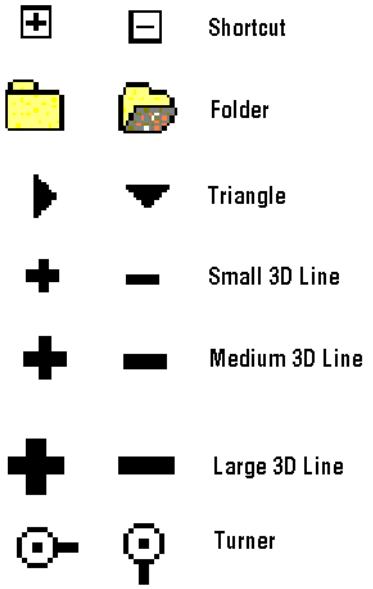


Figure 7 The seven pre-defined image icon choices.

You specify the folder icon style through the call:

```
HiGrid.setFolderIconStyleIndex(style);
```

where `style` is one of the aforementioned constants. You have the option of using your own icons. In this case, you use the `setFolderIcon` method:

```
public void setFolderIcon(Image icon, int type)
```

where you supply an icon and a type.

1.3.17 The Aggregate Classes

These classes are designed as a convenient way to calculate the information that is often required in summary records. The main class, called `AggregateAll`, implements the `Aggregate` interface. It contains methods common to all the different types of calculations that the "Aggregate" specialty subclasses define. The `SummaryColumn` class, through its parameters `identifier` (sets the column identifier), `columnType` (one of `COLUMN_TYPE_UNKNOWN`, `COLUMN_TYPE_LABEL`, `COLUMN_TYPE_DATASOURCE`, `COLUMN_TYPE_AGGREGATE`, `COLUMN_TYPE_UNBOUND`), and `aggregateType` (one of

AGGREGATE_TYPE_NONE, AGGREGATE_TYPE_COUNT, AGGREGATE_TYPE_SUM, AGGREGATE_TYPE_AVERAGE, AGGREGATE_TYPE_MIN, AGGREGATE_TYPE_MAX, AGGREGATE_TYPE_FIRST, AGGREGATE_TYPE_LAST), sets up the column object for the summary record types. The individual Aggregate classes compute results. For instance, AggregateAverage computes the average value of a given column.

These classes are not used directly. Instead, you set up the meta data for footer rows as a list (Vector) of summary columns using the columnType enums (class constants). Here are some examples:

Setup:

```
MetaDataModel orderDetailMetaData = null;
    RowFormat orderDetailFooterFormat = null;
    SummaryMetaData orderDetailFooterMetaData = null;
    RowFormat orderDetailBeforeDetailsFormat = null;
    SummaryMetaData orderDetailBeforeDetailsMetaData = null;

    TreeIterator ti = node.getIterator();
    if (ti.hasMoreElements()) {
        node = (FormatNode) ti.get();
        node.setDefaultSortData(new SortData("prod_id",
            SortGrid.DESCENDING));
        orderDetailMetaData = (MetaDataModel)
            node.getRecordFormat().getMetaData();
        orderDetailFooterFormat = (RowFormat) node.getFooterFormat();
        orderDetailFooterMetaData = (SummaryMetaData)
            orderDetailFooterFormat.getMetaData();
        orderDetailBeforeDetailsFormat = (RowFormat)
            node.getBeforeDetailsFormat();
        orderDetailBeforeDetailsMetaData = (SummaryMetaData)
            orderDetailBeforeDetailsFormat.getMetaData();
    }
```

Label:

```
    SummaryColumn column = new SummaryColumn("Total Quantity: ");
    orderDetailFooterMetaData.appendColumn(column);
```

Aggregate:

```
SummaryColumn column = new SummaryColumn(orderDetailMetaData,
       "LineTotal",
       SummaryColumn.COLUMN_TYPE_AGGREGATE,
       SummaryColumn.AGGREGATE_TYPE_SUM);
   orderDetailFooterMetaData.appendColumn(column);
```

Data Source:

```
SummaryColumn column = new SummaryColumn(orderDetailMetaData,
       "ProductID",
       SummaryColumn.COLUMN_TYPE_DATASOURCE);
   orderDetailBeforeDetailsMetaData.appendColumn(column);
```

1.3.18 Virtual Columns

A Virtual Column performs an analogous aggregation operation by using row data to produce a derived value. In the case of a virtual column, mathematical operations are defined on the cells of a row, such as applying a sales tax calculation to a cell containing the purchase price of an item. The computed total price with the sales tax added on is displayed in a newly defined cell. When all rows of the table are taken into account, these cells form a column that we are calling a *virtual column*. The virtual column may be computed from the values of two or more cells in the row. In turn, its value may be aggregated by the methods described in this section to produce a footer detail.

Note: Please see Virtual Columns, in Chapter 6, for an extended discussion of the use of virtual columns.

1.4 The Data Model for JClass HiGrid

JClass HiGrid is capable of displaying hierarchical data structures because its underlying data model is also capable of maintaining actual data tables that imitate the structure of a hierarchical design. The relationship that one table bears to another is called meta data to distinguish it from the actual data that the grid displays. To be specific, we present a design for a sales order system shown diagrammatically in Figure 8. The root data table is extracted from a database table called *Orders*. Each row consists of the fields *OrderID*, *CustomerID*, *EmployeeID*, *OrderDate*, *PurchaseOrderNumber*, and *RequiredDate*. Sales orders are more fully described in two separate sub-tables, *Customers* and *OrderDetails*. The *Customers* table is linked to its parent by matching the *CustomerID* fields in each. The field *OrderID* in the *OrderDetails* table is the same as *OrderID* in the *Orders* table so a join on these two fields properly associates an *Orders* row with *OrderDetails*. Additional

information about an *OrderDetails* row is obtained by a detail row called *Products-Categories* consisting of product information and the category this product falls into.

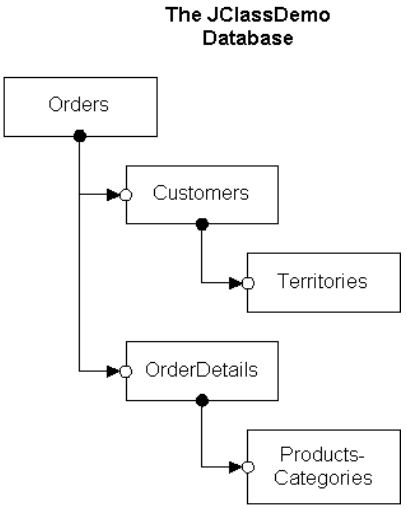


Figure 8 The meta data design of a sales order tracking system.

This structure can be captured in the data model and displayed using JClass HiGrid. The programming example based on this model is called `DemoData`, in `jclass.datasource.examples`.

Running the program produces output similar to this:

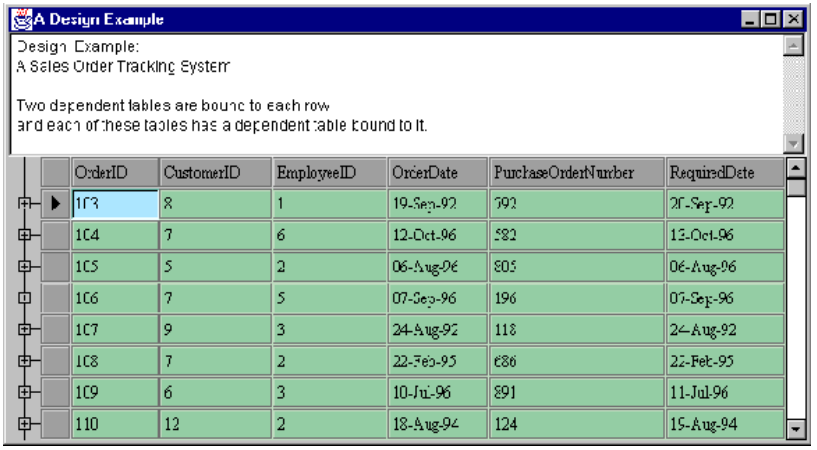


Figure 9 JClass HiGrid retrieves and displays the root table called Orders.

Clicking on any one of the folder icons marked "+" exposes the related tree-structured data. This example has three levels in its meta data. The second sub-level is accessed in the same way as the first, by clicking on the "+" expander button at the left hand side of the row.

When these levels are opened, the grid looks like this:

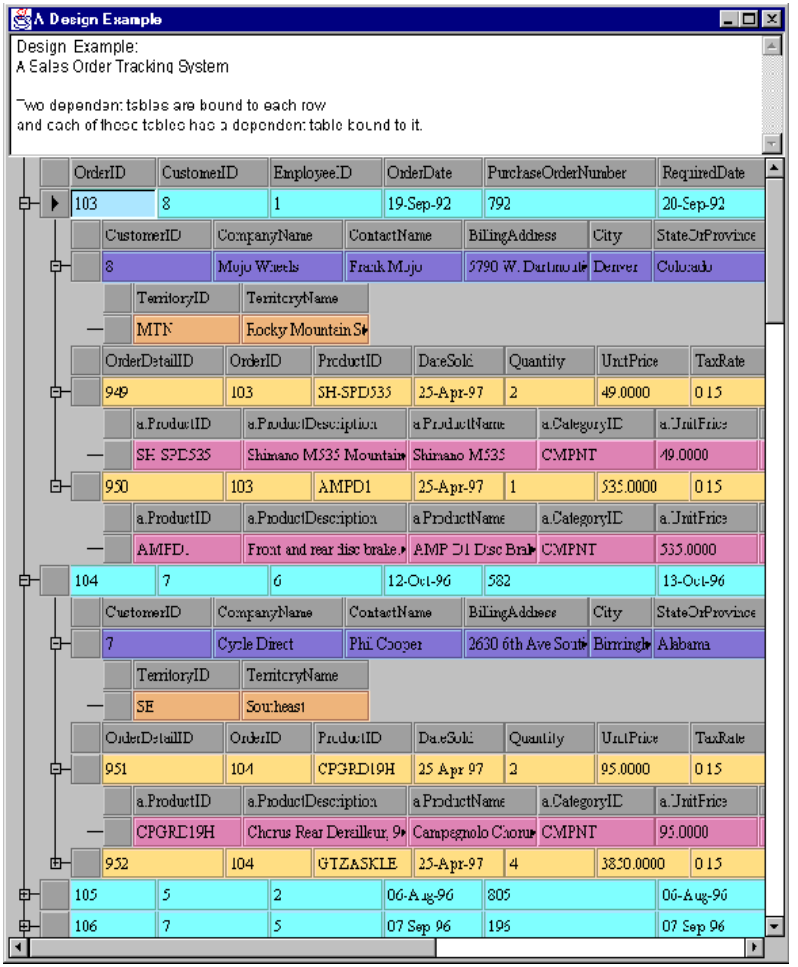


Figure 10 The expanded view of the sales orders tracking system with all levels of the first two rows opened.

The example shows that JClass HiGrid gives your application the ability to present a multi-layered view of the tables in your corporate databases. Compared to a design based on forms, the hierarchical grid allows you to present considerable data in a relatively small space, while also providing the organization that makes it easy for end-users to navigate to detail levels.

1.4.1 A Closer Look at the Data Model

There are two main areas in the design of the data model, the design-time meta data and the run-time data tables. Of these, the most apparent to the end-user is the data table mechanism that stores the data for subsequent display by a grid component, like JClass HiGrid, or on a form using data bound components like those provided by JClass Chart, JClass Field, JClass LiveTable, or JClass DataSource. Since JClass HiGrid defines a mechanism for describing data relationships in a hierarchical way, a parallel structure is needed to describe the way that various tables relate to each other. This is accomplished by using Swing's interface called `TreeNode` which describes the nodes of a `TreeModel`, a generic interface for a `Tree` hierarchy. This tree interface is used for organizing the meta data and the actual data for the JClass HiGrid. (**Note:** `TreeModel` contains `TreeNodeModel`, and `Tree` is a container for tree nodes.) The `DataSourceTreeNode` class, combined with the `MetaDataModel` interface, helps to define abstract class `BaseMetaData`, and then the concrete class `MetaData`. It is only this last one that is source data format dependent. This forms the meta data definition mechanism.

On the data side, we also subclass from `DataSourceTreeNode`, but the `DataTableModel` interface is used to define `BaseDataTable`. `DataTableModel` is the interface for data storage for the JClass HiGrid data model. The data model will request data from instances of `BaseDataTable` and will manipulate that data through the `DataTableModel` interface. That is, rows can then be added, deleted or updated through this DataTable. `BaseDataTable` is a default implementation of the methods and properties common to various implementations of the `DataTableModel`. This class must be extended to concretely implement those methods not implemented in it. The class that accomplishes this is in the DataTable category, and is one of a number of specially constructed classes specifically tailored to the source data format. A copy of the data returned in a JDBC result table will be copied into one of these result tables so the data can be cached. Rows can then be added, deleted or updated through this `DataTable`.

1.5 Internationalization

Internationalization is the process of making software that is ready for adaptation to various languages and regions without engineering changes. JClass products have been internationalized.

Localization is the process of making internationalized software run appropriately in a particular environment. All Strings used by JClass that need to be localized (that is, Strings that will be seen by a typical user) have been internationalized and are ready for localization. Thus, while localization stubs are in place for JClass, this step must be implemented by the developer of the localized software. These Strings are in resource bundles in every package that requires them. Therefore, the developer of the localized software who has purchased source code should augment all .java files within the */resources/* directory with the .java file specific for the relevant region; for example, for France, *LocaleInfo.java* becomes *LocaleInfo_fr.java*, and needs to contain the translated French versions of the Strings in the source *LocaleInfo.java* file. (Usually the file is called

LocaleInfo.java, but can also have another name, such as *LocaleBeanInfo.java* or *BeanLocaleInfo.java*.)

Essentially, developers of the localized software create their own resource bundles for their own locale. Developers should check every package for a */resources/* directory; if one is found, then the *.java* files in it will need to be localized.

For more information on internationalization, go to:
http://java.sun.com/j2se/1.4.2/docs/guide/intl/index.html.

2

Properties of JClass HiGrid

2.1 Introduction

The various tables that comprise a database are unstructured in the sense that any one of them can be chosen for display through some sort of data "control." Moreover, what is or is not a dependent table, and which fields are of interest, usually depends on some particular user's information needs. This structural design is up to you. Once it is completed and the table dependencies and relevant fields are established, you use JClass HiGrid to connect to the data source and display the results.

All of the database model-related code is contained within JClass DataSource, which is included with JClass HiGrid. You can use HiGrid's customizer, described in the next chapter, to get an IDE to assist you in producing the code to connect to a database, then define the tables, fields, and joins that define the hierarchical structure and its contents. This chapter takes more of a programming-related standpoint. If you are interested in a pictorial run-through of the steps necessary to connect to a database, define meta data levels, and format both data and summary rows, read JClass HiGrid Beans, in Chapter 3, first.

2.2 Programming JClass HiGrid

You set JClass HiGrid's properties programmatically by referring to its API. The next chapter shows how properties are set using a customizer. This design-time tool is a convenient way of setting the properties because a builder tool assists you in creating the underlying code.

The basic steps required to create a grid are:

1. Connect to a database.
2. Define the meta data levels by specifying the table hierarchy and the names of the relevant fields in each table.

3. Set the format of the grid.
4. Handle events if you need to inspect and perhaps prohibit certain end-user actions.

This chapter is devoted primarily to issues relating to the appearance of the grid and the outline of the event handling mechanism. An example of how to specify a database connection without the aid of the customizer is given next, and in The DemoData Program, in Chapter 9. Refer to JClass DataSource Overview, in Chapter 5, for details on specifying the meta data and formulating SQL queries.

2.2.1 Disposing a Grid

Your application may make use of multiple instances of JClass HiGrid objects. Keeping references to grid objects once they are no longer needed is, in effect, a memory leak. You can recover memory by deleting all references to unused grids. Use the `dispose()` method in `HiGrid` for this purpose.

2.2.2 Associating the Grid to a Data Source using JDBC or JDBC-ODBC Drivers

The section on Loading and Registering a Driver in the *JClass DesktopViews Installation Guide* showed how to register a Type 1 driver. It is possible to get an uninitialized connection object using `jclass.datasource.jdbc.DataTableConnection`'s zero-parameter constructor, but it is much more common to supply the driver, URL, login name and password, and the database name when calling the constructor to instantiate a connection. An example of loading a Type 4 driver follows.

```
c = new DataTableConnection(
    "com.sybase.jdbc.SybDriver", // driver
    "jdbc:sybase:Tds:localhost:1498",// url
    "dba",// user
    "sql",// password
    "JClassDemoSQLAnywhere");// database
```

The String containing the data source name has the form of a URL:

```
jdbc:sybase:Tds:localhost:1498
```

The subprotocol and subname vary from one supplier to another, depending on whose is used. The JDBC `DriverManager` uses the subprotocol as part of its choice of driver. Other common names for the subprotocol are “Oracle” and “odbc”. The location of the driver itself must be specified by giving its path, which in this case is:

```
com.sybase.jdbc.SybDriver
```

2.3 Cell Formats and Cell Styles

A cell consists of a border area and a drawing area. Typically, the drawing area contains textual information retrieved from a field in a database, but cells in header and footer rows, before detail and after detail rows, and even summary columns appended to record

rows usually contain information computed from the contents of a number of fields. All these have diverse formatting requirements. JClass HiGrid provides a suite of cell editors and renderers suitable for most purposes.

All cells that contain information extracted from a database have a data type corresponding to that in the source. Other Java data types are included so that cells can contain a wide selection of data types. The supported values in JClass HiGrid are `MetaDataModel.TYPE_BIG_DECIMAL`, `TYPE_BYTE`, `TYPE_BYTE_ARRAY`, `TYPE_DOUBLE`, `TYPE_FLOAT`, `TYPE_INTEGER`, `TYPE_LONG`, `TYPE_OBJECT`, `TYPE_SHORT`, `TYPE_SQL_DATE`, `TYPE_SQL_TIME`, `TYPE_SQL_TIMESTAMP`, `TYPE_STRING`, and `TYPE_UTIL_DATE`.

Normally, all cells have the same insets and border style, chosen from a predefined set. Possible border styles are `jclass.higrid.Border.NONE`, `ETCHED_IN`, `ETCHED_OUT`, `IN`, `OUT`, `PLAIN`, `FRAME_IN`, `FRAME_OUT`, `CONTROL_IN`, and `CONTROL_OUT`. There are two sets of insets defined for a cell. Conceptually, border insets define the placement of the border within a cell no matter what border style is chosen. Margin insets define the writable area within the border, usually chosen so that text will not seem to crowd against the border. `Inset` objects comprise four independently settable parameters for the number of pixels to reserve as unused space within the top, left, bottom, and right edges of the enclosing rectangle. The four parameters are normally chosen to be equal because a centered writable area generally has the best appearance.

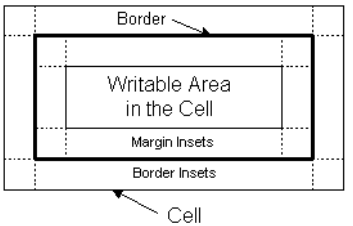


Figure 11 Border insets and margin insets.

Note: The class constants that describe border styles are named for the way the cell appears within its border. For instance, `Border.ETCHED_IN` is the constant for the case where the cell appears etched in with respect to the border.

JClass HiGrid provides three parameters to position contents horizontally within the cell, `com.klg.jclass.cell.JCCellInfo.LEFT`, `CENTER`, and `RIGHT`. To position a cell's contents vertically, use `com.klg.jclass.cell.JCCellInfo.TOP`, `CENTER`, or `BOTTOM`.

Cell editors need to decide how large they should be in relation to the size of the cell they are editing. Normally, a cell's editor will adjust itself to the display size of the cell, but this is not always possible, so HiGrid defines three possibilities: `EDIT_SIZE_TO_CELL`, which fits the cell editor to the cell size, `EDIT_ENSURE_MINIMUM_SIZE`, which uses the cell editor's minimum size (the default), and `EDIT_ENSURE_PREFERRED_SIZE`, which uses the cell

editor's preferred size. These policies can be independently applied to the cell's height and width attributes.

You can provide an indication that a cell is too small to completely display its contents. The small marker arrows are called *clip hints*. In Figure 12, if the arrow at the far right is showing, it indicates that the width of the cell is too small, and the double arrows indicate that the cell's height is too small. Whether or not these arrows appear is under your control. The possible values for clip hints are `SHOW_NONE`, `SHOW_HORIZONTAL`, `SHOW_VERTICAL`, and `SHOW_ALL`. These constants are defined in `com.klg.jclass.cell.JCCellInfo`. The position of the clip hint icons depends on which `CellInfo` positioning parameter is in effect.

Figure 12 The icons that indicate that a cell's contents have been truncated.

There's much more to cells, including cell editors and renderers. See Displaying and Editing Cells, in Chapter 4, for more information.

2.3.1 Properties of JClass HiGrid's CellFormat Class

The table shows the property names in the `CellFormat` class for which get and set methods exist. The property's return type and default value are listed as well.

HiGrid CellFormat Get/Set Method Name	Return Type	Default Value	Description
`allowWidthSizing`	Boolean	True	If `false`, the cell's width cannot be resized.
`background`	java.awt.Color	255,255,255	The background color.
`borderInsets`	java.awt.Insets	[2,2,2,2]	The border insets.
`borderStyle`	int	Border.ETCHED_IN	The border style–choices are listed in this section.
`cellEditor`	java.lang.Class	(null)	Retrieves or sets the cell editor for the current cell.

HiGrid CellFormat Get/Set Method Name	Return Type	Default Value	Description
cellRenderer	java.lang.Class	(null)	Retrieves or sets the cell renderer for the current cell.
clipHints	int	CellInfo.SHOW_ALL	How the renderer should draw an indication that the entire contents of the cell cannot be rendered within the given area.
dataType	java.lang.Class	class java.lang.Object	The cell's data type.
drawingArea	java.awt.Rectangle	*dynamic*	Read-only – the drawing area for the cell being displayed or edited, already adjusted for BorderSize and MarginSize – read-only.
editHeightPolicy	int	EDIT_ENSURE_ MINIMUM_SIZE	Instruct the cell editor which size policy to use – see the explanation in this section.
editWidthPolicy	int	EDIT_SIZE_TO_CELL	Instruct the cell editor which size policy to use – see the explanation in this section.
editable	Boolean	True	Whether input is currently allowed in the cell.
enabled	Boolean	True	Read-only, and always enabled. Needed to implement the CellInfo interface.

HiGrid CellFormat Get/Set Method Name	Return Type	Default Value	Description
font	java.awt.Font	*dynamic*	The cell's font.
fontMetrics	java.awt. FontMetrics	*dynamic*	The cell's font metrics.
foreground	java.awt.Color	0,0,0	The cell's foreground color.
height	int	*dynamic*	The cell's height.
horizontal Alignment	int	CellInfo.LEFT	The horizontal positioning of the cell's contents.
marginInsets	java.awt.Insets	[2,2,2,2]	The insets for the information area within the cell – inside the border and its insets.
name	java.lang.String	(null)	The name of the cell.
otherAllowWidth Sizing	Boolean	True	Sets whether width sizing is currently allowed for the other header/record pair.
parent	jclass.higrid. RowFormat	(null)	The row format of the parent row.
preferredTotalAr ea	java.awt.Rectangle	[x=0,y=0,width=8, height=8]	Read-only – the preferred area for the cell being displayed or edited.
selectAll	Boolean	True	Whether the editor should select all the cell's contents before editing.
selected Background	java.awt.Color	0,0,0	The background color when the cell's contents are selected.

HiGrid CellFormat Get/Set Method Name	Return Type	Default Value	Description
selected Foreground	java.awt.Color	255,255,255	The foreground color when the cell's contents are selected.
text	java.lang.String	(null)	The text label of this object.
totalArea	java.awt.Rectangle	*dynamic*	Read-only – the total area for the cell being displayed or edited.
type	int	MetaDataModel. TYPE_OBJECT	The Java data type used to map a JDBC data type.
verticalAlignment	int	JCCellInfo. CENTER	The vertical positioning of the cell's contents.
width	int	0	The width of the cell. Note that setWidth is a protected method.

If you wish to apply a new cell style to a group of cells, use the CellStyle class. It does not contain any dependency on the data in the cell, making it easy to copy a style to other cells. To apply your own styles to the different types of rows in JClass HiGrid, subclass DefaultHeaderCellStyle, DefaultFooterCellStyle, DefaultBeforeDetailsCellStyle, DefaultAfterDetailsCellStyle, and DefaultRecordCellStyle.

2.3.2 Setting Border Styles

From the table you see that cell border styles are set using setBorderStyle. Border styles, like the numerous other cell properties, are applied to individual cells. There is a convenience method in the RowFormat class, again called setBorderStyle, that lets you set the border style for the chosen row type, including its edit status cell. If you wish to set the border style globally for all rows, including records, headers, footers, and so on, you can follow the procedure outlined in FormatNodeExample.java. This example program in jclass.higrid.examples shows how to recursively walk the grid's format tree to set border styles. The example program's setBorderStyle method calls CellFormat's setBorderStyle method, but it could call any other method that changes cell properties, making global changes to them.

If you are using an IDE, you can use the customizer to set border styles for row types. See Setting General Column Properties, in Chapter 3.

2.4 Data Rows and Summary Lines

Besides presenting rows of data retrieved from database records, you can organize the tables into groups that both label and summarize what they contain. Four types of rows are available for this purpose. A table is usually introduced by a *header row* that by default contains the database record names for its columns. A *footer row* is often used to summarize the data in one or more columns, for instance by totaling the cost of all the individual entries in a *cost_price* column. Two other row types, *before details* and *after details*, are available. They provide another level where you can place summary data.

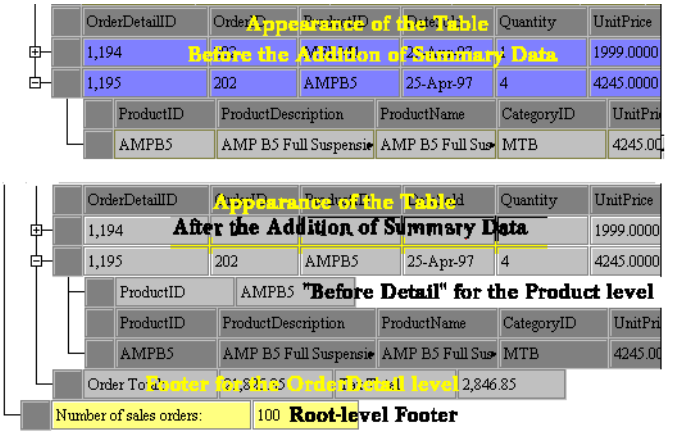


Figure 13 Appearance of the grid before and after the addition of summary data.

The remaining type of row is the data row: the one containing fields extracted from tables in the underlying data source. This section describes the formats that you can apply to these rows and shows, via code snippets, how you can add summary lines to your grid.

2.4.1 Row Formats

Row formats help you to display your hierarchically structured data design in a visually appealing way. Sub-tables are indented with respect to their parents, but they may be color coded and formatted to make them stand out, or to emphasize their relationship with their parent.

Rows begin with a default height, but may be changed if height sizing is allowed. The row width is simply the sum of the widths of all the cells in that row. The cell width depends

on its contents. Re-sizing may be permitted. A row of a sub-table is indented by a default amount, but the indentation may be changed by the application.

2.4.2 Header and Footer Formats

Header and footer formats are described by classes `HeaderFormat`, `FooterFormat` and `FooterMetaData`. You have the choice of making headers repeat or not. The default is to show the header only once, at the top of the first group of rows. This means that if a header is applied to the rows of the root table and some of that table's children are exposed, there will be a header at the top of the root table but no header will be present when the root rows begin again following the child rows. On the other hand, if `setRepeatHeader` is true, each group of the rows at the level for which this property is set has an identical copy of the header row introducing it. There can be only one footer row for a root-level table. If a footer is added to a sub-table, there is a footer row associated with each of these tables. Footer rows are made visible by calling `setShowing(true)` on the `RowFormat` object for the node in question.

The width of a header cell is determined by its associated cell in the data row, but the height, border style, and font attributes can be specified independently.

Footer rows are created on a per-table basis. At any given level, you cannot arrange for some rows to have footers and others not. Since footer rows may contain computed data, their meta data description is common to all of them but the contents of their cells will in general be different.

2.4.3 Adding Custom Headers and Footers

You can include headers and footers for each node in the meta data structure. The code examples in Section 2.4.5, Adding Summary Lines, show how a footer can be placed under each of the group of rows that comprise the second child of the root table. The overall design is shown in Figure 8 in JClass HiGrid Overview, in Chapter 1.

Column headers may be set using `setColumnLabel()` method . This allows you to supply your own custom labels rather than relying on the database table name for that column.

2.4.4 Before Detail and After Detail Formats

Before Detail and After Detail rows are rows of one level that encompass all the children of the next level. This distinguishes them from headers and footers, which always surround a single table rather than a group of tables. The top level cannot have before and after detail entries because it there is no parent with which they may be associated.

2.4.5 Adding Summary Lines

The points to consider when you are thinking of adding summary information to the tables and sub-tables in your grid are:

- Ensure that the information will be visible – this step is important because summary rows are not visible by default.
- Decide what information each summary cell will contain and how the information should be ordered.
- Create the group of column objects dictated by your design. Typically, the design will include paired columns: a column containing a label introducing a column containing computed summary information.
- Create the summary column.

After you have decided what information you wish to generate and have chosen a layout, you begin your program by organizing the various objects that you need. First, obtain the format-tree root object of your grid.

```
...
// Assume a HiGrid called "grid" has been instantiated
// and get the root of its format tree
...
FormatTree formatTree = grid.getFormatTree();
FormatNode node = (FormatNode) formatTree.getRoot();
```

Next, get the meta data model for the root level, the row format object for its footer row, and the summary meta data for the footer.

```
MetaDataModel ordersMetaData =
   (MetaDataModel) node.getRecordFormat().getMetaData();
RowFormat ordersFooterFormat = (RowFormat) node.getFooterFormat();
SummaryMetaData ordersFooterMetaData =
   (SummaryMetaData) ordersFooterFormat.getMetaData();
```

Declare more variables that will be needed.

```
MetaDataModel orderDetailMetaData = null;
RowFormat orderDetailFooterFormat = null;
SummaryMetaData orderDetailFooterMetaData = null;
RowFormat orderDetailBeforeDetailsFormat = null;
SummaryMetaData orderDetailBeforeDetailsMetaData = null;
```

Navigate to the node at which you want to place a footer.

```
TreeIterator ti = node.getIterator();
if (ti.hasMoreElements()) {
   node = (FormatNode) ti.nextElement();  // first child: Customers
   node = (FormatNode) ti.nextElement();  // second child, OrderDetails
   orderDetailMetaData =
       (MetaDataModel) node.getRecordFormat().getMetaData();
   orderDetailFooterFormat = (RowFormat) node.getFooterFormat();
   orderDetailFooterMetaData =
       (SummaryMetaData) orderDetailFooterFormat.getMetaData();
   orderDetailBeforeDetailsFormat =
       (RowFormat) node.getBeforeDetailsFormat();
   orderDetailBeforeDetailsMetaData =
       (SummaryMetaData) orderDetailBeforeDetailsFormat.getMetaData();
}
```

Set up a footer for the root level.

```
SummaryColumn column = null;

column = new SummaryColumn("Number of sales orders:");
ordersFooterMetaData.appendColumn(column);
           "OrderID",
           SummaryColumn.COLUMN_TYPE_AGGREGATE,
           SummaryColumn.AGGREGATE_TYPE_COUNT);
ordersFooterMetaData.appendColumn(column);

formatTree.setSummaryFormat(ordersFooterFormat,
                            ordersFooterMetaData);
ordersFooterFormat.setVisible(true);
```

Set up a footer for the second child of the root node.

```
column = new SummaryColumn("Order Total: ");
orderDetailFooterMetaData.appendColumn(column);
column = new SummaryColumn(orderDetailMetaData,
           "LineTotal",
           SummaryColumn.COLUMN_TYPE_AGGREGATE,
           SummaryColumn.AGGREGATE_TYPE_SUM);
orderDetailFooterMetaData.appendColumn(column);
```

Compute the sum you wish to display.

```
column = new SummaryColumn("Tax Total: ");
orderDetailFooterMetaData.appendColumn(column);
column = new SummaryColumn(orderDetailMetaData,
           "SalesTax",
           SummaryColumn.COLUMN_TYPE_AGGREGATE,
           SummaryColumn.AGGREGATE_TYPE_SUM);
orderDetailFooterMetaData.appendColumn(column);
```

Set the summary format and make sure the row will be visible.

```
formatTree.setSummaryFormat(orderDetailFooterFormat,
   orderDetailFooterMetaData);
orderDetailFooterFormat.setVisible(true);
```

Set up the Before Details for the same node.

```
column = new SummaryColumn("ProductID");
orderDetailBeforeDetailsMetaData.appendColumn(column);
column = new SummaryColumn(orderDetailMetaData,
           "ProductID",
           SummaryColumn.COLUMN_TYPE_DATASOURCE);
orderDetailBeforeDetailsMetaData.appendColumn(column);
```

Set the summary format and make sure the row will be visible.

```
formatTree.setSummaryFormat(orderDetailBeforeDetailsFormat,
   orderDetailBeforeDetailsMetaData);
orderDetailBeforeDetailsFormat.setVisible(true);
```

Regenerate the run-time grid. This refreshes all the information needed to display the grid. If you are adding summary columns dynamically to an already existing grid, the run-time grid needs to be refreshed by the following command:

```
grid.resetRuntimeGrid();
```

The result is shown in Figure 14. The root-level footer contains two cells, the first being a label, and the other an aggregate that counts the number of sales orders. The footer for the second level contains two aggregates, one for the total dollar value of sales orders and the second for the total tax.

Also shown is a *Before Detail* row for the second node of the first sub-level. Note that even though its association is with the first sub-level, it makes its appearance just before the third level headers. *Before Detail* and *After Detail* rows are not visible until the sub-level to the one with which they are associated is opened.

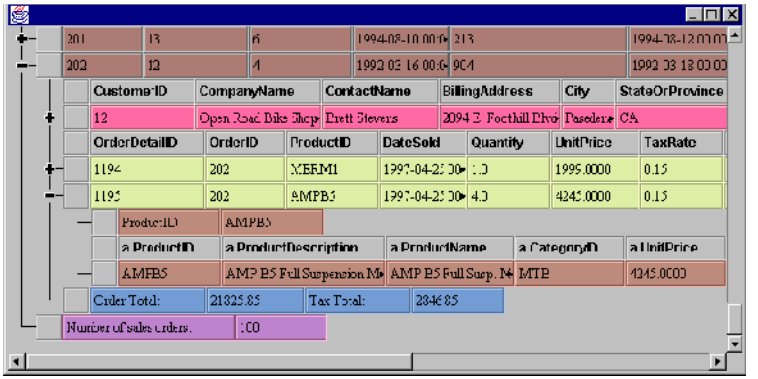


Figure 14 Adding a footer to the root level and summary data to a sub-table.

In short, the sequence is: get the meta data for the summary row, define a new column cell based on the type of data it is to contain, append the cell, set the summary format, ensure that it is visible, then reset the runtime tree.

2.4.6 Computed Summary Information

The easiest way to compute summary information is to override the `calculate` method of the `Aggregate` class, just as do all the aggregate classes with predefined operations, such as `AggregateMax`. Since you need to have an implementation of the `Aggregate` interface, you provide a new subclass that extends `AggregateAll`, and override the `calculate` method there.

```
public void calculate(RowNode rowNode) {
   if (isSameMetaID(rowNode)) {
       Object quantity = getRowNodeResultData(rowNode, "Quantity");
```

```
        Object unitPrice = getRowNodeResultData(rowNode, "UnitPrice");
        if (quantity != null && unitPrice != null) {
            double amount = getDoubleValue(quantity) *
                getDoubleValue(unitPrice);
            addValue((Object) new Double(amount));
        }
    }
}
```

The way that this code is reached is to name the class in which it resides. Assume that the name of the class is `OrderDetailTotalAmount`. Your calling method would issue a command like

```
column = new SummaryColumn(orderDetailMetaData,
    "jclass.higrid.examples.OrderDetailTotalAmount",
    SummaryColumn.COLUMN_TYPE_UNBOUND,
    SummaryColumn.AGGREGATE_TYPE_NONE,
    MetaDataModel.TYPE_DOUBLE);
```

Note the second parameter. Where there would be a *columnID* name if one of the predefined aggregate types were going to be used, there is a path name to the class that performs the special calculation. As always, the CLASSPATH variable is used to provide the first part of the path. The other parameters are chosen to be consistent with unbound data and the data type of the result of the calculation.

2.4.7 Managing the Visibility of Rows

By default, a new grid shows only the root level. If you wish to expand other levels, use `openFolder`. For instance, if you wish to open the first level on the first row, use:

```
grid.getRowTree().openFolder(grid.getCurrentRowNode());
```

If you want to open a row further down the hierarchy, you first need to navigate to that level. The row tree contains the row organization for the grid. It is analogous to the data table tree in JClass DataSource.

2.4.8 Moving Between Rows

The grid has a concept of a current row and various methods are available to move the current row pointer. All of these ultimately rely on the existence in the data model of a *bookmark*. The way that the data model keeps track of its records is described in The Data Model, in Chapter 6. In brief, there is a unique bookmark for every row. There is one cursor per table. The cursor is used to decide where the current row in the table should be. Also, an index is used to number the rows of a data table. At all times, one row of the grid is deemed to be the current row.

In general, moving to another row presumes that you already know its bookmark.

The `DataModel` interface's method `moveToRow` is used to tell this data model which row should be marked as current. Depending on the commit policy, moving the internal cursor may automatically commit changes to the originating data source.

The com.klg.jclass.datasource.DataTableModel interface declares methods absolute, next, previous, first, and last, which manipulate bookmarks and cursors. These are implemented in com.klg.jclass.datasource.BaseDataTable, com.klg.jclass.datasource.beans.JCData, and com.klg.jclass.datasource.util.NavigatorDataBinding. As the selected row moves around in a table, the cursor's value changes, but this is independent of the bookmark values.

There are menu choices in the edit popup menu corresponding to these motions. The following table lists the possibilities. See The Popup menu in its long form. (Figure 16). The choices are arranged in a two-tiered form near the bottom of the popup. The short form of the edit popup menu does not contain the choices that allow you to move between rows.

Edit Popup Menu Choice
MoveToGridRecord
MoveToGridRecordFirst
MoveToGridRecordLast
MoveToGridRecordNext
MoveToGridRecordPrevious
MoveToTableRecord
MoveToTableRecordFirst
MoveToTableRecordLast
MoveToTableRecordNext
MoveToTableRecordPrevious
MoveToParent

Row Status

Rows either reflect the contents of the database records from which they are derived, or they contain changes made by the end-user. A row that is consistent with the database record displays a plain rectangle in its edit status cell. If a row has been inserted, or some cell in an existing row has been modified, the edit status icon changes to a pencil, indicating an edit has occurred. A deleted row's edit status icon shows an *X*.

The constants that return the status of the current row, are DataTableModel.INSERTED, DataTableModel.UPDATED, DataTableModel.DELETED, and DataTableModel.COMMITTED.

2.4.9 Adding a Message Dialog

When a user interaction produces a predictable but unacceptable result, or just one that you want to present some information about, you can show a message dialog window. The `MessageDialog` class in `com.klg.jclass.util` is designed for this purpose.

```
MessageDialog dialog = new MessageDialog(myFrame, "Window Title",
    "Your message goes here");
```

A MessageDialog is always modal.

2.4.10 Using the Edit Popup Menu

JClass HiGrid's popup menu collects a number of useful functions and organizes them for convenient access. It comes in a short and a long version, or you can customize it by building your own version.

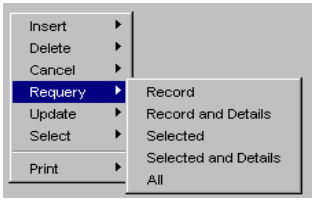


Figure 15 The Popup menu in its short form.

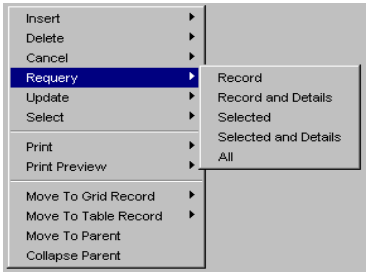


Figure 16 The Popup menu in its long form.

The methods for customizing the popup menu are found in the `EditPopupMenu` class. Assuming you have a `EditPopupMenu` object, perhaps by using the `getEditPopupMenu` method, you can choose either the short or the long versions. Use this command for the short version:

```
setDefaultMenuList(EditPoupuMenu.DEFAULT_SHORT_POPUPMENU_LIST)
```

Or use this one for the long version:

```
setDefaultMenuList(EditPoupuMenu.DEFAULT_LONG_POPUPMENU_LIST)
```

Programmatic access to the Edit Popup is available through the `HiGrid` methods shown in the following table.

HiGrid Method	Class Constants for Option Selection

`cancelRows`	`CANCEL_ROWS_ALL`
	`CANCEL_ROWS_CURRENT`
	`CANCEL_ROWS_SELECTED`

`deleteRows`	`DELETE_ROWS_SELECTED` `DELETE_ROWS_CURRENT`

`moveToRow`	`MOVE_TO_ROW_HIGRID_FIRST`
	`MOVE_TO_ROW_HIGRID_PREVIOUS`
	`MOVE_TO_ROW_HIGRID_NEXT`
	`MOVE_TO_ROW_HIGRID_LAST`
	`MOVE_TO_ROW_TABLE_FIRST`
	`MOVE_TO_ROW_TABLE_PREVIOUS`
	`MOVE_TO_ROW_TABLE_NEXT`
	`MOVE_TO_ROW_TABLE_LAST`
	`MOVE_TO_ROW_PARENT`

`print`	`PRINT_AS_EXPANDED` `PRINT_AS_DISPLAYED`
`print preview`	`PRINT_AS_EXPANDED` `PRINT_AS_DISPLAYED`

`requeryRows`	`REQUERY_ROWS_SELECTED`
	`REQUERY_ROWS_SELECTED_AND_DETAILS`
	`REQUERY_ROWS_RECORD_AND_DETAILS`
	`REQUERY_ROWS_RECORD`

`selectRows`	`SELECT_ROWS_ALL`
	`SELECT_ROWS_ALL_IN_SAME_TABLE`
	`SELECT_ROWS_ALL_IN_SAME_LEVEL`
	`SELECT_ROWS_CURRENT`

`updateRows`	`UPDATE_ROWS_ALL`
	`UPDATE_ROWS_SELECTED`
	`UPDATE_ROWS_CURRENT`

2.4.11 A Cell's Data Type

A cell's data type can contain any Java data type. Additionally, a cell may contain an image. The `MetaDataModel` interface defines the Java data types that are used to map JDBC data types, which in turn mirror SQL-92 data types.

Class Constant	SQL Type	Java Type
TYPE_BYTE	TINYINT	byte
TYPE_BYTE_ARRAY	VARBINARY or LONGVARBINARY	byte[]
TYPE_DOUBLE	DOUBLE	double
TYPE_FLOAT	REAL	float
TYPE_INTEGER	INTEGER	int
TYPE_LONG	BIGINT	long
TYPE_OBJECT		Object
TYPE_SHORT	SMALLINT	short
TYPE_SQL_DATE	DATE	java.sql.Date
TYPE_SQL_TIME	TIME	java.sql.Time
TYPE_SQL_TIMESTAMP	TIMESTAMP	java.sql.Timestamp
TYPE_STRING	VARCHAR or LONGVARCHAR	String
TYPE_UTIL_DATE		java.util.Date

2.4.12 Sorting Columns

Columns are sorted by clicking on their headers. Repeated clicking on the column header alternates the sort order between ascending and descending. The sort algorithm checks the column's data type. The sort is lexicographic for String-based data and numeric for integer, float, and double data. Dates are sorted by first converting them to numbers.

2.5 JClass HiGrid Listeners and Events

JClass HiGrid responds to events fired from different sources, of which mouse and keyboard actions form one class. `HiGrid`'s controller interprets mouse and keyboard actions to decide on a course of action. For instance, a right click on a cell area brings up a popup menu. `JCCellEditorEvent`s are a closely related class. Cell editors inform all listeners when they are finished a particular operation by posting a `JCCellEditorEvent`. It contains the event that originated the operation in the cell. This event (typically a key event) is interpreted by the container. The data model fires events that cause `HiGrid` to

redisplay its data based on the changing state of the database. Finally, HiGrid itself is a source of events. The pertinent classes are java.awt.event.ItemEvent, com.klg.jclass.cell.JCCellEditorEvent(extends java.util.EventObject), com.klg.jclass.datasource.DataModelEvent, and com.klg.jclass.higrid.HiGridEvent.

JClass HiGrid no longer relies on JClass DataSource's DataModelEvent. Instead, it defines a JDK 1.1-style delegation event model to dispatch events to interested listeners. The base class for most of these events is HiGridEvent.

Those classes interested in specific events must register themselves as listeners.

There are adapter classes for all the event classes. These classes are abstract, allowing you to override just the cases you need.

Event Method	Description
HiGridColumnSelection Event	The event that occurs when a column is selected. Constant: SELECT_COLUMN. Method in HiGridColumnSelectionListener: selectColumn()
HiGridErrorEvent	Process an event that occurs when HiGrid is running. Constant: none. Method in HiGridErrorListener: processError()
HiGridEvent	The base class for HiGrid events.
HiGridExpansionEvent	Cases are BEFORE_EXPAND_ROW, BEFORE_COLLAPSE_ROW, AFTER_EXPAND_ROW, and AFTER_COLLAPSE_ROW. The event can be canceled. Methods in HiGridExpansionListener: beforeExpandRow() beforeCollapseRow() afterExpandRow() afterCollapseRow()
HiGridFormatNodeEvent	Cases are BEFORE_CREATE_FORMAT_NODE_ CONTENTS, and AFTER_CREATE_FORMAT_NODE_ CONTENTS. Methods in HiGridFormatNodeListener: beforeCreateFormatNode Contents() afterCreateFormatNode Contents()

Event Method	Description
HiGridMoveCellEvent	Cases are BEFORE_MOVE_COLUMN and AFTER_MOVE_COLUMN. Note that it is possible to cancel an end-user's attempt to move a column by catching the BEFORE_MOVE_COLUMN event. Methods in HiGridMoveCellListener: beforeMoveColumn() afterMoveColumn()
HiGridPrintEvent	Cases are PRINT_HEADER and PRINT_FOOTER, PRINT_END. Methods in HiGridPrintListener: printPageHeader() printPageFooter() printEnd()
HiGridRepaintEvent	The single case is BEFORE_REPAINT_ROW. Method in HiGridRepaintListener: beforeRepaintRow()
HiGridResizeCellEvent	Cases are BEFORE_RESIZE_ROW, BEFORE_RESIZE_COLUMN, AFTER_RESIZE_ROW, and AFTER_RESIZE_COLUMN. The event can be canceled. Method in HiGridResizeListener: beforeResizeRow() beforeResizeColumn() afterResizeRow() afterResizeColumn()
HiGridRowSelectionEvent	Cases are BEFORE_SELECT_ROW and AFTER_SELECT_ROW. Note that by intercepting a BEFORE_SELECT_ROW event, row selection becomes a cancelable event. Methods in HiGridRowSelectionListener: beforeSelectRow() afterSelectRow()
HiGridSortTableEvent	Cases are BEFORE_SORT_TABLE and AFTER_SORT_TABLE. Methods in HiGridSortTableListener: beforeSortTable() afterSortTable()
HiGridTraverseEvent	The single case is AFTER_TRAVERSE. Method in HiGridTraverseListener: afterTraverse()

Event Method	Description
HiGridUpdateEvent	Cases are AFTER_RESET_FORMAT_DATA, AFTER_RESET_HIGRID_DATA, and AFTER_CREATE_ROW. Methods in HiGridUpdateListener: afterCreateRow() afterResetFormatData() afterResetHiGridData()
HiGridValidateEvent	Cases are VALUE_CHANGED_BEGIN, VALUE_CHANGED_END, and STATE_IS_INVALID. Methods in HiGridValidateListener: valueChangedBegin(), valueChangedEnd(), stateIsInvalid()
beans.JCHiGridEvent (for the JCHiGrid JavaBean)	No constants are defined in this class. Method in JCHiGridEventListener: JCHiGridValueChanged()

In the foregoing table, the listener method's parameter is the associated event object, for example, beforeResizeRow(HiGridResizeCellEvent event).

2.5.1 Moving Columns

Using the JClass HiGrid model and its event mechanism is exemplified by the task of moving a column of cells. An end user may use the mouse to drag a column header from one position to another. The index labeling the column's position changes. The first two methods of the example simply print out the initial and final column indices. The code in getColumnIndex() gets the column name and the Vector of data formats for the row, then searches the data formats for the cell with the given name, thus determining the new column index.

```
class ViewEventsHiGridMoveCellListener extends HiGridMoveCellAdapter {

public void beforeMoveColumn(HiGridMoveCellEvent e) {
    System.out.println("Before move column index = "+
        getColumnIndex(e));
}
public void afterMoveColumn(HiGridMoveCellEvent e) {
    System.out.println("After move column index = "+
        getColumnIndex(e));
}
public int getColumnIndex(HiGridMoveCellEvent e) {
    // get the name of the column that we are interested in
    String name = e.getColumn();
    // get the vector of cells for this row
    Vector dataFormats =
        e.getRowNode().getRowFormat().getDataFormats();
    // look for a CellFormat with the given name
    for (int i = 0; i < dataFormats.size(); i++) {
```

```
        CellFormat cellFormat = (CellFormat) dataFormats.elementAt(i);
        if (cellFormat.getName().equals(name)) {
            // found it!
            return i;
        }
    }
    // not found
    return -1;
}
}
```

2.6 JClass DataSource Events and Listeners

JClass HiGrid relies on the event handling mechanism of JClass DataSource for everything related to data model events. These occur when the data being displayed by the grid is edited and committed by user action, or because one or more of the database fields currently being displayed was changed by another agent. In either case, the grid must be synchronized with the database, and data model changes must be propagated to the grid.

The following diagram depicts the classes and interfaces that JClass DataSource uses to manage changes to its data model. There are two listener interfaces, `ReadOnlyBindingListener` and its extension, `DataModelListener`, but only one event class, `DataModelEvent`. `ReadOnlyBindingListener` is for the read-only events in `DataModelEvent`, and `DataModelListener` extends it, adding methods for listeners that will make changes to the data model.

The `ReadOnlyBindingModel` interface provides a single-level, two-dimensional view of a set of data. It groups all non-update methods and handles read-only events. This interface exists only to provide a logical separation between read-only and non-read-only methods and event handling. Update methods are declared by `BindingModel`.

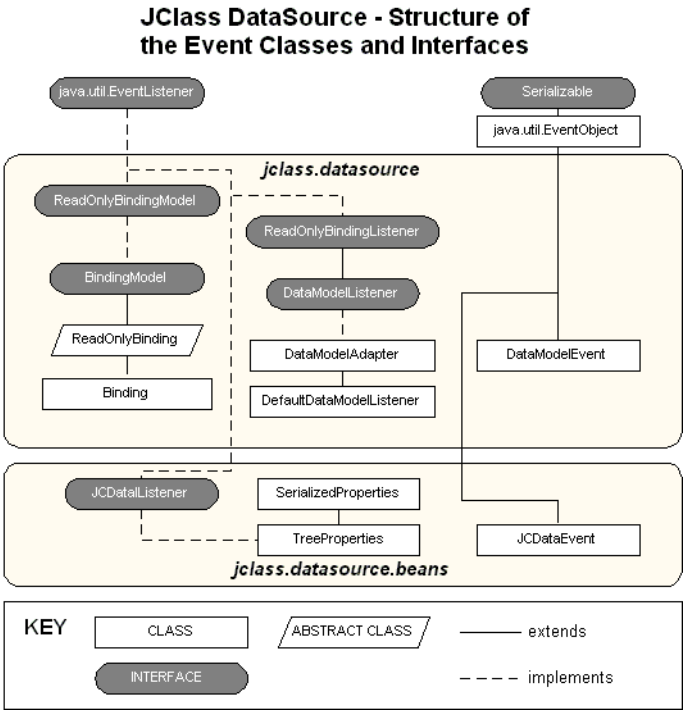


Figure 17 Classes and interfaces related to event handling in JClass DataSource.

JClass DataSource's `DataModelEvent` describes changes to the data source. An interested listener can query this data source to reflect the changes in its display. `DataModelEvent` defines these methods:

Event Method	Description
cancelProposedAction	Cancels the proposed action. This method can be used if the action is cancelable. You may want to test that `isCancelable` is `true` before calling `cancelProposedAction`.
getAncestorBookmarks	Returns a list of the bookmarks which comprise the path from the root to the event node.
getBookmark	Returns the bookmark of the changed row.
getCancelled	Sees if the proposed action was cancelled by the listener.
getColumn	Returns a String indicating which column changed.

Event Method	Description
`getCommand`	Gets the command which indicates what action should be taken.
`getRowIndex`	Returns the row index of the changed row.
`getOriginator`	Returns the `DataModelListener` which initiated this event. Allows a listener to determine if it was also the originator of the event.
`getTable`	Returns the `DataTableModel` related to this event.
`isCancelable`	Returns true if this event can be cancelled.

Events are characterized by the class constants given in the following table. Listeners can distinguish various cases within the event structure by examining these constants and taking the appropriate action. Some of these constants are for rare situations, or for internal use. These are the minimum to which a listener should respond:

```
AFTER_CHANGE_OF_ROW_DATA
AFTER_INSERT_ROW
AFTER_DELETE_ROW
AFTER_RESET
AFTER_REQUERY_ROW_AND_DETAILS
AFTER_MOVE_TO_CURRENT_ROW
```

See the `DataModelEvent` API and the following table for the full list of event constants.

2.6.1 The Class Constants Defined in DataModelEvent

Applications that simply need to display a grid find that all event handling is done transparently. Events do need to be caught and handled by applications that need to inspect and possibly deny some of the actions that end-users may take. The “BEFORE” events shown in the table below can be used to let your application inspect changes made by the end-user and perform its own validation before passing them back to the data source.

The data model's events are distinguished by the group of class constants listed here:

DataModel Event Class Constants and Corresponding Listener Method	Description
AFTER_CHANGE_OF_ROW_DATA ReadOnlyBindingListener method: afterChangeOfRowData()	A row has changed, re-read all its cells and its status to reflect the new values. If this event is the result of a cell edit, call DataModelEvent.getColumn() to get the name of the column which changed. If getColumn() returns " ", re-read the entire row. Called when one of the following is true: ■ a row is deleted and getShowDeletedRows() == true ■ a cell was edited ■ row edits are cancelled and getRowStatus == UPDATED ■ row edits are cancelled and getRowStatus == DELETED and getShowDeletedRows == true ■ row is committed and getRowStatus == UPDATED or INSERTED ■ row is requeried and getRowStatus != INSERTED
AFTER_DELETE_ROW ReadOnlyBindingListener method: afterDeleteRow()	Removes the row from the display. A row has been physically deleted and needs to be removed from the display or has been logically deleted but the showDeletedRows property has been set to false. Called when ■ a row has been logically deleted and getShowDeletedRows == false ■ row changes have been cancelled and getRowStatus == INSERTED ■ a row is committed, getRowStatus == DELETED, and getShowDeletedRows == true ■ a row has been requeried and getRowStatus == INSERTED
AFTER_INSERT_ROW ReadOnlyBindingListener method: afterInsertRow()	A new row has been added to the datasource. Listeners need to display the row. Rows are always added to the end of DataTableModels.

DataModel Event Class Constants and Corresponding Listener Method	Description
AFTER_MOVE_TO_CURRENT_ROW ReadOnlyBindingListener method: afterMoveToCurrentRow()	The global cursor has moved to a new row. Listeners should position their cursor on the indicated row. In a master-detail relationship, child levels should refresh themselves to reflect data sets which correspond to the new parent row by calling DataModel.getCurrentDataTable() or, for field controls, DataModel.getCurrentDataItem().
AFTER_REQUIRY_ROW_AND_DETAILS ReadOnlyBindingListener method: afterRequeryRowAndDetails()	Re-reads the indicated row and refreshes all open children under this row.
AFTER_REQUERY_TABLE ReadOnlyBindingListener method: afterRequeryTable()	Re-reads this table and refreshes all open children in the table.
AFTER_RESET ReadOnlyBindingListener method: afterReset()	Listeners must close all expanded views and reset/re-read the root node. The previous pointer to the root node is no longer valid. Call DataModel.getDataTableTree().getRoot() for the new root table. Called when the datasource has been reset. See DataModel.requeryAll().
BEFORE_CANCEL_ALL BEFORE_CANCEL_ROW_CHANGES BEFORE_EDIT_CELL BEFORE_COMMIT_ALL BEFORE_COMMIT_ROW BEFORE_COMMIT_CONDITIONAL BEFORE_MOVE_TO_CURRENT_ROW BEFORE_REQUERY BEFORE_RESET BEFORE_DELETE_ROW BEFORE_INSERT_ROW	These "BEFORE_" events can be ignored. They are simply to allow applications to cancel the event.

DataModel Event Class Constants and Corresponding Listener Method	Description
BEFORE_CANCEL_ALL DataModelListener method: beforeCancelAll()	Event fired before all changes are cancelled. Can be cancelled. AFTER_INSERT_ROW and AFTER_CHANGE_OF_ROW_DATA events can follow this event. See DataModel.cancelAll().
BEFORE_CANCEL_ROW_CHANGES DataModelListener method: beforeCancelRowChanges()	Event fired before all edits to a row are undone. Can be cancelled. An AFTER_DELETE_ROW or AFTER_CHANGE_OR_ROW_DATA event will follow. See DataTableModel. cancelRowChanges().
BEFORE_COMMIT_ALL DataModelListener method: beforeCommitAll()	Event fired before all changes are committed. Can be cancelled. All modified, deleted and inserted rows at all levels are about to be committed. BEFORE_DELETE_ROW and AFTER_CHANGE_OF_ROW_DATA events will follow depending on the operations performed on the modified rows being saved. Results from a call to DataModel.updateAll(). See DataModel.updateAll().
BEFORE_COMMIT_CONDITIONAL DataModelListener method: beforeCommitConditional()	Called when the root-level bookmark for a subtree changes. When this happens, those nodes in the previous subtree which are not COMMIT_MANUALLY are committed. Can be cancelled. If cancelled the cursor moves but the changes are automatically committed.
BEFORE_COMMIT_ROW beforeCommitRow()	Called before single row is committed to data source. Can be cancelled, in which case the row edits are not written to the datasource and the rows status remains modified. AFTER_DELETE_ROW or AFTER_CHANGE_OF_ROW_DATA events will follow depending on the status of the row to be committed. See DataTableModel.commitRow().

DataModel Event Class Constants and Corresponding Listener Method	Description
BEFORE_DELETE_ROW DataModelListener method: beforeDeleteRow()	Event fired before a row is [logically] deleted. Can be cancelled. If not cancelled, this event will be followed by an AFTER_ROW_DELETE or a ROW_STATUS_CHANGED message if the commit policy is COMMIT_MANUALLY or COMMIT_LEAVING_ANCESTOR. See DataTableModel.deleteRow(), MetaDataModel.getCommitPolicy().
BEFORE_DELETE_TABLE DataModelListener method: beforeDeleteTable()	The indicated Data Table will be deleted and flushed from the cache. Can be cancelled.
BEFORE_EDIT_CELL DataModelListener method: beforeEditCell()	Event fired before a cell is edited. Can be cancelled. See DataTableModel.updateCell().
BEFORE_INSERT_ROW DataModelListener method: beforeInsertRow()	Event fired before a row is inserted. Can be cancelled. If not cancelled, this event will be followed by an AFTER_INSERT_ROW event. See DataTableModel.addRow().
BEFORE_MOVE_TO_CURRENT_ROW DataModelListener method: beforeMoveToCurrentRow()	The global cursor will move to a new row. Can be cancelled.
BEFORE_REQUERY DataModelListener method: beforeRequery()	Event fired when either DataTableModel.requeryRow AndDetails() or DataTableModel.requeryRow() is called. If not cancelled, this event will be followed by an, AFTER_REQUERY_ROW_AND_DETAILS event, an AFTER_ROW_DELETE event in the case getRowStatus() == INSERTED, or a ROW_STATUS_CHANGED event in the case getRowStatus() == UPDATED or COMMITTED. See DataTableModel.requeryRow(), DataTableModel.requeryRowAnd Details().

DataModel Event Class Constants and Corresponding Listener Method	Description
BEFORE_RESET DataModelListener method: beforeReset()	Event fired before entire grid is reset. Can be cancelled. If not cancelled this event will be followed by an AFTER_RESET event. This event will result from a call to DataModel.requeryAll().
BEGIN_EVENTS ReadOnlyBindingListener method: beginEvents()	Notification that multiple events are coming. Multiple events will be nested between BEGIN_EVENTS and END_EVENTS events. Allows listeners to treat the events as a batch to, for example, reduce repaints.
END_EVENTS ReadOnlyBindingListener method: endEvents()	Notification that multiple events are complete. Multiple events will be nested between BEGIN_EVENTS and END_EVENTS events. Allows listeners to treat the events as a batch, to, for example, reduce repaints. Called when DataModel.updateAll() is called.
ORIGINATOR_NAVIGATE_ROW	The current row has been deleted and the originator of the deletion should now reposition the global cursor to a new, valid row.

The grid's events extend java.util.EventObject and are defined in subclasses of HiGridEvent. They are categorized by the group of class constants listed next.

HiGridEvent Subclass Constant	Description
AFTER_COLLAPSE_ROW	In HiGridExpansionEvent, the row node has been collapsed.
AFTER_CREATE_FORMAT_NODE_CONTENTS	In HiGridFormatNodeEvent, a new Format Node's contents has been created.
AFTER_CREATE_ROW	In HiGridUpdateEvent, the new row node has been created.
AFTER_EXPAND_ROW	In HiGridExpansionEvent, the row node has been expanded.
AFTER_MOVE_COLUMN	In HiGridMoveCellEvent, the column has been moved.
AFTER_RESET_FORMAT_DATA	In HiGridUpdateEvent, the format data has been reset.

HiGridEvent Subclass Constant	Description
AFTER_RESET_HIGRID_DATA	In HiGridUpdateEvent, the grid data has been reset.
AFTER_RESIZE_COLUMN	In HiGridResizeCellEvent, the column has been resized.
AFTER_RESIZE_ROW	In HiGridResizeCellEvent, the row has been resized.
AFTER_SELECT_ROW	In HiGridRowSelectionEvent, the row node has been selected or de-selected.
AFTER_SORT_TABLE	In HiGridSortTableEvent, the table has been sorted.
AFTER_TRAVERSE	In HiGridTraverseEvent, the current cell has been moved to a new row or column.
BEFORE_COLLAPSE_ROW	In HiGridExpansionEvent, the row node is being collapsed. A listener to this event can cancel the action.
BEFORE_CREATE_FORMAT_NODE_CONTENTS	In HiGridFormatNodeEvent, a new Format Node's contents is about to be created.
BEFORE_EXPAND_ROW	In HiGridExpansionEvent, the row node is being expanded. A listener to this event can cancel the action.
BEFORE_MOVE_COLUMN	In HiGridMoveCellEvent, the column is being moved.
BEFORE_REPAINT_ROW	In HiGridRepaintEvent, a row node is being repainted.
BEFORE_RESIZE_COLUMN	In HiGridResizeCellEvent, the column is being resized.
BEFORE_RESIZE_ROW	In HiGridResizeCellEvent, the row is being resized.
BEFORE_SELECT_ROW	In HiGridRowSelectionEvent, the row node is being selected or de-selected.
BEFORE_SORT_TABLE	In HiGridSortTableEvent, the table is being sorted.
ERROR	In HiGridErrorEvent, an error has been caught.

HiGridEvent Subclass Constant	Description
STATE_IS_INVALID	A HiGridValidateEvent.
VALUE_CHANGED_BEGIN	A HiGridValidateEvent.
VALUE_CHANGED_END	A HiGridValidateEvent.

Use the methods in the following table to glean information about the type of event that was generated. Since JClass HiGrid receives events from the data source as well, method isDataModelEvent permits you to find out whether the event comes from HiGrid itself or is simply being passed on. Use method isCancelable() to confirm that your application can actually intervene in the processing of the event. Most "BEFORE" events are cancellable.

HiGridEvent Method	Description
getEventType()	Retrieves the type of this event.
getException()	Retrieves the exception value of this event.
getRowNode()	Retrieves the RowNode associated with this event.
isCancelable()	Returns true if this event can be cancelled.
isDataModelEvent()	Determines if the event is actually a data model event.

2.6.2 Print Events

A HiGridPrintEvent is issued for each page when printing is in progress so that you can add custom headers and footers, and possibly do some post-processing when the print job finishes. The relevant constants are HiGridPrintEvent.PRINT_END, PRINT_FOOTER, and PRINT_HEADER.

2.6.3 Validating Events

HiGridValidateEvent's three constants are STATE_IS_INVALID, VALUE_CHANGED_BEGIN, and VALUE_CHANGED_END. JClass HiGrid's style of validation uses these methods to fire the events: fireValueChangedBegin, fireValueChangedEnd, fireStateIsInvalid, and fireValidateEvents. The method fireValidateEvents is in the support class (jclass.cell.ValidateSupport), since it is called by all JClass editors. Use HiGridValidateEvent.getValidateEvent to see which event subclass is involved.

2.6.4 Data Model Events

JClass HiGrid encapsulates data model events so that it can inspect all events before passing them on.

2.6.5 Mouse and Keyboard Events

The controller implements the following keystroke combinations:

Keystroke	Action
Enter	Completes the edit. The cell remains selected. If the cell is selected but no editor is active, the keystroke is ignored.
Esc (Escape)	Cancels the edit and replaces the contents of the cell with the original.
Up Arrow Key	With a cell in column *n* selected, moves up to column *n* in the next visible row. If the row contains fewer than *n* columns, moves to the last column in the row.
Down Arrow Key	Same as UP, but in the other direction
Left Arrow Key	Moves to the next cell on the left–does not wrap to the next row upon reaching the beginning of the current row.
Right Arrow Key	Moves to the next cell on the right–does not wrap to the previous row upon reaching the end of the current row.
Page Up	Replaces the current screen with the one that is conceptually just above it without overlapping. If there are too few rows in the new screen, begins at row one and shows a full screen.
Page Down	Replaces the current screen with the one that is conceptually just below it without overlapping. If there are too few rows in the new screen, ends at the last row and shows a full screen.
Alt+Page Up	Replaces the current screen with the one that is conceptually to its right, without overlaps – useful when the width of the grid is larger than the view.
Alt+Page Down	Replaces the current screen with the one that is conceptually to its left, without overlaps – useful when the width of the grid is larger than the view.
Home	Goes to the first cell on the current row.
Ctrl+Home	Goes to the first cell on the first row.
End	Goes to the last cell on the current row.
Ctrl+End	Goes to the last cell on the last row.
Tab	Moves to the adjacent cell on the right and wraps to the first cell on the row below when a TAB is received in the last cell on a row.

Keystroke	Action
Shift+Tab	Moves to the adjacent cell on the left and wraps to the last cell on the row above when a TAB is received in the first cell on a row.
Ctrl+Numeric+ (Plus sign on the number pad)	Opens the sub-level for the current row – if this level is already showing, do nothing.
Ctrl+Numeric- (Minus sign on the number pad)	Closes the sub-level for the current row – if this level is already closed, do nothing.

Mouse Actions

Besides traversing the grid by using the horizontal and vertical scrollbars, and selecting a cell for editing with a left mouse click, a right mouse click provides two popup menus, depending on the location of the mouse pointer when the event occurs. On a cell that is open for editing, the standard edit popup appears. On any other location in the grid, the JClass HiGrid edit popup menu appears.

Figure 18 JClass HiGrid's right-click popup menu.

2.7 Printing a Grid

Printing an image of the grid is facilitated by classes `PrePrintRowTreeWalk`, `PrePrintWalk`, `PrintGrid`, and `PrintWalk`.

The *Print* function is one of the choices in the popup menu that is accessed by a right-click anywhere within JClass HiGrid's grid area. There are two further choices, print *As Displayed* and print *As Expanded*. The latter choice prints the entire grid while the former prints just those rows that have been expanded. All rows of the root table are always printed, not just those that are visible in the grid window. If a sub-level is exposed anywhere and the `PRINT_AS_EXPANDED` option is chosen, all related sub-levels are printed.

You can access a limited number print routines programmatically as well. To initiate printing programmatically, use HiGrid's no-argument print method. Use setPrintFormat with either of the parameters HiGrid.PRINT_AS_DISPLAYED or HiGrid.PRINT_AS_EXPANDED to set the type of printing desired.

2.7.1 Printing Headers and Footers

To print grid information and include a page header and a page footer, add a print listener and process two of the three types of HiGridPrintEvent. These are PRINT_FOOTER, and PRINT_HEADER. The other case is PRINT_END, which signals the end of the print job. You can listen for this event if you want to do any clean-up after printing.

An example of code that might be used for a footer is included here. It is used by the print routine to produce the default footer. Assuming you have set the page margins elsewhere, the graphics context is used to get the clip bounds for the footer rectangle. You could replace the variable height/2 with some other value, but you must ensure that it is within the bounds defined by the rectangle or you won't see anything.

```
/**
 * Prints default footer.
 */
public void printPageFooter(HiGridPrintEvent event) {
   Graphics gc = event.getGraphics();
   Rectangle r = gc.getClipBounds();

   Font font = grid.getFont();
   if (font != null) {
       gc.setFont(font);
       String page = "Page " + event.getPage() +
           " of " + event.getNumPages();
       gc.drawString(page, 0, r.height/2);
   }
}
```

2.7.2 Print Preview

The EditPopupMenu includes a **Print Preview**. It also has the **As Displayed...** and **As Expanded...** sub-choices. Choosing one of these brings up a *Print* dialog, but clicking on **OK** causes a print preview window to appear rather than initiating an actual print job. An example of the *Print Preview* window is shown next. It contains buttons **First**, **Previous**, **Next**, and **Last** for selecting pages, and buttons **Print** and **Print All** remove the necessity of closing this window and then choosing **Print** to begin an actual print job. Use the **Print Page** button to print the current page, and use **Print All** to print the entire grid.

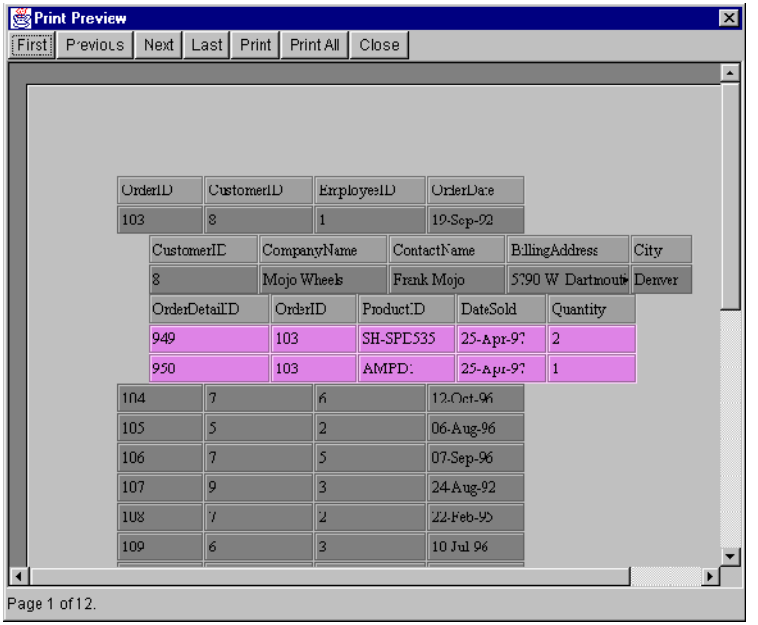


Figure 19 The Print Preview window, showing its navigation and control buttons.

2.7.3 PrintGrid and its Associated Classes

These classes are used internally by HiGrid to implement its print function. Apart from using the methodology of the previous section to provide customized headers and footers, or to do some clean-up at the end of a print job, no other use of these classes is recommended.

3

JClass HiGrid Beans

3.1 JClass HiGrid JavaBeans

The main Bean for HiGrid development is `JCHiGrid` Bean. With it, you can bind and configure the DataSource, and use the HiGrid portion as a viewer/editor on the data. With `JCHiGrid` Bean, the DataSource and HiGrid are tightly coupled as one Bean. The data model can be thought of as 'internal'.

The second Bean, `JCHiGridExternalDS`, is provided for developers who want to separate the DataSource from HiGrid, to have a data source that is 'external' to the HiGrid component. `JCHiGridExternalDS` is almost identical to `JCHiGrid` Bean, except that the data source features have been disabled. With `JCHiGridExternalDS` you can bind with an existing data source component, such as `TreeDataBean`.

This chapter covers using the `JCHiGrid` Bean as one of the components in your application. With the exception of data handling, the two HiGrid Beans are the same. Minor differences are discussed in Section 3.12, The JCHiGridExternalDS Bean.

This chapter demonstrates

- Placing the `JCHiGrid` Bean on a form (we'll use the BDK BeanBox as a generic example) and viewing the `JCHiGridCustomizer`'s "general" property page.
- Inspecting the `JCHiGridCustomizer`'s property sheet.
- Launching the `GridPropertiesEditor`'s custom editor.
- Configuring and saving a serialization file.
- Giving the grid's root table a name and defining the database connection.
- Adding the remaining tables and specifying the queries that select the desired columns and define the appropriate join statements.
- Determining which cells are editable.
- Declaring the commit policy for each level.

- Using the customizer to configure the visual aspects of the grid, such as font and color.
- Selecting which row types to include.
- Setting properties for cells on a per-column basis.

3.1.1 Placing the JClass HiGrid Bean on a Form

After you have installed its JAR file where the IDE you are using can find it, you can place the bean on a form. In the BeanBox, the *Properties - JCHiGrid Bean* window lists the top-level properties, including the `GridProperties`, which is a custom property editor for the description of the hierarchical design, the data binding, and the visual properties of the grid.

Single-click on its **Click to edit...** field and a modal dialog appears, reminding you to specify a name and location for the serialization file that will store the changes you make to the bean's default properties. Decide where the serialization file is to be located relative to the class loader in your distributed application. Type this path along with the name of the serialization file in the *Resource Name* field.

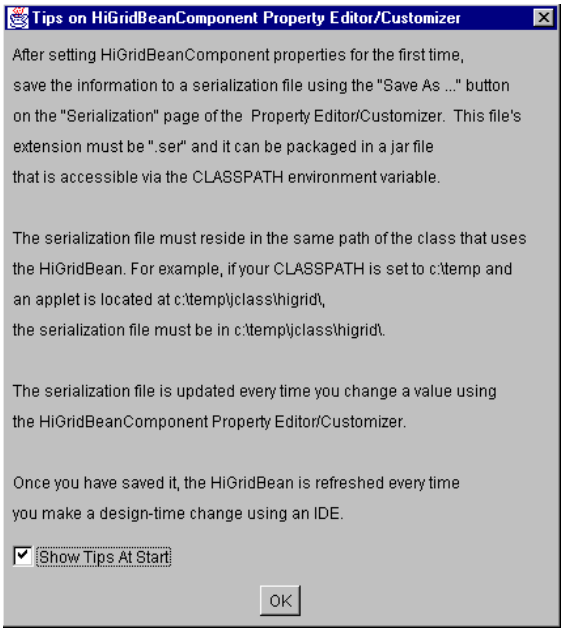


Figure 20 JClass HiGrid Bean's opening dialog.

Uncheck *Show Tips At Start* if you do not want this screen to reappear each time you place a new JClass HiGrid Bean component on the workspace, but be aware that if you change your directory, your IDE will not find the saved properties file and you will see this

screen again. This does not happen in the BeanBox because the startup directory never changes.

3.1.2 Using the HiGrid Bean in an IDE

JClass HiGrid is designed to be used in an IDE. Use HiGrid's powerful customizer to set up the database connection, build a query in a point-and-click fashion, and bind the retrieved data to the grid for display. The upcoming sections demonstrate the use of the customizer. Please refer to your IDE's documentation or the *JClass DesktopViews Installation Guide* for more information on integrating JClass HiGrid with an IDE.

3.2 Properties of JCHiGrid Bean

After you have placed the `JCHiGrid` Bean component on your form (which may be the BeanBox or one of the supported IDEs), you can inspect the main property sheet. It contains property editors for a number of global properties that either apply to the grid as a whole or are included at the top level as a convenient place for inspecting and possibly modifying their default values.

The following figure lists these properties and their default values.

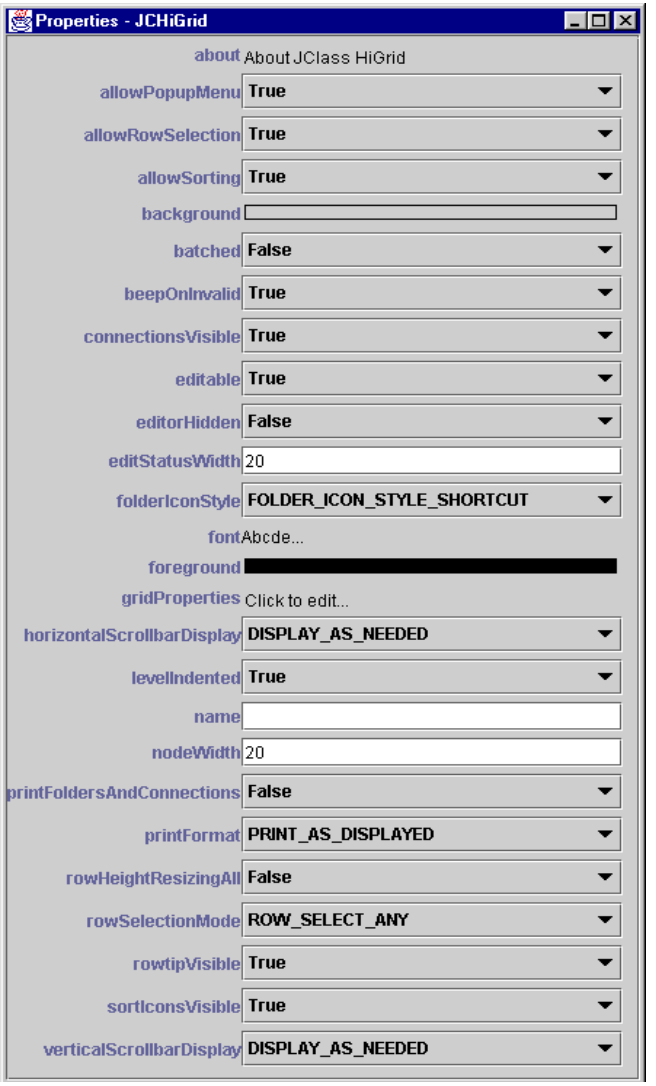


Figure 21 JCHiGrid's top-level properties and their default values.

The following table lists the properties and their descriptions. This is not the full list of properties. Nestled in the middle of the table is the property called `GridProperties` which contains an extensive custom editor. You use this customizer to specify the data source connection, define the meta data model, and set many more visual properties of

individual levels. You also use the customizer to manage the appearance of non-database-record rows, such as headers and footers.

You launch the customizer with a single click on the *GridProperties*' **Click to edit** field, as shown in the next figure.

Figure 22 Click on "Click to edit..." to start the Component Editor.

Note: A single click (mouse-down, mouse-up at the same place) is all that is required to bring up the customizer.

Property	Description
`about`	A String that identifies the product.
`allowPopupMenu`	Shows the edit popup menu if `true`.
`allowRowSelection`	If `false`, prevents rows from being selected.
`allowSorting`	If `false`, disables the sort capability.
`background`	The background color for the entire grid.
`batched`	Sets this mode if you want to prevent multiple updates to the graphical component. This command is usually issued programmatically, turning it on before performing large updates, then turning it back off.
`beepOnInvalid`	If `true`, causes a beep to sound when, during cell editing at run time, a validator decides that the input is invalid.
`connectionsVisible`	If `true`, shows the connecting lines between rows.
`editStatusWidth`	Sets the width of the edit status column.
`editorHidden`	Hides the editor's rectangle
`editable`	If `false`, the whole grid is read-only.
`folderIcon`	Chooses among various folder icon styles.
`font`	Any of the Java fonts may be selected
`foreground`	The foreground color for the entire grid.

Property	Description
gridProperties	This property invokes the customizer that lets you specify the database connection and define the meta data, along with setting many more visual properties. It is discussed in the next section.
horizontalScrollbarDisplay	Three options: as needed, always, or never.
levelIndent	The amount of indent in pixels to apply to sub-levels.
name	The name.
nodeWidth	The node width in pixels for all levels.
printFoldersAndConnections	If true, prints the folder icons and connection lines as well as the rows.
printFormat	There are two choices: as displayed or expand all levels and print.
rowHeightResizingAll	If true, then the user is allowed to resize a row anywhere on the row. If false, the row can be resized only in the EditStatus column.
rowSelectionMode	Determines if the specified selection mode allows the operation to continue. You may want to restrict a group of selections to the same table or at the same level.
rowTipVisible	If true, shows the row tip.
sortIconsVisible	Shows the sort icon when someone clicks on a column header, causing a sort operation.
version	This software's version number.
verticalScrollbarDisplay	Three options: as needed, always, or never.

3.3 Using the Customizer

The customizer provides a way of guiding you through the rather large number of options you have in setting up your grid. Use this JClass HiGrid JavaBean design-time element to specify the database connection and to set most of the visual properties of the grid. At run time, end-users modify the appearance of the grid in ways that were described in the previous chapter, like clicking and dragging, or using the edit popup menu.

You launch the customizer/custom editor as shown in Figure 22.

3.4 Overview of the Customizer's Functions

If you are using an IDE, you use the custom editor to configure the properties of the grid. With it, you can add or remove columns containing unbound data. (Header columns are a special case. They cannot be removed, but they can be made invisible.) You cannot delete bound columns because they are defined by the meta data, and can only be modified by changing it. You can specify **Before Details** and **After Details** formats, except at the root level. You may decide to change the meta data design while still in the development phase of your application. If this is the case, you can use the **Retrieve Columns** button to retrieve the new column information and continue using the customizer. The **Clear Format** button is used to start formatting all over again.

The customizer displays the class names for the cell renderer and cell editor objects that it will use automatically for the selected column. The choice is based on the column's data type, and is determined by the underlying database data type for that field. If you decide to define your own cell editor, you will have to enter its class name manually.

The custom editor's main areas are: a panel containing a **Data** tab and a **Format** tab. The data tab contains three main areas: **Data Source Type**, **Data Access**, and **JDBC**.

The tab structure is shown in the diagram. The contents of each panel is described in detail in the subsequent sections.

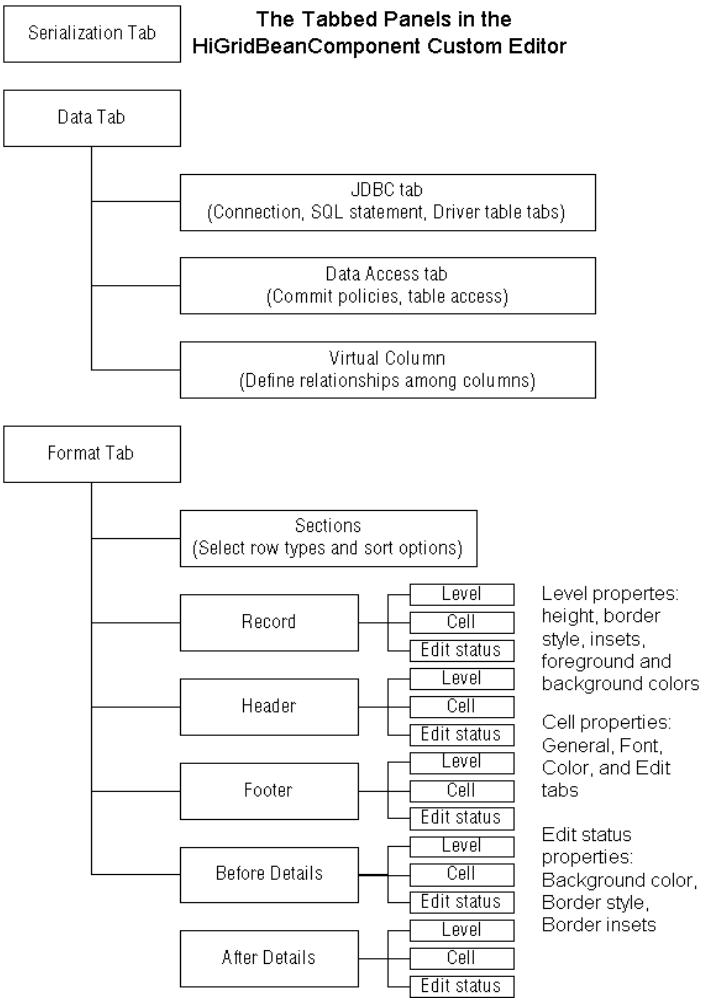


Figure 23 The GridProperties Custom Editor's tabbed dialogs.

3.5 The Serialization Tab

You begin the process by choosing a name and location for your serialization file, then clicking on the **Add** button in the meta data outliner on the left hand side of the Component Editor's main page.

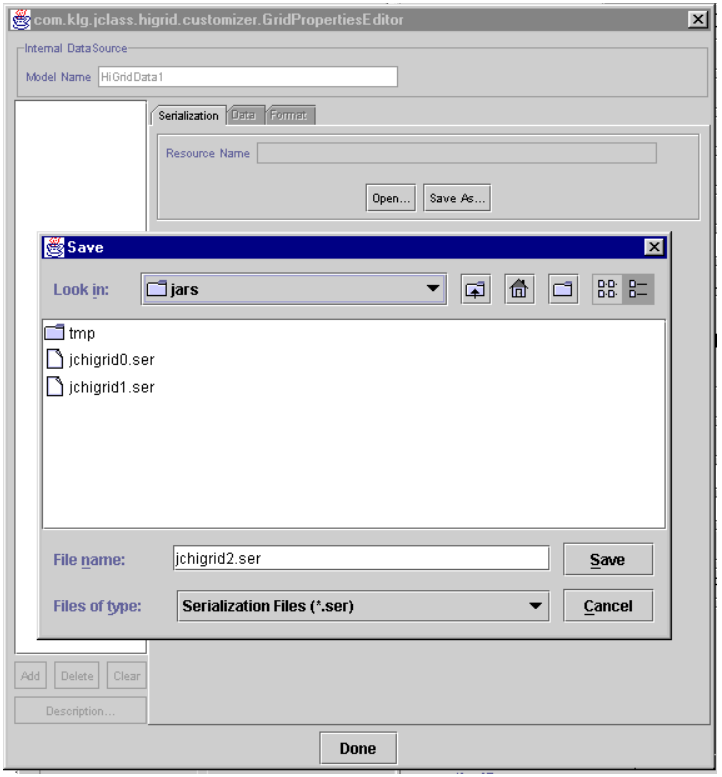


Figure 24 The GridPropertiesEditor's opening page.

1. Click the **Serialization** tab.
2. Click **Save As....**
3. A file dialog appears. A default filename is supplied, which you can change if you wish. Click **Save**.
4. Now the **Add** button is enabled. Click **Add** and give the root level data table a name.

Important: You need to name the root level of the data source's meta- data, or choose the default name, before you can make a database connection.

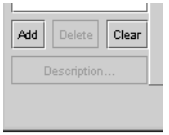


Figure 25 The Add button is at the lower left of the grid properties editor.

5. Click **OK**. The root table will have the name you give it, but you can change it at any time. This can be useful, since you may want to use an exact name from the database to which you are about to connect. You can revise the label for the data table after the connection is made and the table names are available by clicking the **Description...** button. We have chosen the name *Orders* for the data table. The name appears next to a folder icon in the left-hand panel, as shown in Figure 26. The connection tab is similar to the `JCData` and the `JCTreeData` that will be discussed later on. It is shown below.

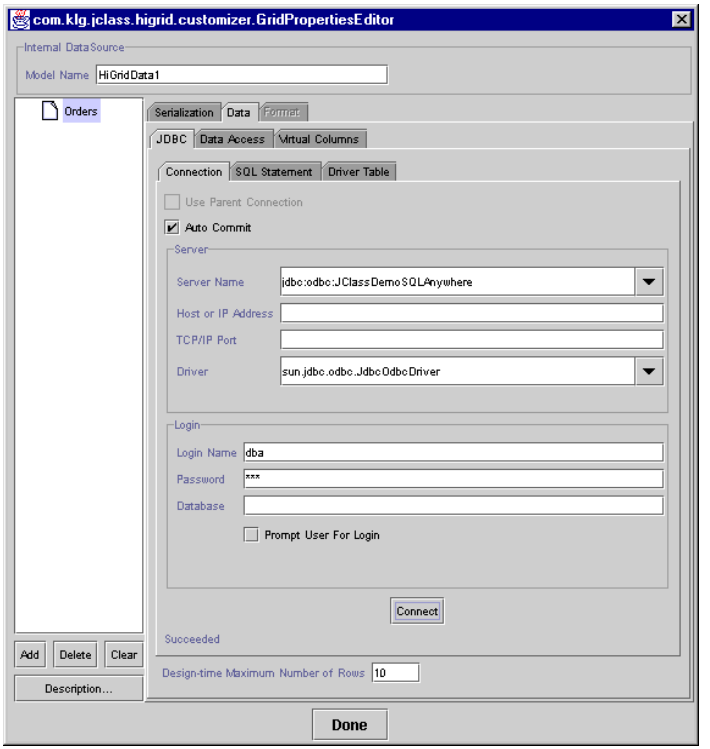


Figure 26 JClass HiGrid's connection panel.

The **Connection** panel is described both in the *JClass DesktopViews Installation Guide* and in the Data Bean chapter[1]. You supply the server name, the driver, and you give any additional information that may be necessary, such as a login name and password. If, as part of database connection URL, a host address and port number are required, they are specified as well.

3.6 Specifying the Data Sources

Up to this point we have not specified which of the database's tables are to be used. This is accomplished in the **SQL Statement** panel. A single level can comprise more than one table, but in this example we will use only the *Orders* table. You place a table in the top part of the SQL Statement panel either by clicking on the **Add Table** button or by right-clicking on the top panel itself. Clicking on **Add Table** brings up a *Table Chooser* dialog containing all the names, and the *Orders* table is chosen.

If you right-click on the top panel a popup menu appears:

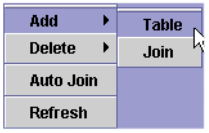


Figure 27 The popup menu resulting from a right click on the upper SQL Statement panel.

Click on **Add > Table** to choose the table name, then click the **Close** button on the *Table Chooser*. You'll see the table has been added to the panel, but one further step is required to make the addition permanent. This step is done after formulating the SQL statement.

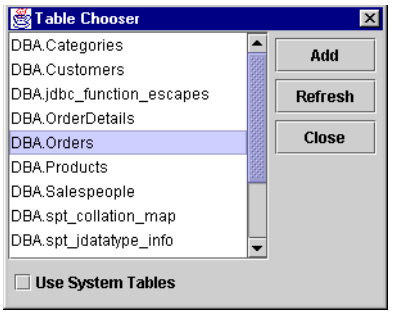


Figure 28 The Table Chooser dialog resulting from clicking Add Table.

1. In the IDE-specific cases, not as much customization can take place because the connection is not checked at design time, so table and column names are not made available to the customizer. Thus, the customizer cannot be used to format the grid.

The bottom panel, also called *SQL Statement*, is the text area that holds the actual SQL query. The next figure shows the table and a SQL statement. The customizer knows which table to select, but it does not make any assumptions about what columns of the table you want to display. This you do by pointing to a column name in the table to select it, then clicking on **Add Selected Column(s)**. At this point you have a valid SQL statement and you can pass this to the data model by pressing the **Set/Modify** button.

Now we'll add a detail level, called *OrderDetails*, using the same connection to the database as its parent.

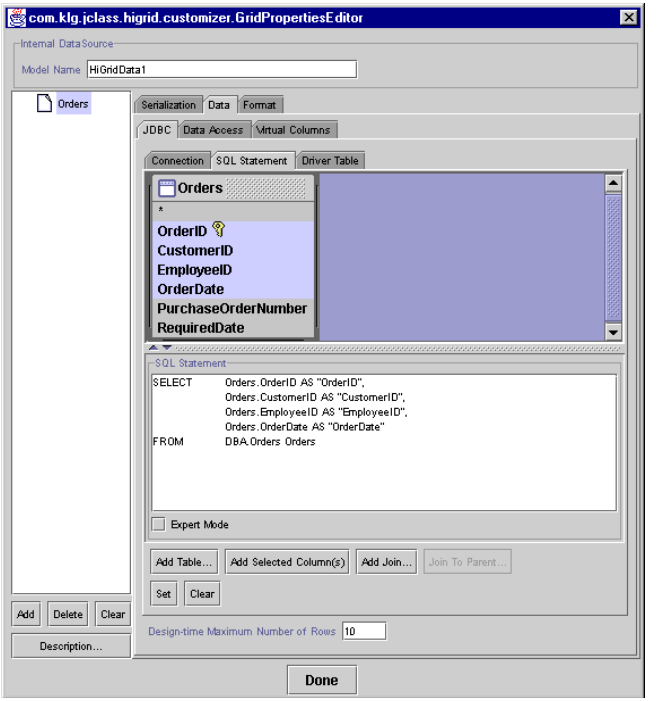


Figure 29 Setting the root-level data table.

3.6.1 Summary of the SQL Statement Page Buttons

Meta data design panel

Add – Adds a new entry to the design. The new entry appears as a child of the highlighted level.

Delete – Deletes the highlighted entry. All children of the highlighted node are deleted as well.

Clear – Clears the design. All levels are cleared and all formats are reset.

Description – Brings up an edit window containing the name assigned to the level.

SQL Statement Panel

Add Table... – Launches the *Table Chooser* window, containing a list of available database tables.

Add Selected Column(s) – After a table has been added to the *SQL Statement* panel, multiple fields can be chosen by dragging or control-clicking on them. Clicking on **Add Selected Column(s)** adds the columns name(s) to the SELECT clause of the SQL Statement. The same thing can be accomplished by double-clicking on a field name.

Add Join... – Invokes the *Join* dialog, whose buttons are **Add**, **Modify**, **Delete**, **Auto Join**, and **OK**. It simplifies construction of a WHERE clause when a level contains two or more tables

Join To Parent – Invokes the *Join To Parent...* dialog, whose buttons are **Add**, **Modify**, **Delete**, **Auto Join**, and **OK**. It simplifies construction of a WHERE clause joining parent-child tables.

Set – Sets the parameters that have been chosen for the current level.

Clear – Clears all settings for the current level.

Expert Mode – Only the *SQL Statement* text area is active. The customizer makes no attempt to parse or modify the query.

3.7 Joining Tables

A join matches common fields in two tables so that meaningful associations are formed. In the example shown, the *Orders* table has an *orderID* field that matches the one in *OrderDetails*. The result of forming a join on these fields is an association of an order with further detailed information about that order. Without the join, all *OrderID* rows would be listed in the sub-table that follows every *Orders* row and no useful information is conveyed.

There are three ways of specifying the join. You can type the comparison part of the *WHERE* clause that will specify the join directly into the edit box, you can choose the dependent table and the join column names from the drop down lists, or you can select the **Auto Join** feature. The parent table's field must be one of the selected fields for you to

be able to specify it in the join but the associated field in the dependent table need not be visible.

Figure 30 Joining the dependent table to its parent.

3.8 The Driver Table Panel

A level can be populated by more than one table. If it is, your application should specify which one is the primary table, that is, the one that will be used by the database to drive requeries. Use the *Driver Table* dialog shown in Figure 31 for this purpose.

3.9 Driver Limitations

Some JDBC drivers do not return a list of table names, therefore, they do not return a table's primary key. For these drivers, you must specify the primary key and those columns that are to be used when performing a requery. The *Join To Parent* dialog that has just been discussed does not permit you to choose the parent table. Instead, the edit field is disabled and is marked "<PARENT QUERY>", making it impossible to specify which parent table to use there. The *Driver Table* panel lets you specify the table to be used as the parent in the *Join To Parent* dialog. Here's the procedure:

1. Select the folder that you deem to be the driver table.
2. Select the **Driver Table** tab.
3. In the **Table** combo box, select the one you want to choose as the master table.
4. This table's primary key is shown in the text area.

As you would expect, the **Add** and **Delete** buttons are not active.

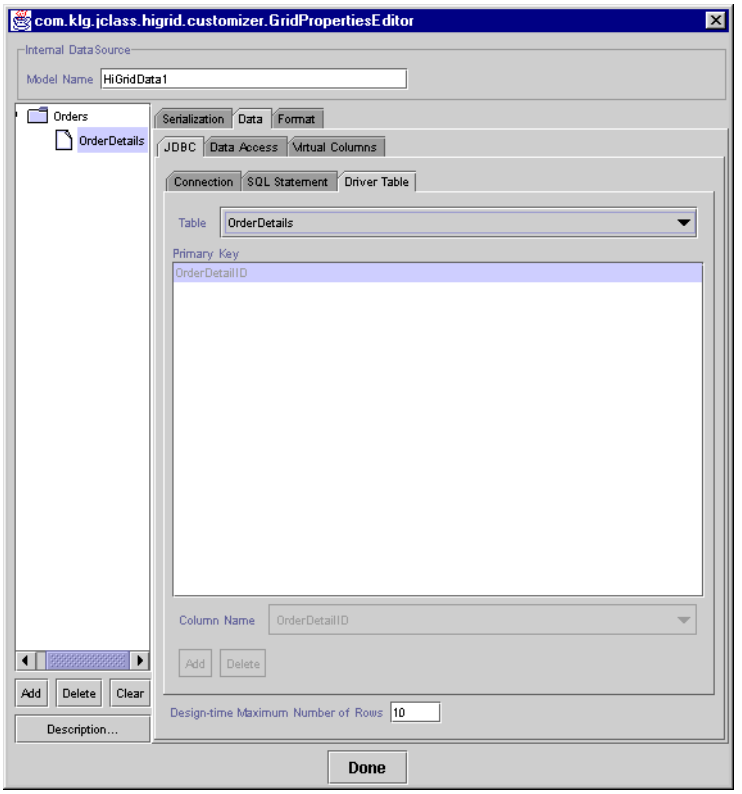


Figure 31 The Driver Table panel, resulting from selecting the Driver Table tab.

3.10 Setting Properties on the Format Tab

There are several actions that can be performed from the Format tab. The most important procedures are explained, including choosing row types, setting level properties, and setting column properties.

3.10.1 Choosing Row Types

Using the customizer makes it easy to choose which row types you want to include at any level.

1. Click on a level in the "Folder Tree" (the left hand panel in the `JCHiGrid` Bean Component Editor showing the meta data design) to select it.

2. Choose the **Format** tab, then the **Sections** tab.
3. Choose which of the four optional row types you want included.

You can also perform the following actions from this tab:

- Use the *Connections Foreground Color* panel to set the color of the connections lines in the *Folder Icons* column.
- Use the *Default Sort Data* panel to choose the column on which to sort, and the policy: either **Ascending**, **Descending**, or **None** (no attempt is made to sort the column).
- Click on *Current Level*>**Retrieve Fields** after you have modified the data design (the meta data) to refresh the list of column names in the currently selected level that appears under the **Record** row tab. Note that you will have to fill in the **Record**, **Header**, and so on, information again after using **Retrieve Fields**.
- Choose *Current Level*>**Clear Format** if you want to completely redo the data design (the meta data) at the current level and to refresh the list of table names that appears under the row tabs. Note that you will have to fill in the **Record**, **Header**, and so on, information again after using **Clear Format**.
- *All Levels*>**Retrieve Fields** is used after you have modified the data design (the meta data) to refresh the list of table names in the currently selected level that appear under the **Record** row tab. Note that you will have to fill in the **Record**, **Header**, and so on, information again after using **Retrieve Fields**.
- *All Levels*>**Clear Format** is used if you want to completely redo the data design (the meta data) at the current level. Note that you will have to fill in the **Record**, **Header**, and so on, information again after using **Clear Format**.

- A refresh mechanism propagates changes made at design-time to the grid so that it can update the view.

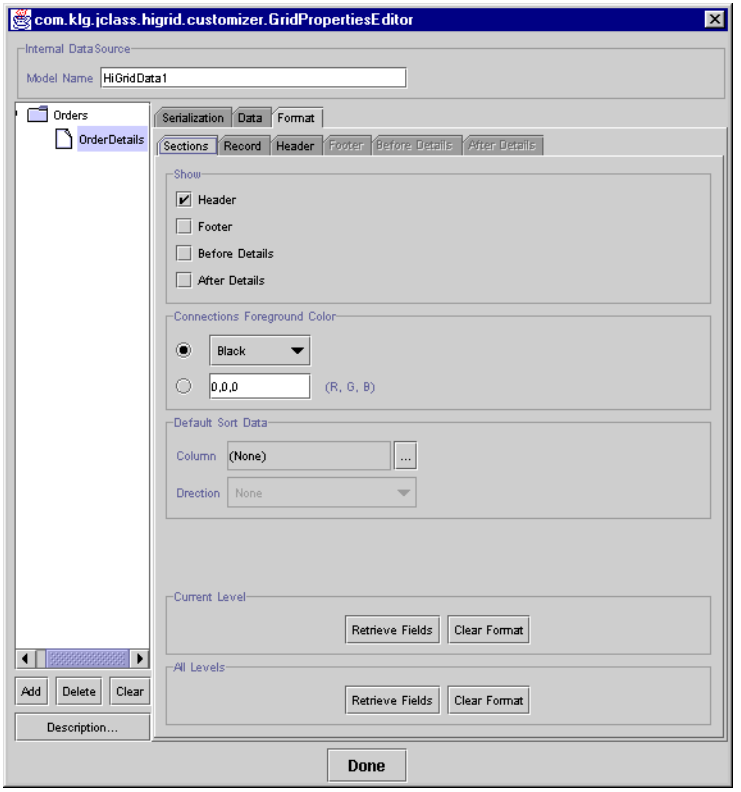


Figure 32 Choosing the types of rows to include.

3.10.2 Setting Level Properties

When you first attempt to choose the **Format** tab, you may see this error dialog:

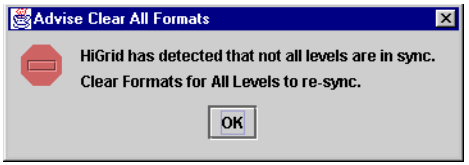


Figure 33 An error dialog that might appear on your first attempt to choose the Format tab.

If you encountered this error, you will have to clear all formats, start again, and reformat all levels.

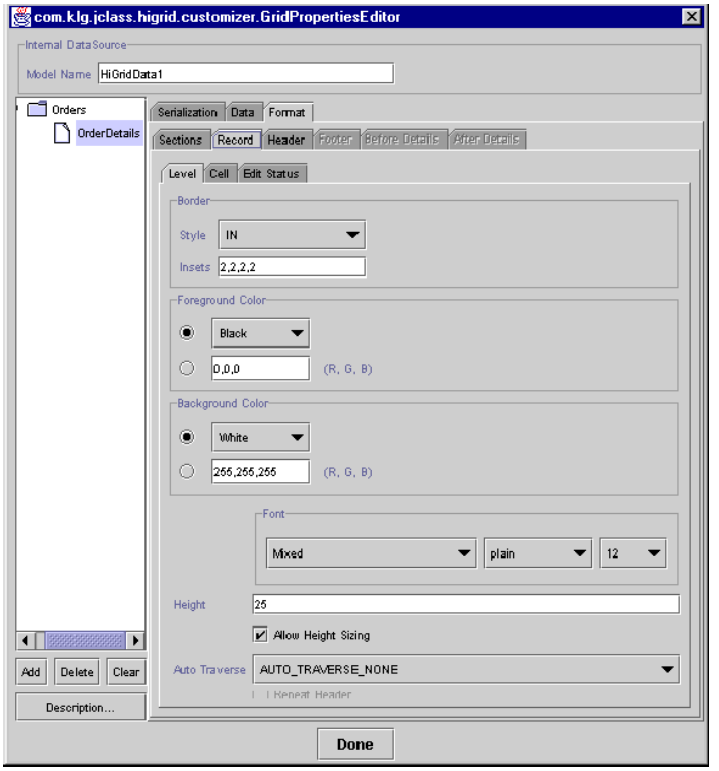


Figure 34 The General Level properties window where you can clear all formats.

Use this panel to set the properties of all rows of the selected type. The three panels contain these choices:

Border

- **Border Style** – Choose one from the drop-down list.
- **Border Insets** – Edit this text field to change their default values. As a general rule, an object appears well positioned in a cell when all border insets have the same value.

Background and Foreground Colors, and Fonts

- **Foreground Color** – Use the drop-down list to choose one of the standard colors defined in Java's Color class, or type in three comma separated RGB (red, green, blue) values in the lower text field.

- **Background Color** – Use the drop-down list to choose one of the standard colors defined in Java's Color class, or type in three comma separated RGB (red, green, blue) values in the lower text field.
- **Font** –Use the font chooser bar to select any font on your system.

Height

- **Height** – The height of the row in pixels.
- **Allow Height Sizing** – Ensure that this box is checked if you want to allow the end-user to adjust the height of this type of row. Note that allowing rows at one level to be resized is independent of the resizing policy at any other level.

3.10.3 Setting General Column Properties

There are four tabs for setting the numerous properties associated with cells. We'll discuss the ones that are settable on the **General** tab first. The properties relate to the general layout of the cell, that is, its size and the alignment of data within it.

Follow these steps to set the general cell properties for a single column.

1. Click on the *Cell* > **General** tab after having defined the grid's meta data.
2. Choose one of the meta data levels by clicking on it.
3. The text area to the left of the tab lists all the fields for this level. Choose the one whose properties you wish to set.

4. Edit any of the parameters that require it.

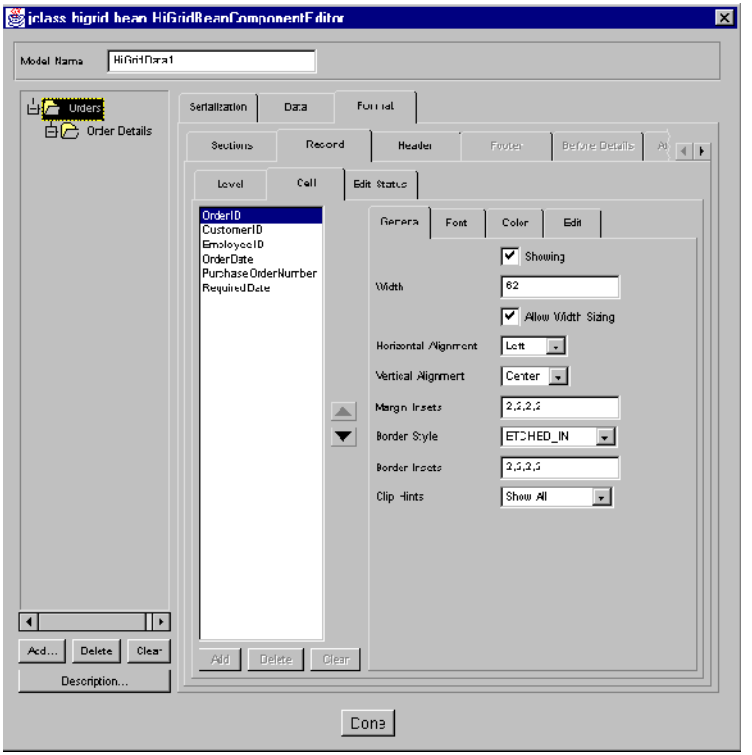


Figure 35 Cell properties that are settable using the General tab.

The General Tab

- **Showing** – If disabled, the column will not appear on the screen. One example of its usefulness is to hide a key field. If you need to join two tables based on a key, it must be mentioned in the SELECT clause for that table, which means it must be one of the fields retrieved from the database. Such a field is usually of no value to an end-user and should be hidden from view.
- **Text** – For headers, footers, before details, and after details rows, the column label.
- **Width** – The width of the column. Setting the width of any one cell sets the width for that column. Note that the height cannot be set at design time.
- **Allow Width Sizing** – If enabled, the column width may be resized at run time.
- **Horizontal Alignment** – Choices are **Top**, **Center**, **Bottom**
- **Vertical Alignment** – Choices are **Left**, **Center**, **Right**

- **Margin Insets** – The insets for the cells in a column.
- **Border Style** – See the section on Cell Formats and Cell Styles, in Chapter 2, for a list.
- **Border Insets** – See Border insets and margin insets., in Chapter 2.
- **Clip Hints** – Choices are **Show None**, **Show Horizontal**, **Show Vertical**, and **Show All**.

Setting a Column's Font Properties

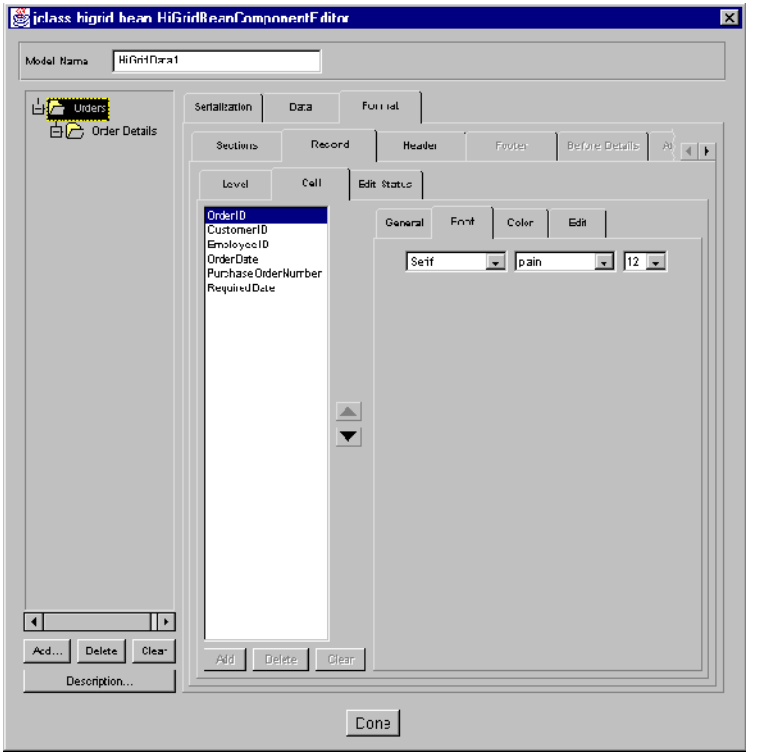


Figure 36 The Font tab where you can control the font's appearance.

- **Font** – The font for the column. The drop-down list contains the standard Java fonts.
- **Font Style** – Choices are **plain**, **bold**, and **italic**.
- **Font Size** – Font sizes up to 48 points are selectable.

Setting a Column's Color Properties

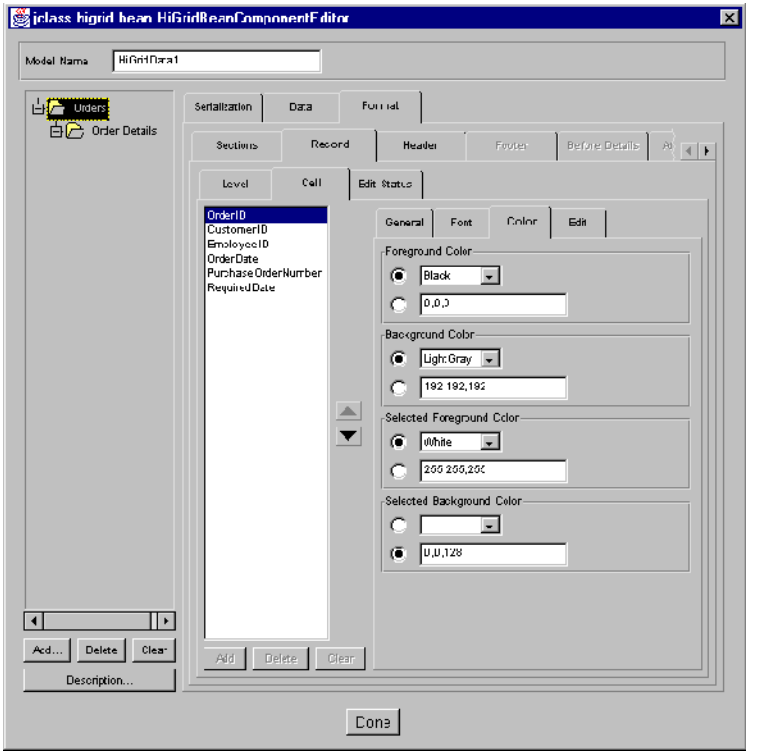


Figure 37 The Color tab, where you can set a column's color properties.

You can use the two types of color selectors to choose from Java's standard set of colors or you can specify RGB (red–green–blue) color values as a triplet of numbers between 0 and 255.

Color

- **Foreground Color** – The foreground color for the column.
- **Background Color** – The background color for the column.
- **Selected Foreground Color** – (Record rows only) The color to use when a cell has focus.
- **Selected Background Color** – (Record rows only) The color to use when a cell has focus.

A table of color values has been included in the Appendix to help you choose color values. See Appendix C, Colors and Fonts, for more information.

Setting a Column's Edit Properties

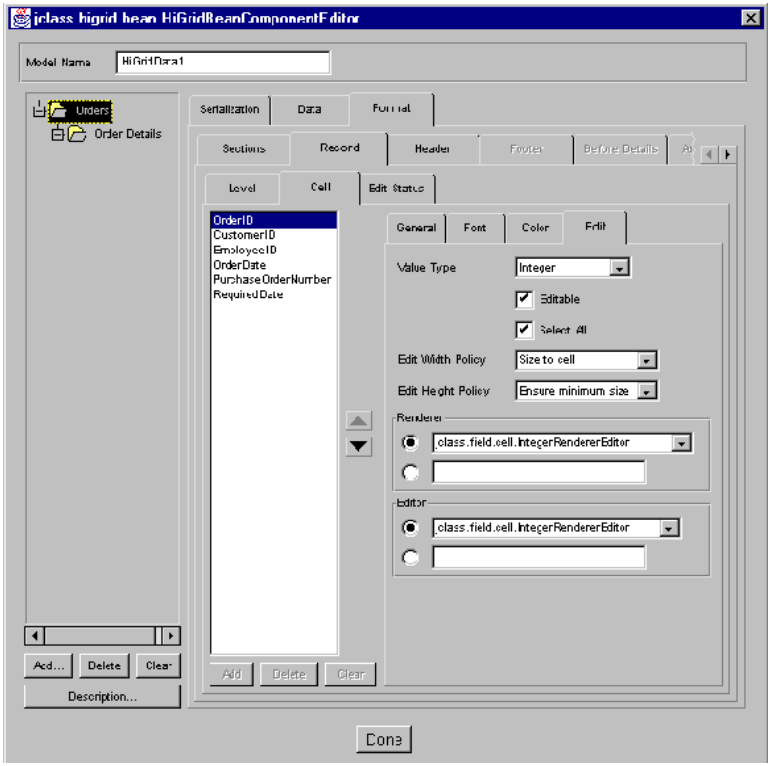


Figure 38 The Cell – Edit tab, where you can set a column's edit properties.

Editing

- **Value Type** – The data type for the column; it depends on the data type that the database defines for the column in question. Standard JDBC data types are allowed.
- **Editable** – The column may be specified as read-only by un-checking this checkbox.
- **Select All** – If true, the cell editor causes the entire field to be selected. Any keystroke replaces the entire field with that key. If false, the edit cursor is placed at the end of the field.
- **Edit Width Policy** – A cell editor's size is not necessarily the size of the cell itself. The options are `SIZE_TO_CELL` (the cell editor is fitted within the borders of the cell), `ENSURE_MINIMUM_SIZE` (the cell editor is sized to some predefined minimum size), and

ENSURE_PREFERRED_SIZE (the cell editor has a size that is appropriate for its type. For instance, if the cell contains a date and the editor is a calendar popup type, its preferred size is big enough to show all the days of the month.)

- **Edit Height Policy** – Options are the same as for *Edit Width*.
- **Renderer** – The cell renderer for the column's data type. Cell renderers are defined for all common database data types. You can define your own cell renderer if you wish. You would type its path name in the lower text field.
- **Editor** – The cell editor for the column's data type. Cell editors are defined for all common database data types. You can define your own cell editor if you wish. You would type its pathname in the lower text field. See Displaying and Editing Cells, in Chapter 4, for details.

3.11 Setting a Column's Edit Status Properties

All row types have an Edit Status cell so that the grid appears properly aligned, although only record rows use them to show whether edits have been made. If you change a row's appearance, and you want a consistent look, you need to match the edit status cell to the other changes. The properties that may need changing are:

- **Background Color** – The background color of the Edit Status cell.
- **Border Style** – Ten choices. For more information, please see Cell Formats and Cell Styles, in Chapter 2.

- **Border Insets** – The Edit Status cell's border insets. For more information, please see Cell Formats and Cell Styles, in Chapter 2.

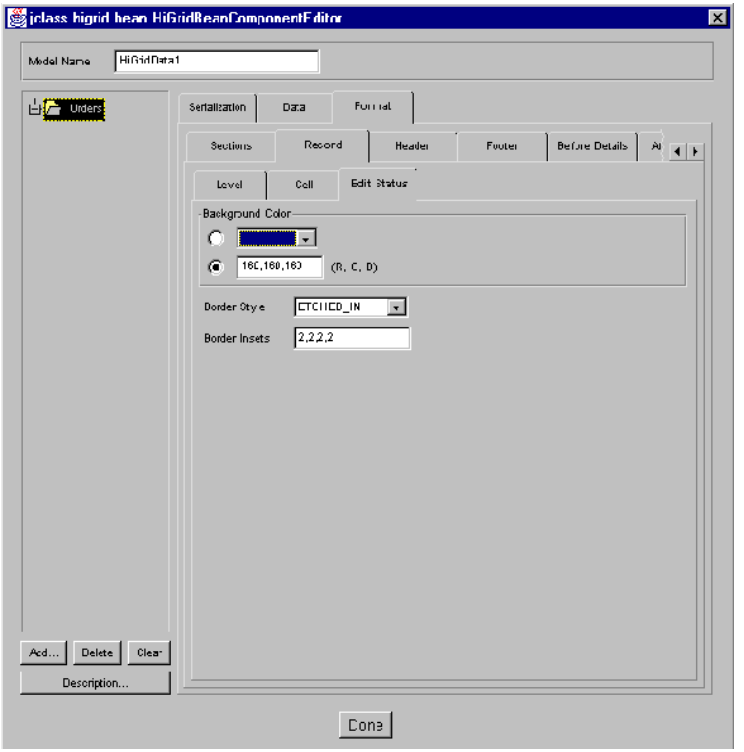


Figure 39 The Edit Status tab, where you can set a column's edit status properties.

3.12 The JCHiGridExternalDS Bean

With most features of the `JCHiGridExternalDS` Bean, the `JCHiGrid` Bean reference applies. However, there are some differences between the two Beans. `JCHiGridExternalDS` behaves more like other data bound components, such as JClass LiveTable, in that it does not manage the DataSource. It can access and modify data from different levels of the data hierarchy, but it cannot alter the DataSource structure or set-up. You will have to use a DataSource Bean, such as TreeDataBean, to set up your data source first.

In the `JCHiGridExternalDS` customizer you will notice that the **Data** tab, **Add, Delete**, **Clear**, and **Description** buttons have been 'grayed-out'. This means that you can't create SQL queries, or adjust the structure of the data model.

Instead of defining a DataSource, as in `JCHiGrid` Bean, you select a DataSource in the *external datasource* field, which has an editor that comes up when you click the **...** button:

Figure 40 The External DataSource Field. Click the ... button to bring up the editor.

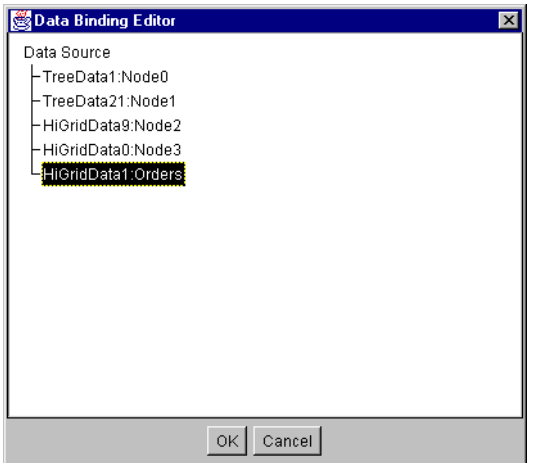


Figure 41 The DataBinding Editor, which allows you to select an external data source.

Select a data source from the list. Your data will display in `JCHiGridExternalDS` and you can configure the view on the data as you would with `JCHiGrid` Bean.

For information on configuring the DataSource and the data hierarchy, see JClass DataSource Beans, in Chapter 7.

4

Displaying and Editing Cells

4.1 Overview

JClass HiGrid offers a flexible way to display and edit any type of data contained in its cells. The following sections explain the techniques for displaying and editing cells in your programs.

In order to display a cell, JClass HiGrid has to know what type of data renderer is associated with the cell so it knows how to paint that data into the cell area. Similarly, in order for users to edit the cell values, HiGrid has to know what editor to return for that data type.

These operations are performed using the classes in the JClass cell package, which is structured as follows:

JClass Cell Package	Contents
`com.klg.jclass.cell`	Contains editor/renderer interfaces and support classes, including these interfaces: `JCCellEditor`: Used to define an editor. `JCCellRenderer`: The common and basic interface for renderers. `JCComponentCellRenderer`: Allows the creation of renderers that are based on `JComponent`. `JCLightCellRenderer`: Allows the creation of renderers based on direct drawing.
`com.klg.jclass.cell.editors`	Contains editors for common data types. Please see Section 4.4.1, Default Cell Editors, for details.
`com.klg.jclass.cell.renderers`	Contains renderers for common data types. Please see Section 4.3.1, JClass Cell Renderers, for details.

JClass Cell Package	Contents
com.klg.jclass.cell.validate	Contains data validation interfaces and support classes.

This JClass cell package is generic; renderers and editors written for JClass HiGrid will work with other JClass products. In addition, JClass Field components can work as renderers and editors within HiGrid, allowing very lightweight operation.

JClass HiGrid has been designed to identify the type of data being retrieved from the data source and to provide the appropriate cell renderer and cell editor for that data type. Often, however, you will want to control the way data in a particular area of the grid is rendered, or assign a specific type of editor for that data. An example of this is rendering String data in multiple lines and using `javax.swing.JTextArea` as the editor, rather than rendering and editing single-line Strings.

The following sections describe the techniques for rendering and editing cells by beginning with the easiest default methods, followed by detailed explanations for setting specific renderers and editors, mapping renderers and editors to a particular data type, and creating your own renderers and editors.

4.2 Default Cell Rendering and Editing

Basic Editors and Renderers

When the grid draws itself, it accesses the data source and attempts to paint the contents of each cell. In doing so, it:

1. Looks for a renderer and editor for that data in its list of default editors and renderers.
2. Assigns a renderer and an editor to each cell type. You can override this default mapping if you wish.

The following table lists the cell renderers and editors for common data types included with JClass HiGrid, which are found in the `com.klg.jclass.cell.renderers` and `com.klg.jclass.cell.editors` packages, respectively. When going through the above steps, JClass HiGrid uses these default mappings.

Data Type	Renderer	Editor
Big Decimal	JCStringCellRenderer	JCBigDecimalCellEditor
Boolean	JCCheckboxCellRenderer	JCCheckboxEditor
Byte	JCStringCellRenderer	JCByteCellEditor

Data Type	Renderer	Editor
Byte Array (for Images)	JCRawImageCellRenderer	JCImageCellEditor
Double	JCStringCellRenderer	JCDoubleCellEditor
Float	JCStringCellRenderer	JCFloatCellEditor
Integer	JCStringCellRenderer	JCIntegerCellEditor
Long	JCStringCellRenderer	JCLongCellEditor
Object	JCStringCellRenderer	JCStringCellEditor
Short	JCStringCellRenderer	JCShortCellEditor
SQL Date	JCStringCellRenderer	JCSqlDateCellEditor
SQL Time	JCStringCellRenderer	JCSqlTimeCellEditor
SQL Timestamp	JCStringCellRenderer	JCSqlTimestampCellEditor
String	JCStringCellRenderer	JCStringCellEditor
Util Date	JCStringCellRenderer	JCDateCellEditor

Although these editors and renderers are included with JClass HiGrid, you might find that you need more control over the way data is displayed and edited than simply relying on these defaults. The following sections explain cell rendering and cell editing in detail.

4.3 Rendering Cells

Cell rendering is simply the way in which data is drawn into a cell. JClass HiGrid includes renderers that you can use in your grid. Additionally, two rendering models, `JCLightCellRenderer` and `JCComponentCellRenderer`, are provided if you want to create your own renderer. Each model caters to different rendering needs.

More information about included renderers are found in the next section, and information about the two rendering models on which you can base customized renderers is found in Section 4.3.3, Creating your own Cell Renderers.

4.3.1 JClass Cell Renderers

As shown in the table above, JClass HiGrid maps standard data types to specific renderers when the program does not specify a renderer for that data type. This means

that most grids are easily rendered without any special coding. The renderers are internally assigned.

Name	Data Type	Description
JCCheckBoxCellRenderer	Boolean	Defines a JCComponentCellRenderer object that paints Boolean objects in a grid cell using Swing's JCheckBox.
JCComboBoxCellRenderer	integer	Defines a JCComponentCellRenderer that paints integer objects in a grid using Swing's JComboBox.
JCImageCellRenderer	image	Defines a JCLightCellRenderer object that paints Image objects in a grid cell.
JCLabelCellRenderer	String	Defines a JCLabelCellRenderer object that uses Swing's JLabel to render cell contents
JCRawImageCellRenderer	image	Defines a JCLightCellRenderer object that paints unconverted Image objects in a grid cell (extends JCScaledImageCellRenderer)
JCScaledImageCellRenderer	image	Defines a JCLightCellRenderer object that paints scaled Image objects in a grid cell.
JCStringCellRenderer	String, Boolean, double, float, integer, object.	Defines a JCLightCellRenderer object that can draw Strings.
JCWordWrapCellRenderer	String	Defines word-wrapping logic for multi-line display of Strings in cells.

The default mappings and these special renderer classes should provide rendering for most data types. Few programmers work under ideal conditions, however, and you may need to extend the capability of these renderers. JClass HiGrid includes ways for you to customize cell rendering, as described in Section 4.3.3, Creating your own Cell Renderers.

4.3.2 Mapping a Data Type to a Cell Renderer

Because there are many different data types within a row, JClass HiGrid creates a *mapping* between data types and cell renderers. The mapping takes a data type and associates it with a cell renderer; whenever the container encounters that type of data, it uses the

mapped JCCellRenderer. This mapping is performed automatically for standard JDBC data types.

Mapping a JCCellRenderer object to a data type takes the following construction, where cellType is the cell renderer and thisCell is the current cell:

```
thisCell.CellFormat.setCellRendererName(String cellType);
```

Normally, you would use these mappings in a construction that would test for the presence of the renderer you specify, and throw an exception if the class was not found.

It is possible to use a new feature of the com.klg.jclass.cell package, called the EditorRendererRegistry, to associate a data type with an editor and a renderer. Imagine a case where you have a column of Boolean quantities. You wish to allow your users to edit a cell by typing a zero or a one, but you wish to display the result in a checkbox. The following code associates the desired editor and renderer with the underlying data type:

```
EditorRendererRegistry.getCentralRegistry().addClass(
    "java.lang.Integer",
     null,
    "com.klg.jclass.cell.editors.JCIntegerCellEditor",
    "com.klg.jclass.cell.renderers.JCCheckBoxCellRenderer");
```

JClass HiGrid will now use these editors/renderers for integer types.

Note however that a mapping via the central registry is global, in the sense that it will be used by all the JClass components in your application that use editors and renderers.

See the *JCLASS_HOME/examples/higrid/data* example for changing editors and renderers on a per-column basis. To see how to override the default associations between data types and editors/renderers, see EditorRendererExample in the *JCLASS_HOME/examples/higrid/visual* directory.

4.3.3 Creating your own Cell Renderers

Naturally, the renderer classes provided with JClass HiGrid will not meet every programmer's specific needs. However, they can be convenient as bases for creating your own renderer objects by subclassing the original classes. If you want to create your own renderer classes, you can build your own renderer from scratch. Both techniques are discussed below.

Subclassing the Default Renderers

A simple way to create your own renderer objects is to subclass one of the renderers provided with JClass HiGrid. For example, *CurrencyRenderer.java* is an example of

subclassing from the `JCStringCellRenderer` in the `com.klg.jclass.cell.renderers` package:

```
import com.klg.jclass.cell.renderers.JCStringCellRenderer;
import com.klg.jclass.cell.JCCellInfo;

import java.awt.Graphics;

public class CurrencyRenderer extends JCStringCellRenderer {

public void draw(Graphics gc, JCCellInfo cellInfo,
                 Object o, boolean selected) {
    if (o instanceof Double) {
        double d = ((Double)o).doubleValue();
        o = formatLabel(d, 2);
    }
    super.draw(gc, cellInfo, o, selected);
}
```

Creating a Drawing-based Cell Renderer with JCLightCellRenderer

One way JClass HiGrid lets you write your own cell renderer is with `JCLightCellRenderer`. This model is used for drawing directly into a cell, which is ideal for custom painting and rendering text.

To create a drawing-based renderer object of your own, you must implement `com.klg.jclass.cell.JCLightCellRenderer`:

```
public interface JCLightCellRenderer {
public void draw(Graphics gc, JCCellInfo cellInfo, Object o, boolean
                 selected);
public Dimension getPreferredSize(Graphics gc, JCCellInfo cellInfo,
                 Object o);
}
```

The `JCLightCellRenderer` interface requires that you create two methods:

- A `draw()` method, which is passed a `JCCellInfo` object (see Section 4.5, The JCCellInfo Interface, for more details) containing information from the container about the cell, a `java.awt.Graphics` object, and the object to be rendered. The `Graphics` object is positioned at the origin of the cell (0,0), but is not clipped.
- A `getPreferredSize()` method, which is used to allow the renderer to influence the container's layout. The container may not honor the renderer's request, depending on a number of factors.

The following code, *TriangleCellRenderer.java*, draws a triangle into the cell area:

```
import java.awt.Polygon;
import java.awt.Dimension;
import java.awt.Graphics;
import java.awt.Rectangle;
import com.klg.jclass.cell.JCCellInfo;
import com.klg.jclass.cell.JCLightCellRenderer;

public class TriangleCellRenderer implements JCLightCellRenderer {

public void draw(Graphics gc, JCCellInfo cellInfo,
                 Object o, boolean selected) {
    Polygon p = makePolygon(o);
    gc.setColor(selected ? cellInfo.getSelectedForeground()
        :cellInfo.getForeground());
    gc.fillPolygon(p);
}

public Dimension getPreferredSize(Graphics gc, JCCellInfo cellInfo, Object
o) {
    // Make a polygon from the object
    Polygon p = makePolygon(o);
    // Return no size if no polygon was created
    if (p == null) {
        return new Dimension(0,0);
    }
    // Bounds of the polygon determine size
    Rectangle r = p.getBoundingBox();
    return new Dimension(r.x+r.width,r.y+r.height);
}

private Polygon makePolygon(Object o) {
    if (o == null) return null;
    if (o instanceof Number) {
        return makePolygon(((Number)o).intValue());
    }
    else if (o instanceof Polygon) {
        return (Polygon)o;
    }
    return null;
}


public Polygon makePolygon(int s) {
    Polygon p = new Polygon();
    p.addPoint(0,0);
    p.addPoint(0,s);
    p.addPoint(s,0);
    return p;
}
}
```

The above program creates a triangle renderer object that can handle both `Integer` and `Polygon` objects.

As required by JCCellRenderer, the program contains a draw() method in the lines:

```
public void draw(Graphics gc, JCCellInfocellInfo, Object o boolean
                selected) {
Polygon p = makePolygon(o);
gc.getColor(selected ? cellInfo.getSelectedForeground():
                cellInfo.getForeground());
gc.fillPolygon(p);
}
```

The draw() method renders the object o by making it into a polygon and drawing the polygon using the gc provided. The grid, as the container, automatically translates and clips the gc, draws in the background of the cell, and sets the foreground color.

The parameter cellInfo can be used to retrieve other cell property information through the JCCellInfo interface (see Section 4.5, The JCCellInfo Interface).

The second required method, getPreferredSize(), is provided in the lines:

```
public Dimension getPreferredSize(Grahpics gc, JCCellInfo cellInfo,
                                  Object o) {
Polygon p = makePolygon(o);
if (p == null) {
    return new Dimension(0,0);
}
Rectangle r = p.getBoundingBox();
return new Dimension(r.x+r.width,r.y+r.height);
}
```

Here, the object is used to create a polygon (using a local method called makePolygon()). If it doesn't create a polygon from the object, the object is deemed to have no size (0,0) and will not be displayed by the renderer. If a polygon was created from the object, the polygon's bounds determine the size of the rectangle in the drawing area of the cell. The size returned is only a suggestion; control of the cell size can be overridden by the grid container.

Creating a Component-based Cell Renderer with JCComponentCellRenderer

While JCLightCellRenderer is useful for drawing directly into cells (i.e. text rendering and custom cell painting), it is a cumbersome model to use if you want to draw a component as part of an editor/renderer pair. For example, if you wanted to use a dropdown list in a grid cell, creating a renderer based on JCLightCellRenderer forces you to write the code that draws the arrow button. Obviously, it is more desirable to use the actual code for the component itself, for which is exactly what JCComponentCellRenderer is best suited.

Component-based cell renderers use an existing lightweight component for rendering the contents of a cell. As such, the JCComponentCellRenderer interface can be used to create a component-based cell renderer:

```
public interface JCComponentCellRenderer extends JCCellRenderer {
public Component getRendererComponent(JCCellInfo cellInfo, Object o,
           boolean selected);
}
```

The `getRendererComponent` returns the component that is to be used to render the cell. It is the responsibility of the implementor to use the information provided by `getRendererComponent` to set up the component for rendering:

- `cellInfo` contains information from the container about the cell (see Section 4.5, The JCCellInfo Interface for more details).
- `o` is the object to be rendered.
- `selected` is a Boolean indicating whether the cell is selected. Many implementors use this information to modify the component appearance.

As an example, consider *JCLabelCellRenderer.java* from `com.klg.jclass.cell.renderers`, which uses a Swing `JLabel` for rendering String data.

```
import com.klg.jclass.cell.JCComponentCellRenderer;
import com.klg.jclass.cell.JCCellInfo;
import javax.swing.JLabel;
import javax.swing.JComponent;

public class JCLabelCellRenderer extends JLabel
implements JCComponentCellRenderer {

public JCLabelCellRenderer() {
   super();
}

public JComponent getRendererComponent(JCCellInfo cellInfo,
                                       Object o,boolean selected) {
   if (o != null) {
       if (o instanceof String) {
           setText((String)o);
       }
       else {
           setText(o.toString());
       }
   }
   else {
       setText("");
   }
   setBackground(cellInfo.getBackground());
   setForeground(cellInfo.getForeground());
   return this;
   }
   }
```

In this example, note that `JCLabelCellRenderer` extends `JLabel`, which makes it easier for the renderer to control the label's appearance.

In `getRendererComponent()`, object `o` is converted to a String and used to set the `Text` property of the label. Then, the font, foreground color, and background color are extracted from the `cellInfo`. Finally, the `JLabel` instance is passed back to the container.

`JCComponentCellRenderer` is a very powerful rendering model. While it is not as flexible as `JCLightCellRenderer`, it allows the reuse of code by using a lightweight component as

a rubberstamp for painting in a cell. Any existing lightweight container can be used to render data inside of a cell – even other JClass components.

4.4 Editing Cells

While rendering cells is fairly straightforward, handling interactive cell editing is considerably more complex. Cell editing involves coordinating the user-interactions that begin and end the edit with cell data validation and connections to the data source. In JClass, cell editing is handled using the `JCCellEditor` interface.

A typical cell edit works through the following process:

- The container listens for events that come from the editor by implementing `JCCellEditorListener`.
- When a user initiates a cell edit with either a mouse click or a key press, the container calls `JCCellEditor.initialize()` and passes a `JCCellInfo` object with information about the cell, and the object (data) that will be edited.
- The `JCCellEditor` displays the data and changes it according to user input.
- If the user traverses out of the cell, then the *container* calls the `stopCellEditing()` method, which asks the `JCCellEditor` to validate the edit. If the edit is not valid (that is, `stopCellEditing()` returns `false`) the container then retrieves the original cell value from the data source. If the edit is valid, then the container calls `getCellEditorValue()` on the editor to retrieve the new value of the cell and send it to the data source.
- If the user types a key that the editor interprets as "done" (for example, **Enter**), the editor will inform the grid that the edit is complete by sending an `editingStopped` event to the grid. Typical editors will validate the user's changes before sending the event.
- If the user types a key that the editor interprets as "cancel" (for example, **Esc**), the editor will instruct the grid to cancel the edit by sending an `editingCanceled` event.

Because cell editing has been designed to be flexible, you can have as little or as much control over the editing process as you want. The following sections explain cell editing in further detail.

4.4.1 Default Cell Editors

The following editors are provided in the `com.klg.jclass.cell.editors` package:

Editor	Description
`BaseCellEditor`	Provides a base editing component for other editors.

Editor	Description
JCBigDecimalCellEditor	An editor using a simple text field for BigDecimal objects.
JCBooleanCellEditor	Provides a simple text editing component that allows the user to set the Boolean value as either 'true', 'false', 't' or 'f'.
JCByteCellEditor	An editor using a simple text field for Byte objects.
JCCheckBoxCellEditor	An editor for Boolean data that automatically changes the checked state.
JCComboBoxEditor	An editor using a simple Swing JComboBox for editing an enum.
JCDateCellEditor	An editor using a simple text field for Date objects
JCDoubleCellEditor	An editor using a simple text field for Double objects.
JCFloatCellEditor	An editor using a simple text field for Float objects.
JCImageCellEditor	An editor using a simple text field for Image objects.
JCIntegerCellEditor	An editor using a simple text field for Integer objects.
JCLongCellEditor	An editor using a simple text field for Long objects.
JCMultilineCellEditor	A simple text editing component for multiline data.
JCShortCellEditor	An editor using a simple text field for Short objects.
JCSqlDateCellEditor	An editor using a simple text field for SQL Date objects.
JCSqlTimeCellEditor	An editor using a simple text field for SQL Time objects.
JCSqlTimestampCellEditor	An editor using a simple text field for SQL Timestamp objects.
JCStringCellEdtitor	Provides a simple text editing component.
JCWordWrapCellEditor	Provides a simple text editing component that wraps text.

While these classes provide editing capability for most data types, many real-world situations require greater control over cell editing, editing components, and their relationships to specific data types. The following sections explore how you can more minutely control the cell editing mechanism in your programs.

4.4.2 Mapping a Data Type to a Cell Editor

It is likely that your grid is designed in such a way that there are many different data types within a row. In this case HiGrid creates a *mapping* between data types and cell editors. The mapping takes a data type and associates it with a cell editor; whenever the container encounters that type of data, it uses the mapped `JCCellEditor`.

Mapping a `JCCellEditor` object to a data type takes the following construction, where `cellType` is the cell editor and `thisCell` is the current cell:

```
thisCell.CellFormat.setCellEditorName(String cellType);
```

Normally, you would use these mappings in a construction that would test for the presence of the editor you specify, and throw an exception if the class was not found.

4.4.3 Creating Your Own Cell Editors

To create a cell editor object, you must implement the `com.klg.cell.JCCellEditor` interface. The following code comprises the `JCCellEditor` interface:

```
public interface JCCellEditor extends JCCellEditorEventSource,
    serializable{
public void initialize(AWTEvent ev, JCCellInfo info, Object o);
public Component getComponent();
public Object getCellEditorValue();
public boolean stopCellEditing();
public boolean isModified();
public void cancelCellEditing();
public JCKeyModifier[] getReservedKeys();
}
```

Consider each of the methods in `JCCellEditor`:

Method and Description
`public void initialize(AWTEvent ev, JCCellInfo info, Object o);`
The table calls `initialize()` before the edit starts to let the editor know what kind of event started the edit, using `java.awt.AWTEventObject`. The size of the cell comes from the `JCCellInfo` interface (detailed below). The `initialize()` method also provides the data object `(Object o)`.
`public Component getComponent();`
Returns the AWT component that does the editing. The component should be lightweight.
`public Object getCellEditorValue();`
Returns the value contained in the editor. This method is called by the table when the edit is complete. The value will be sent to the data source.

Method and Description
`public boolean stopCellEditing();` When this method is called by the table, the editor can refuse to commit invalid values by returning `false`. This tells the container that the edit is not valid.
`public boolean isModified();` The container uses this method to check whether the data has changed. This can save unnecessary access to the data source when the data has not actually changed.
`public void cancelCellEditing();` Called by the table to stop editing and restore the cell's original contents.
`public JCKeyModifier[] getReservedKeys();` Retrieves the keys the editor would like to reserve for itself. In order to avoid the container overriding key processing in the editor, the editor can pass back a list of keys it wishes to reserve. The container can refuse the editor's request to reserve keys. Most editors can simply return null for this method.

Because the `JCCellEditor` interface extends `JCCellEditorEventSource`, the following two methods are required to manage `JCCellEditor` event listeners:

Method and Description
`public abstract void addCellEditorListener(JCCellEditorListener l);` Adds a listener to the list that's notified when the editor starts, stops, or cancels editing.
`public abstract void removeCellEditorListener(JCCellEditorListener l);` Removes the listener.

In addition to implementing the methods of `JCCellEditor`, an editor is responsible for monitoring events and sending `editingStopped` and `editingCanceled` events to the grid. This functionality is further explained in Section 4.4.3, Creating Your Own Cell Editors.

Subclassing the Default Editors

One easy way to create your own editor is to subclass one of the editors provided in the `com.klg.jclass.cell.editors` package. The following code creates a simple editor that extends the `JCStringCellEditor` class. The `MoneyCellEditor` class formats the data as money (two digits to the right of the decimal point) instead of a raw String; but `JCStringCellEditor` does most of the work.

The `initialize()` method in `MoneyCellEditor` takes the object passed in and creates a Money value for it. The `getCellEditorValue()` method will pass the Money value back to the container.

```
import java.awt.Dimension;
import com.klg.jclass.cell.editors.JCStringCellEditor;
import com.klg.jclass.cell.JCCellInfo;
import java.awt.AWTEvent;

public class MoneyCellEditor extends JCStringCellEditor {

Money initial = null;

public void initialize(AWTEvent ev, JCCellInfo info, Object o) {
    if (o instanceof Money) {
        Money data = (Money)o;
        initial = new Money(data.dollars, data.cents);
    }
    super.initialize(ev, info, initial.dollars+"."+initial.cents);

}

public Object getCellEditorValue() {
    int d, c;
    String text = getText().trim();
    Money new_data = new Money(initial.dollars, initial.cents);

    try {
        // one of these will probably throw an exception if
        // the number format is wrong
        d = Integer.parseInt(text.substring(0,text.indexOf('.')));
        c = Integer.parseInt(text.substring(text.indexOf('.')+1));

        new_data.setDollars(d);
        // this will throw an exception if there's an invalid
        // number of cents
        new_data.setCents(c);
    }
    catch(Exception e) {
        return null;
    }

    return new_data;
}

public boolean isModified() {
    if (initial == null) return false;
    Money nv = (Money)getCellEditorValue();
    if (nv == null) return false;
    return (initial.dollars != nv.dollars || initial.cents != nv.cents);
}
}
```

Starting with one of the cell editors provided with the `com.klg.cell.editors` package can save you a lot of work coding entire editors on your own.

Writing Your Own Editors

Of course, you may not want to subclass any of the editors provided with the `com.klg.jclass.cell.editors` package. The following code fragment is from an editor that was written without subclassing an existing editor. By implementing the `JCCellEditor` interface, we have written an editor that will edit triangles. The editor handles both `Integer` and `Polygon` data types. It initializes the editor with the object to be edited, either a `Number` or a `Polygon`:

```
....

public void initialize(AWTEvent ev, CellInfo info, Object o) {
  if (o instanceof Polygon) {
    orig_poly = (Polygon)o;
  }
  else if (o instanceof Number) {
    // Create polygon from the number
    int s = ((Number)o).intValue();
    orig_poly = new Polygon();
    orig_poly.addPoint(0,0);
    orig_poly.addPoint(0,s);
    orig_poly.addPoint(s,0);
  }

  new_poly = null;

  margin = info.getMarginSize();
}
```

The editor also needs to retrieve the AWT component that will be associated with it. In this case the editor is an a `javax.swing.JComponent` object.

```
....
public Component getComponent() {
  return this;
}
```

The `isModified()` method checks to see if the editor has changed the data, and `getCellEditorValue()` which returns the new Polygon created.

```
....
public boolean isModified() {
  return new_poly != null;
}

public Object getCellEditorValue() {
  return new_poly;
}
```

The `JCCellEditor` interface defines the `stopCellEditing()` method, which stops and commits the editing operation. In the case of this example, there isn't any validation taking place, so the `stopCellEditing()` method will be unconditionally obeyed. The

`TriangleCellEditor` also defines a `cancelCellEditing()` method, which resets the new Polygon.

```
....
public boolean stopCellEditing() {
  return true;
}

public void cancelCellEditing() {
  new_poly = null;
  return;
}
```

The editor contains a local method for retrieving a non-null polygon for drawing:

```
....
private Polygon getDrawPoly() {
  if (new_poly == null)
    return orig_poly;
  return new_poly;
}
```

The editor also has to determine the minimum size for the cell.

```
....
public Dimension minimumSize() {
  Rectangle r = getDrawPoly().getBoundingBox();
  return new Dimension(r.width+r.x,r.height+r.y);
}
```

Finally, the editor needs to know how to paint the current polygon into the cell:

```
    ....
public void paintComponent(Graphics gc) {
    // No L&F, so paint your own background.
    if (isOpaque()) {
        if (!gc.getColor().equals(getBackground())) {
            gc.setColor(getBackground());
        }
        Rectangle r = getBounds();
        gc.fillRect(0, 0, r.width, r.height);
    }

    int x, y;

    Polygon local_poly = getDrawPoly();
    gc.setColor(cellInfo.getForeground());
    gc.translate(margin.left, margin.top);
    gc.fillPolygon(local_poly);

    for(int i = 0; i < local_poly.npoints; i++) {
        x = local_poly.xpoints[i];
        y = local_poly.ypoints[i];
        gc.drawOval(x-2,y-2,4,4);
    }

    gc.translate(-margin.left, -margin.top);
}
```

Much of the rest of the editor handles mouse events to drag the triangle points, or to move the whole triangle inside the cell. See the example file for this code.

Finally, the editor contains event listener methods that add and remove listeners from the listener list. These listeners are notified when the editor starts, stops, or cancels an edit.

```
    JCCellEditorSupport support = new JCCellEditorSupport();
    ....
    public void addCellEditorListener(CellEditorListener l) {
      support.addCellEditorListener(l);
    }

    public void removeCellEditorListener(CellEditorListener l) {
      support.removeCellEditorListener(l);
    }
```

Note that an instance of `com.klg.jclass.cell.JCCellEditorSupport` is used to manage the listener list. The `JCCellEditorSupport` class is a useful convenience class for editors that want to send events to JClass HiGrid programs.

The `TriangleCellEditor` is an example of a fairly complex implementation of the `JCCellEditor` interface. It contains all of the core methods of the interface, and extends the capabilities for an interesting type of data. You can use this example to help you to write your own `JCCellEditor` classes that handle any type of data you care to display and edit.

Handling Editor Events

The com.klg.jclass.cell package contains several event and listener classes that enable cell editors and their containers to inform each other of changes to the cell contents, and allow you to control validation of the cell's edited contents.

The simplest way to handle JCCellEditor events is to use the JCCellEditorSupport convenience class. JCCellEditorSupport makes it easy for cell editors to implement standard editor event handling by registering event listeners and providing easy methods for sending events.

JCCellEditorSupport methods include:

Method	Description
addCellEditorListener()	Adds a new JCCellEditorListener to the listener list.
removeCellEditorListener()	Removes a JCCellEditorListener from the list.
fireStopEditing()	Sends an editingStopped event to all listeners.
fireCancelEditing()	Sends an editingCanceled event to all listeners.

For example, consider the TriangleCellEditor. The changes made are not actually sent to the data source until the user clicks on another cell. It is more useful to have the editor send an editingStopped event when the mouse button is released:

```
public void mouseReleased(MouseEvent e) {
  support.fireStopEditing(new JCCellEditorEvent(e));
}
```

For more complete control, however, you will have to use the other event handling classes provided in the com.klg.jclass.cell package:

Method	Description
JCCellEditorEvent	Sent when the JCCellEditor finishes an operation. The JCCellEditorEvent contains the event that originated the operation in the editor.
JCCellEditorListener	The container registers a JCCellEditorListener to let the JCCellEditor inform it when editing has stopped or been canceled.
JCCellEditorEventSource	This class defines the add and remove methods for an object that posts JCCellEditorEvents.

Editor Key Control

Sometimes, you may want your cell editor to be able to accept keystrokes that have already been reserved for a specific purpose in the container (a **Tab** key in HiGrid, for

example). To do this, you need to use the `JCKeyModifier` class to reserve a key/modifier combination:

```
JCKeyModifier(int key, int modifier, boolean canInitializeEdit);
```

Using this class, you can reserve a key for a particular modifier or for all modifiers. To reserve **Ctrl-Tab** and **Shift-Tab** you would specify two `JCKeyModifier` objects with standard `KeyEvent` modifiers; for example, `KeyEvent.ALT_MASK`.

If you want to reserve all **Tab** keys for the editor, you can use either of the following:

- `new JCKeyModifier(KeyEvent.VK_TAB, KeyModifier.ALL);`
- `new JCKeyModifier(KeyEvent.VK_TAB);`

Note that the container can still choose to ignore reserved keys.

4.5 The JCCellInfo Interface

You can see that `JCComponentCellRenderer`, `JCLightCellRenderer` and `JCCellEditor` use the `JCCellInfo` interface to get information about the cell. The `JCCellInfo` interface provides information about how the container wants to show the cell. The renderer and editor determine whether or not to honor the container's request.

The `JCCellInfo` interface gives the renderer and editor access to cell formatting information from the cells of the grid, including:

- foreground color
- background color
- selected foreground color
- selected background color
- font
- font metrics
- horizontal and vertical alignment

This information is fairly generic. The `com.klg.jclass.higrid` package also contains an object called `CellFormat`, which extends `CellStyle` and implements `JCCellInfo` to include more detailed information from the grid.

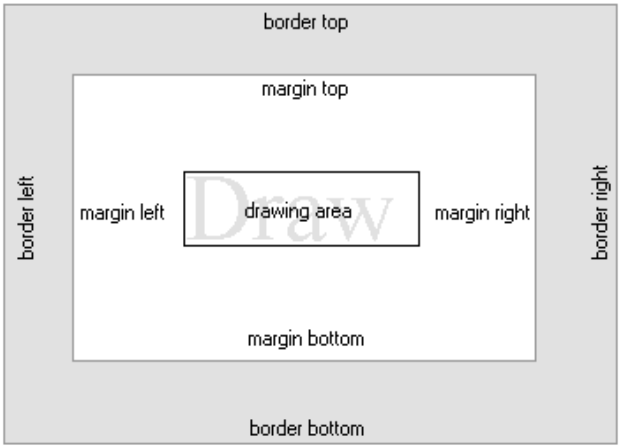


Figure 42 The relationship of border sides, margins, and drawing area provided by JCCellInfo.

The following code comprises the `com.klg.jclass.cell.JCCellInfo` interface:

```
import java.awt.Color;
import java.awt.Rectangle;
import java.awt.Font;
import java.awt.Insets;
import javax.swing.SwingConstants;

public interface JCCellInfo {
public Color getBackground();
public Color getForeground();
public Color getSelectedBackground();
public Color getSelectedForeground();
public Font getFont();
public int getHorizontalAlignment();
public int getVerticalAlignment();
public Insets getMarginInsets();
public Insets getBorderInsets();
public int getBorderStyle();
public Rectangle getDrawingArea();
public boolean isEditable();
public boolean isEnabled();
public boolean getSelectAll();
public int getClipHints();
public Class getDataType();
```

5

JClass DataSource Overview

5.1 Introduction

JClass HiGrid is a visual component that includes a mechanism for accessing data. The classes and methods that retrieve, organize, and store data items form a separate package called JClass DataSource. You can use it with or without JClass HiGrid to interface both to databases and to unbound data sources. With it, you can connect to any type of data source that has a JDBC driver. Its functionality also includes the ability to connect to databases that has JDBC-ODBC driver support, and even to array data produced by another application. The application may be retrieving information from any source, or producing the data itself. You can structure your design to provide top-level information and as many sub-levels as you deem necessary. You can provide your own visual component, or you can use JClass HiGrid as an easy and functional way of providing end-users with a tool that they can use to display, navigate through, and modify retrieved data. Because you have structured the data hierarchically, end-users are able to expand or collapse their view of the sub-levels. You can use JClass DataSource to maintain multiple views of the data. For instance, you might provide your users with a HiGrid to make it easy for them to scroll through many records quickly and at the same time provide them with a form containing data bound components that replicate the fields in the active row of the grid. You control whether edits can be made both in the form and in the grid.

JClass DataSource provides data binding capabilities for JClass Chart, JClass Field, and JClass LiveTable, as well as for JClass HiGrid itself, thereby multiplying your options for an elegantly designed form.

5.2 The Two Ways of Managing Data Binding in JClass DataSource

The core of JClass DataSource is its ability to manage hierarchical data through its data model. The data binding mechanism is built on top of the data model. It contains convenience classes that can be used to do single-level binding of objects, such as a text field to a particular column in a database table. This organization makes it easy for you to bind display components built with JClass Chart, JClass LiveTable, and JClass Field, or other similar components, to a particular database field without having to take account of JClass DataSource's mechanism for handling hierarchical data structures.

This simplified approach to data binding begins with the `ReadOnlyBindingModel` interface. It provides a single-level, two-dimensional view of a data set. It groups all non-update methods and handles read-only events. This interface exists only to provide a logical separation between read-only and non-read-only methods and event handling. It is extended to an interface named `BindingModel`, which extends `ReadOnlyBindingModel` and provides update methods. Operations can be performed on the row currently in focus (for example, by using `getCurrentRowStatus()`), or by specifying a row index (for example, `getRowStatus(rowIndex)`).

Abstract class `ReadOnlyBinding` extends `BindingModel` and provides a base for concrete subclasses. Public class `Binding` extends `ReadOnlyBinding` and provides update methods. Operations can be performed on the row currently in focus. Public class `JDBCBinding` is used to bind to JDBC databases.

Thus, programmers who need to bind a non-grid component to a database need to understand `Binding` and its related classes and interfaces. They need not delve into the intricacies of the `DataModel` and `MetaDataModel` interfaces.

For comparison, there are two ways of accomplishing data binding:

- Using the Data Model:

```
DataModel dm = new TreeData();
MetaDataModel mdm = newMetaDataModel(dm, "select * from orders", c)
table.setDataBinding(dm, mdm, column);
```

- Using Binding:

```
DataModel dm = new TreeData();
MetaDataModel mdm = newMetaDataModel(dm, "select * from orders", c)
table.setDataBinding(mdm.getBinding);
```

5.3 Using JClass DataSource with Visual Components

You can use JClass DataSource with other JClass products and with IDEs that supply data bound visual components. Naturally, the recommended GUI is JClass HiGrid, a versatile and customizable grid built specifically to work side by side with JClass DataSource. You can use JClass LiveTable to bind different tables to a hierarchically-structured data source that you have designed and then built using this product. Or you can connect the data bound components of JClass Field and have a form that displays database records

wherein the end user may make edits. Because JClass Field validates its input based on your specifications, your application is even more functional without you doing all the programming that implementing validation makes necessary. You can use JClass Chart to present values extracted from a database in a visually appealing way, again with customizable features so your application has your own personal flavor.

If you want to use an IDE's visual component, you can still simplify the job of connecting to the database and organizing its tables to meet your application's individual needs.

5.4 JClass DataSource and the JClass Data Bound Components

JClass DataSource is designed to be used for general-purpose data binding needs. In an IDE, all that is required to supply your form with data bound components is to place a `JCData` or `JCTreeData` and use their customizers to configure their properties, which include connecting to a database and, in the case of `JCTreeData`, defining the master-detail relationships between parent and dependent data tables.

JClass components are data-aware. You use their customizers to register with the data source defined with the aid of `JCData` or `JCTreeData`. Custom property editors turn this operation into a sequence of choices – no writing of code is required.

5.4.1 Define the Structure for the Data

For this introduction to JClass DataSource, we'll start with the case where all the needed information is stored in a single database. After a connection to the database is established, the next thing to do is to specify the "root" table. We are assuming that a number of sub-tables are also going to be defined. These sub-tables may, in turn, have sub-tables. Thus, the data is being modeled as a tree structure, and the highest level table is the root of this tree.

This description of the data is called meta data. The `MetaData` class, based on a `MetaDataModel`, is used to capture this hierarchical design. This `MetaData` class connects to a data source through the JDBC or an IDE-specific data-binding mechanism. There is an instance of `MetaData` for each level in the tree, and each instance of `MetaData` has a particular query associated with it. `MetaData` will execute that query and cache the results. When used in the context of JClass HiGrid, multiple result sets will be cached. These result sets will be based on the same query but with different parameters. When used in JClass HiGrid, this object will be a node in the meta tree describing the relationships between SQL queries.

The root table is treated specially. It is distinguished from dependent tables by having its own constructor. The root table retrieves its data when it is instantiated. Dependent tables can wait until they are accessed before they make calls to the database.

5.4.2 JClass DataSource's Organization

You define the abstract relationship between data tables as a tree. This is the meta data structure, and after it has been designed, you query the database to fill data tables with result sets. The abstract model defines the structure and the specific data items are retrieved using a dynamic bookmark mechanism that is imposed on the result set data tables. At the base level of the class hierarchy, the `MetaData` class describes a node in the `MetaTree` structure and the `DataTable` class holds the actual data for that node. There are different implementations of `MetaData` for differing data access technologies, therefore there will be a different `MetaData` defined for the JDBC API and for various IDE-specific data binding solutions. Similarly, there will be different `DataTable` classes depending on the basic data access methodology.

`MetaData` and `DataTable` are concrete subclasses of the abstract classes `BaseMetaData` and `BaseDataTable`. The latter is an abstract implementation of the methods and properties common to various implementations of the `DataTable` model. This class must be extended to concretely implement those methods that it does not, which are all of the methods in the data table abstraction layer. Both these classes are derived from `TreeNode`, which contains a collection of methods for managing tree-structured objects.

The `MetaDataModel` interface defines the methods that `BaseMetaData` and its derived classes must implement. This is the interface for the objects that hold the meta data for `DataTable`s. There is one `MetaDataModel` for the root data table, and there can be zero, one, or more `DataTable` objects associated with one meta data object for all subsequent nodes in the meta data model. Thus it is more efficient to store this meta data only once. In terms of *JClass HiGrid*, meta data objects are the nodes of the meta tree. The meta tree, through instances of the `MetaData` classes, describes the hierarchical relations between the meta data nodes. `DataTableModel` is the interface for data storage for the *JClass HiGrid* data model. It requests data from instances of this table and manipulates that data through this interface. That is, rows can be added, deleted, or updated through this `DataTable`. To allow sorting of rows and columns, all operations access cell data using unique identifiers for rows and columns.

The `DataModel` interface is the data interface for the JClass DataSource. An implementation of this interface will be passed to JClass HiGrid. All data for the data source is maintained and manipulated in this data model through its sub-interfaces. This data model requires the implementation of two tree models, one for describing the relationships of the hierarchical data (`MetaDataModel`) and one for the actual data (`DataTable`). `TreeData` is an implementation of `DataModel` for trees and listener functions. Important methods are `requeryAll`, `updateAll`, `add/removeDataModelListener`, and `enableDataModelEvents`. This last method is useful when you are making many changes to the data without having listeners repaint after each individual change. This is a different procedure than using `DataModelEvent BEGIN_EVENTS` and `END_EVENTS`, where events are still sent but the listener receiving `BEGIN_EVENTS` knows it may choose to disregard the events until it receives `END_EVENTS`.

The DataModel has one "global" cursor. Commit policies rely on the position of this cursor. This cursor, which is closely related to the bookmark structure, can point anywhere in the data that has been retrieved by JClass DataSource and placed in its data tables. It is found using getCurrentGlobalBookmark. Additionally, each DataTableModel has its own "current bookmark" or cursor. This cursor is retrieved using getCurrentBookmark. If another table is referenced, likely via the getTable method, another completely independent row cursor can be found, again using getCurrentBookmark, that can be used to pore over the table using methods such as first, last, next, previous, beforeFirst, and afterLast.

5.5 The Data Model's Highlights

The Data Model performs these major functions:

- Connects to a database.
- Defines the structure for the data that is to be retrieved and displayed.
- Specifies the tables and fields to be accessed at each level.
- Sets the commit policy to be used when updating the database.
- Stores result sets from queries.
- Informs the database about pending deletes, updates, and insertions.
- Instructs the database to commit changes at the correct time.

5.5.1 Making a Database Connection with the Help of the JDBC-ODBC Bridge

The Data Bean and The Tree Data Bean components use a JDBC, Java's specification for using data sources, although other data sources, such as ODBC, can be used with the help of a JDBC-ODBC bridge. Both Beans may have multiple connections, and these may be via different database drivers.

If your development system is running on a Windows NT (Windows 95/98/2000) platform, install the needed database drivers.

1. On Windows NT/95/98: choose **Start** > **Settings** > **Control Panel** > **ODBC** (32-bit ODBC for Windows 95/98)to launch the ODBC data source administrator. On Windows 2000: choose **Start** > **Settings** > **Control Panel** > **Administrative Tools** > **Data Sources** (**ODBC**) to launch the ODBC data source administrator.
2. Click on the **User DSN** tab and observe the *User Data Sources* list.
3. If the data source you need is not already there, click on the **Add** button.
4. Select the driver for your data source from the list in the *Create New Data Source* window.
5. Use the **Configure** button to supply extra information specific to the database engine you will be using.

Other environments define different methods for making the low-level connection to a database. Consult the system documentation for your environment for its recommended connection method.

5.6 The Meta Data Model

Consider a master-detail design such as that shown in Figure 43. You can create a class that captures this model programmatically, or you can describe it using the `JCTreeData`'s customizer in an IDE.

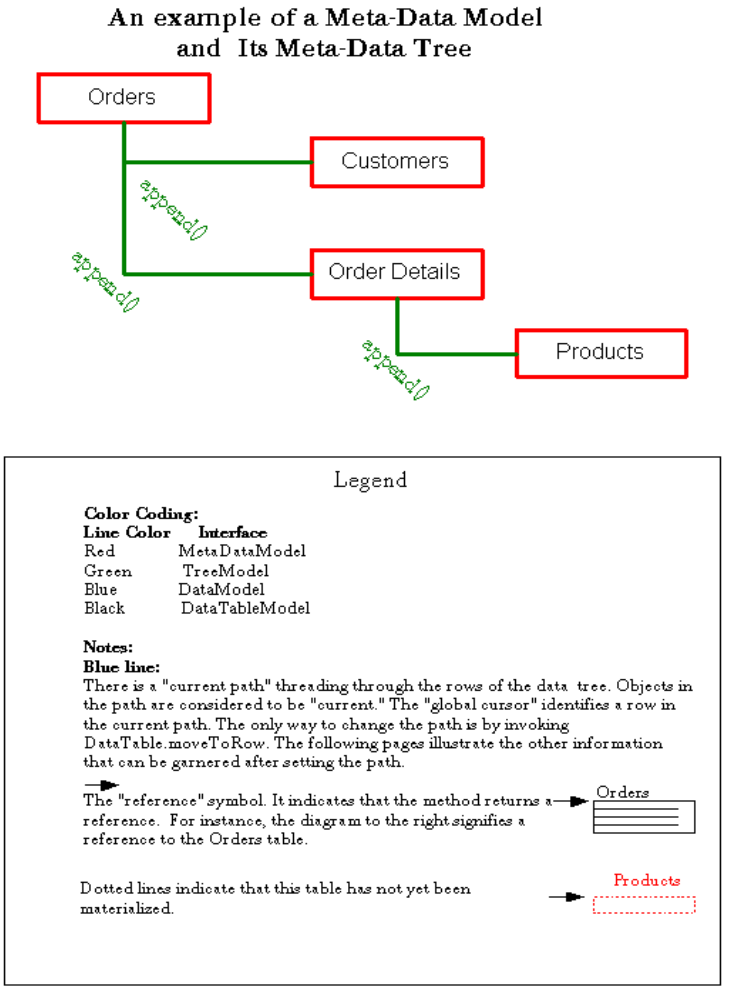


Figure 43 A meta data tree containing meta data objects at its nodes.

Both the HTML and PDF on-line versions of this manual are color-coded to distinguish which objects implement the interfaces mentioned in the Legend. The structure of the meta data tree (green) can be defined after first creating the meta data objects:

```
//"this" is a class extended from TreeData
// Set up the root level: Orders
       BaseMetaData orders         = new BaseMetaData(this);
   // The rest of the meta data is defined the same way

   BaseMetaData customers      = new BaseMetaData(this);
   BaseMetaData orderDetails   = new BaseMetaData(this);
   BaseMetaData products       = new BaseMetaData(this);
```

The hierarchical relationships among these meta data objects are defined using the `append` method:

```
        // now add the meta data objects to the tree to
        // provide the hierarchy. Orders is the root. OrderDetails
        // and Customers are siblings at the next level
        // and Products is a child of OrderDetails.
        getMetaDataTree().setRoot((TreeNode) orders);
        orders.append(Customers);
        orders.append(OrderDetails);
        //Since Products depends on OrderDetails:
        orderDetails.append(products)
```

To sum up, the append method places the meta data objects in their proper positions in the meta data tree. The same thing can be accomplished without coding if you use the `JCTreeData` customizer in an IDE.

5.6.1 Keeping Track of Rows

Now that the meta data has been defined the model can be given over to a grid such as JClass HiGrid, which will manage the display. Behind the scenes, the JClass DataSource has retrieved and cached a number of rows of each table. This number may be zero for any sub-table that has not yet been opened, but all the rows of the root table that match the query are cached. JClass DataSource needs to keep track of these rows, and to accomplish this in an efficient manner more than one strategy is employed. The most important parameter that labels a row is called the *bookmark*. This `long` integer is guaranteed to uniquely label a row at any given time, but it is not guaranteed to be invariant. A row's bookmark most probably will change over time as a result of insert and delete operations on other rows. Other operations may cause a reassignment of bookmarks to existing rows. Thus, if you store a row's bookmark, you must ensure that you do not perform any of the operations that may change the bookmark in the interim before you use it again and expect that it refers to the same row. In fact, after bookmarks have been reassigned, an old bookmark may not refer to any row.

While bookmarks are sufficient to label a row, efficient operation requires other ways of labeling them. A quantity called the *row index* is used to label each row of a given table sequentially, starting at zero. Obviously, these numbers are not unique as soon as there is more than one table in your application, but they do serve to help you to easily loop through the rows of a given table.

A *global cursor* keeps track of the cell containing the editor. This cell is selected and has focus. There is at all times one and only one active editor. Because data bound components can be attached to any meta data level, a mechanism is required to allow that component to decide what data it should display depending on where the global cursor is positioned. A construct called the *current path* assists in this regard. As you follow this discussion, please refer to Figure 44.

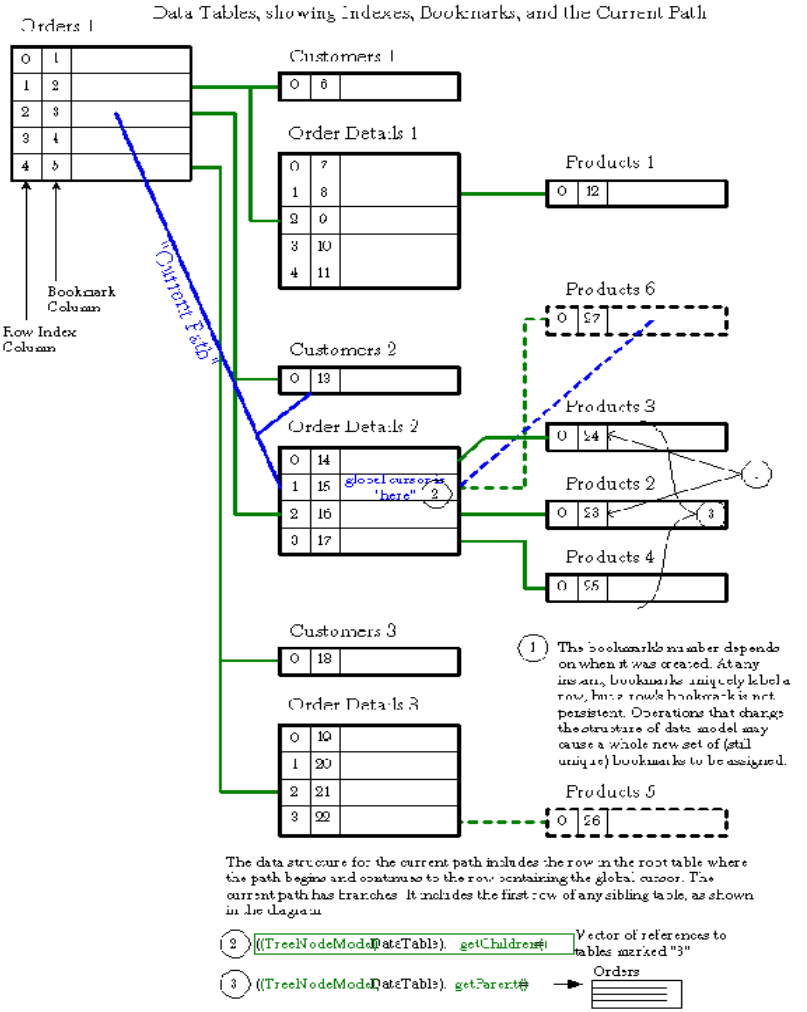


Figure 44 How rows are indexed.

Our example has the global cursor positioned on the row whose bookmark is 15 in the table we have called Order Details 2. This name serves to identify a table in the diagram but it is not a name that would appear anywhere in the Java code. It indicates that it was the second Order Details table created, perhaps as a result of a user clicking on row 3 of the root table, Orders, in a grid. Assume further that you have bound a text field component to one of the columns in the Products meta data. What information, if any, should the field display? In this case the choice is rather obvious: the text field should display the information contained in the chosen column of Products 6 rather than leaving the field blank simply because the user has not yet opened this level using the grid. In a less obvious case, what should be done if the Products 6 table contains more than one row? In this case, the current path points to the first row of the table, and to the first row of all dependent tables if they exist. Because of the possibility of using data bound components, a current path must include branches such as the connection to Customers 2. Again, because an application could have added a data bound component to the Customers level, JClass DataSource must be able to tell which particular piece of information to use when the field itself is not selected. Again in our example, there is only one customer per order so the choice of which row to use does not arise. In general, when there are a number of possibilities, the one with row index 0 is chosen.

What happens when an application containing a data bound component at the Products level starts? From the point of view of the component, and taking a column in table Products as our example, the sequence is as follows. The component requests data. Products has no data in it as yet, so it asks Order Details to supply a reference. Since Order Details has no data, it asks Orders for a reference. Orders responds with its current row, which by default is the row whose index is 0. Order Details can now populate itself, and passes its default row index, again 0, for the table corresponding to row 0 of Orders, to Products. Products populates the referenced data table and the component receives its data. In this way a forward referencing policy is established and components always contain values, even at start up.

If the global cursor is somewhere down the hierarchy, back references are easy: just follow the tree back to its root. In the case where there are two tables at a level and the cursor is in one of them, the row whose index is 0 of the other table is deemed to be on the current path, so any component bound to that table would choose its value (or values) from the index 0 row.

The next figure (Figure 45) shows some of the common ways of using bookmarks to navigate around the hierarchy. The color coding in this figure is the same as that in Figure

43. Some of the methods return references to tables, others return bookmarks and row indexes.

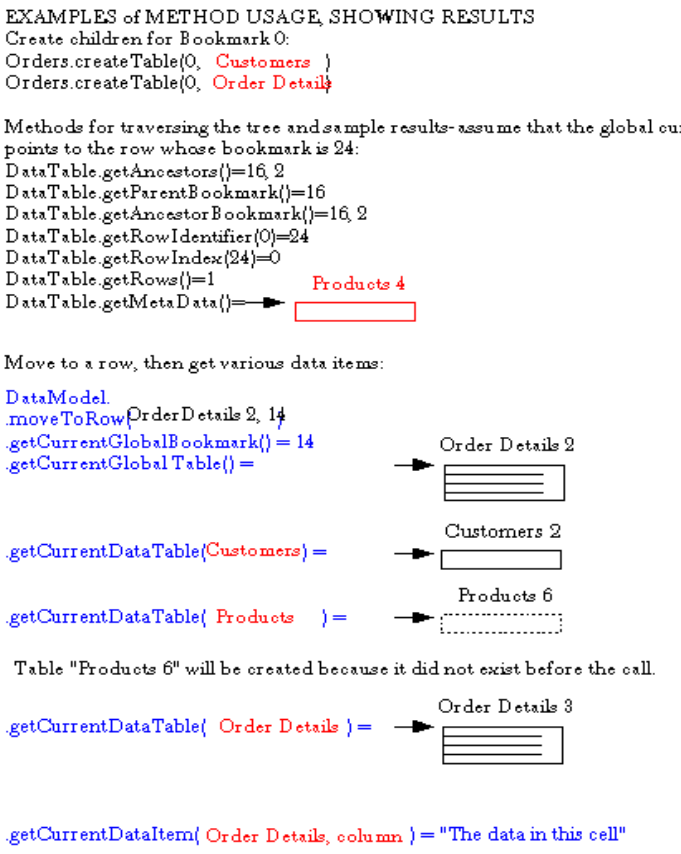


Figure 45 Using bookmarks and row indexes.

If you have noticed that there are some capitalized names in the above examples in places where lower case method names are expected, it is because these capitalized names are used to indicate the class of object that is being talked about, not instances of that class. You must have an instance of the class to produce legal Java code.

The `createTable` method in class `com.klg.jclass.datasource.BaseDataTable` creates and returns the `DataTable`, which corresponds to the specified row in the parent for the indicated child `MetaData` object.

Methods `getAncestors`, `getParentBookmark`, `getAncestorBookmark`, `getRowIdentifier`, `getRowIndex`, `getRows`, and `getMetaData` all return numeric data, except for the last which returns a reference to its own table.

The `getMetaDataTree` method returns a reference to the root of the `MetaData` tree.

5.7 Setting the Data Model

The data model may be set programmatically or through a customizer. Both methods are described here.

Setting Up an Unbound Data Model Programmatically

The `DataModel`, `MetaDataModel`, and `DataTableModel` interfaces define the structure that needs to be established no matter what kind of data source will eventually be used. Base classes `TreeData`, `BaseMetaData`, and `BaseDataTable` are available for use as implementations of these interfaces. The process of creating data tables begins with a `DataModel`, possibly by instantiating or subclassing the `TreeData` class. Normally, the data tables in JClass DataSource are derived from corresponding tables in a database, but that need not be the sole source. They can be created dynamically, as exemplified by the example program called `VectorData`. This programmatic source data is used as an alternate in case there is a problem in connecting to the sample database.

It serves to illustrate the origination of a data source. The `VectorData` class itself extends `TreeData`, so it functions as the data model:

```
public class VectorData extends TreeData
```

Array variables within this class are used to define the columns and their associated data types for the tables that are about to be created. After a data model is available, the next step is to create the meta data objects for the various levels that tables will occupy. The "bare" meta data objects are instantiated through a call to `BaseMetaData`'s constructor, giving the data model as a parameter. An example is the following line of code:

```
BaseMetaData Orders         = new BaseMetaData(this);
```

The meta data objects must be structured by specifying their hierarchy. The example specifies a root table called *Orders* with two children called *OrderDetails* and *Customer.* Capturing this hierarchy reduces to adding the meta data objects to a tree. Since the `getMetaDataTree` method in `TreeData` is an implementation of the one named in `DataModel`, and returns a `TreeModel`, it can be used to set the Orders table as the root of the tree:

```
getMetaDataTree().setRoot((TreeNode) Orders);
```

The children are placed by appending them to the root:

```
Orders.append(OrderDetails);
Orders.append(Customers);
```

The tables' relationships to one another have been set, but the tables themselves have no definition as yet, let alone any content. Since column names and data types are available in arrays called `orders_columns[][]` and `details_columns[][]`, they are used to set up the columnnar structure of the tables as follows:

```
// set up columns for the Orders table
for (int i = 0; i < orders_columns.length; i++) {
BaseColumn column = new BaseColumn();
column.setColumnName(orders_columns[i][0]);
column.setMetaColumnTypeFromSqlType(getType(orders_columns[i][1]));
column.setColumnType(getType(orders_columns[i][1]));
Orders.addColumn(column);
```

A similar block of code sets up the columnar structure of the *OrderDetails* table. Note that columns can be derived from `BaseColumn`, which is an implementation of the `ColumnModel` interface.

At this point the actual data table can be created using the constructor for a `BaseDataTable` and passing a `MetaDataModel` as a parameter. Since `VectorDataTable` is subclassed from `BaseDataTable`, and `Orders` is a `BaseMetaData` object and therefore implements the `MetaDataModel` interface, the following code creates the root level of the data tree:

```
VectorDataTable root = new VectorDataTable(Orders);
getDataTableTree().setRoot((TreeNodeModel) root);
```

The values in the cells of a row are computed. The example merely fills them with random data by declaring an array called `row` and generating data for each cell, that is, for each element of the array. `BaseDataTable` has a method called `addInternalRow(Object row)` that does the job:

```
root.addInternalRow(row);
```

The two sub-tables are instantiated by a call to the other form of `BaseDataTable`'s constructor.

```
public VectorDataTable(MetaDataModel metaData, long parentRow) {
   super(metaData, parentRow);
}
```

In the example program, the tables are instantiated within a custom version of `createTable`. This method is part of the `DataTableModel` interface and is defined in `BaseDataTable`. It is overridden in the example's `VectorDataTable` class so that it can populate tables from array data generated within the program rather than by querying a database. To see how an unbound data table can be generated, check `example.datasource.vector.VectorDataTable.java` in the *examples* directory.

5.7.1 Query Basics

JClass DataSource has not been designed to create databases or their tables (for instance, by adding new columns to the database itself), although, technically, it is possible to do so. It is assumed that you have an existing database and you want to provide a

hierarchical graphical interface for its tables and fields, perhaps adding summary columns of your own design. The contents of a database are examined and modified through the use of SQL's Data Manipulation Language (DML), whose basic statements are SELECT, INSERT, UPDATE, and DELETE. JClass DataSource parses an SQL statement into its clauses but it does not attempt to validate the clause itself. Instead, it passes the clauses on to the underlying database which will determine whether it can process the statement or not.

For instance, in the query:

```
String query = " select *";
query += " from sales_order a. fin_code b";
query += "where a.fin_code_id = b.code";
query += "order by a.id asc";
```

the `where` and `order by` clauses will be passed on to the database without any check on their contents.

You can use *Prepared Statements*. The interfaces `java.sql.PreparedStatement` and `java.sql.Connection` are used for this purpose. Consult the `java.sql` API for further information. You can use the question mark parameter (?) as a placeholder for joins. The question mark is a placeholder for the field that will be supplied when the statements are executed. An example of the use of the question mark placeholder is as follows:

```
order_detail.setStatement("select * from sales_order_items where id = ?");
```

Here, a matching `id` field in the parent table is used in the comparison.

5.7.2 Specify the Tables and Fields to be Accessed at Each Level

If you are using the JDBC but not using an IDE, you must create instances of the `MetaData` class for each level programmatically, specifying both the table and the SQL query to be used. One form of the constructor is required to instantiate the root table. The database query is passed as one of its arguments. All other levels are instantiated using a form that names the instance of the `HiGrid` (or other GUI) being used, the table name, and the database connection object. The query String is passed separately, using a method called `setStatement`.

Other parameters can be set, such as descriptive statements for the table captions, header and footers, columns containing aggregate data, and so on.

5.7.3 Set the Commit Policy to be Used when Updating the Database

You have control over when changes should be committed: you can choose a commit policy that ranges from allowing the end-user to decide when to commit the changes, waiting until the selected row changes (waiting until the selected group changes), or giving your application control over when to commit the changes.

5.7.4 Store Result Sets from Queries

Database queries may result in a varying amount of data; anything from the entire table on which the query was based to a null result in the case where the database returned nothing at all that matched the query. If these results are to be displayed in a grid, the result set must be stored. The result sets for each query that you define at each level are stored separately.

5.7.5 The Data Bean

Use JClass DataSource's `JCData` when you want to present a single level of data to the end-user. In effect, this JavaBean functions as a table whose rows are retrieved from the chosen database and whose columns are the fields that you select from the table (or tables, if two or more are joined).

What follows is an example of using the Data Bean in the BeanBox. We'll show all the steps in setting up the database access, and all the way through to connecting to a JClass HiGrid to display the query's result set.

1. Once you have installed your JAR file in its proper location, which in the case of the *BeanBox* is your *bdk/jars* directory, you should see the JavaBeans called `JCData`, `JCTreeData`, and the Swing-based data bound components `DSdbJComboBox`, and so on, as well as `DSdbJNavigator`, JClass DataSource's data navigator.

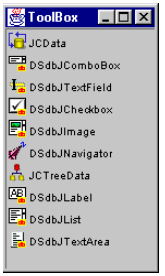


Figure 46 The Bean Development Kit's ToolBox, containing HiGrid's Beans.

2. Click on **JCData** and move the mouse pointer to the BeanBox, where it turns into a crosshair. Click once more and the outline of the data JavaBean appears. The data JavaBean has a property sheet like the one shown next.

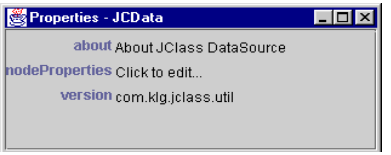


Figure 47 JClass HiGrid's JCData's property sheet.

3. Click on the area to the right of the label *nodeProperties* to access its main custom editor. A modal dialog appears, reminding you about ensuring that the serialization file which is about to be created is on your CLASSPATH.

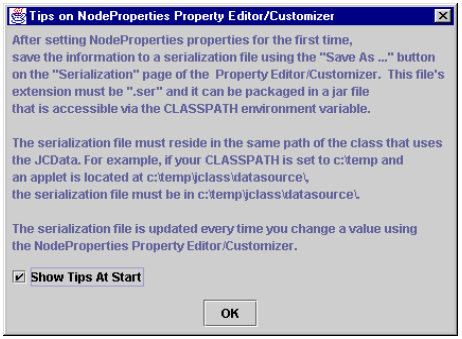


Figure 48 A reminder about creating a serialization file.

4. On the *Node Properties Editor*, click **Open** if you have previously-saved serialization file that you want to use; otherwise, click **Save As**. Type a filename in file dialog, or accept the default name and click **Save**.
5. Click the NodePropertiesEditor's **Data Model** tab.
6. There are two nested tabbed dialog panels. The **JDBC** tab is selected, causing the **Connection** tab panel to be visible. The reason for this choice is that it is assumed that the first thing you want to do is to specify the database connection. The other tabs are **Data Source Type**, **Data Access**, **Virtual Columns** and **IDE**. They will be discussed shortly.

   There are text fields for the server name, host or IP address, TCP/IP port, and Driver, along with a group of fields that may be required to log on to a database that

requires a user name and a password. You fill in as many of these as are required for your particular situation.

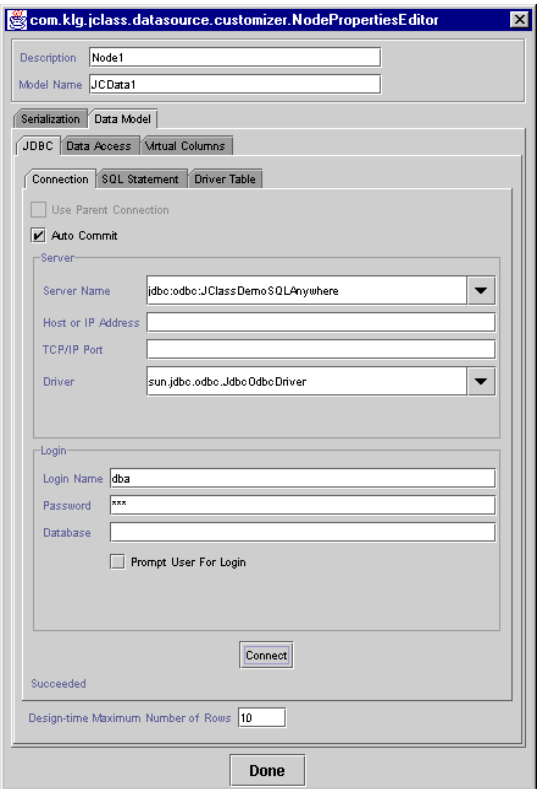


Figure 49 Fill in these fields to connect to your chosen database.

For reference, the other tabs permit you to specify the driver table and the type of data access that will be allowed.

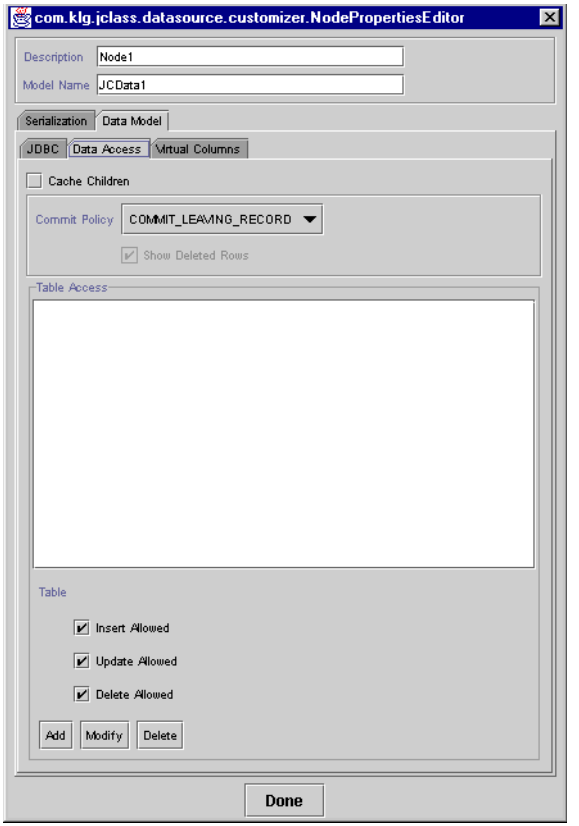


Figure 50 The Data Access tab, where you can set a commit policy and an edit policy.

7. The **Data Access** tab allows you to set a commit policy and an edit policy. Three checkboxes control editing permissions: **Insert Allowed**, **Update Allowed**, and **Delete Allowed**. You can choose one of three commit policies: `COMMIT_LEAVING_RECORD`, `COMMIT_LEAVING_ANCESTOR`, or `COMMIT_MANUALLY`.
8. Return to the **JDBC** tab and click on the **SQL** Statement tab. This exposes the tab containing two scroll panes, shown in Figure 51. The top scroll pane is for placing the smaller scroll panes that represent the tables from your database. The first step in

choosing a table is to right-click on the top scroll pane, or click on the button labeled **Add Table**. Click **Add** in the popup menu that appears.

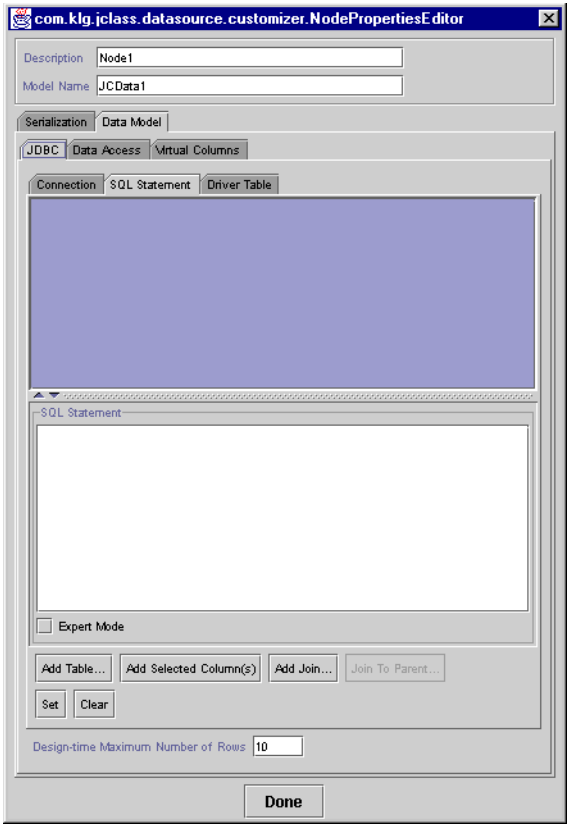


Figure 51 The NodeProperties Editor's SQL Statement tab.

The database is accessed and a list of its tables is retrieved.

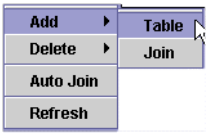


Figure 52 The popup menu for adding tables.

9. A new dialog will appear, allowing you to select a table from the list of retrieved database tables. Choose a table and click **Add**.

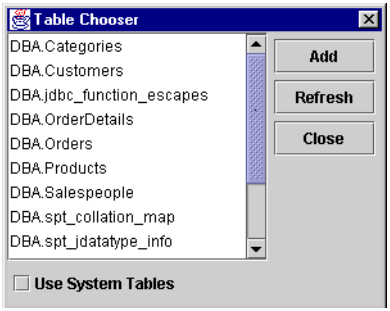


Figure 53 The Table Chooser dialog.

You can choose more than one table if you wish. The result will be a grouping of the two tables, but as yet no columns or joins have been specified. For this operation to be meaningful, it is likely that you will have to first choose the tables whose data you will be accessing, then specify the names of the common fields in each table.

Each data table scroll pane has a label that identifies it. The scroll area contains the list of fields for that table. You build the query by selecting a field in this list and choosing **Add Selected Columns**. This action causes the field name to be added to the select statement in the *SQL Statement* scroll pane.

The query in the *SQL Statement* pane contains an editable text frame. You can refine the query by adding any clause that the database language supports.

A sample, containing two tables, is shown in Figure 54.

10. **Click the Set button to store the query**. If you omit this step and close the SQL Statement tab, all the settings you made will be cleared.

Important: Realize that the *SQL Statement* panel has helped you build a query simply by making the appropriate choices in the customizer, although you can type query

statements into the text area of the SQL statement panel. Your IDE takes it from there and builds the necessary code.

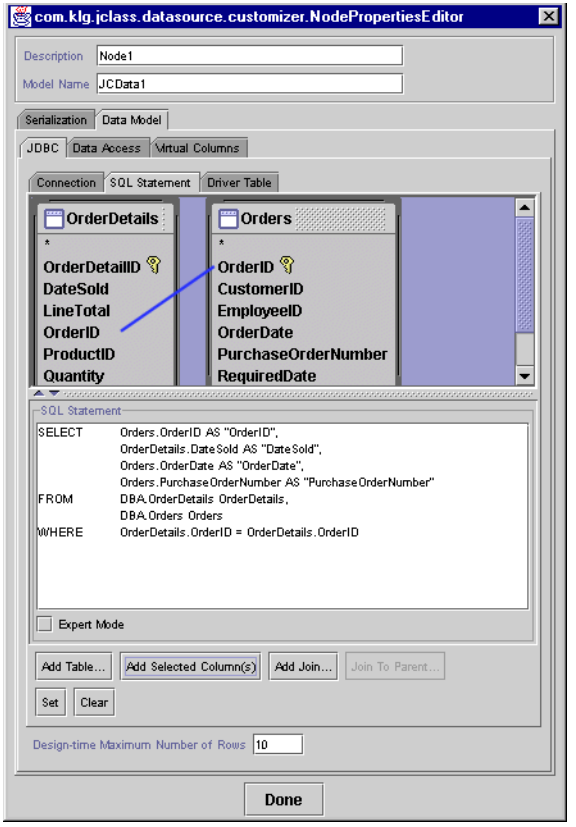


Figure 54 The SQL Statement panel.

The setup of the *JCData* is complete. What remains is to connect this JavaBean to a visual component so that the result set can be displayed. We'll carry on with this example by actually displaying the result of our query.

11. Select *JCHiGrid* in the *ToolBox* and place an instance on the *BeanBox*.
12. Resize it so that it is big enough to hold most of the cells in five or six rows.
13. In the *BeanBox*, highlight the *JCData* and choose **Edit > Events > dataModel > dataModelCreated**. A line in the form of a rubber band extends from the JCData component to the tip of the mouse pointer.

14. Move the tip of the mouse pointer anywhere along the edge of the `HiGridBean` component, click and again choose **dataModelCreated** from the popup menu that appears.
15. Your grid fills with the retrieved data, as shown in Figure 55.

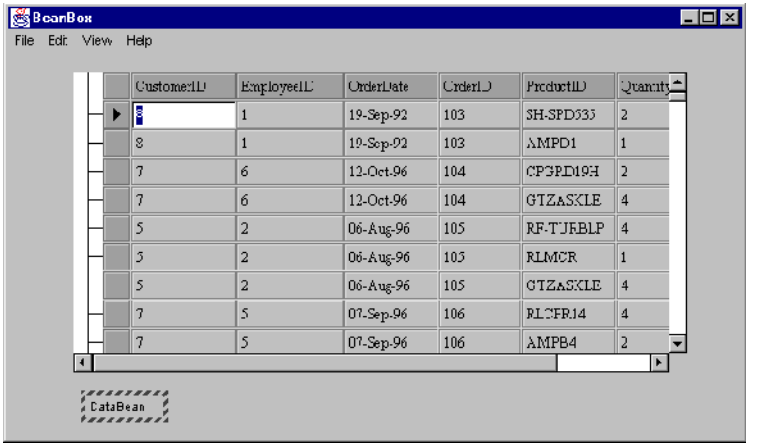


Figure 55 A database table as it appears in the BeanBox.

5.8 JClass DataSource's Main Classes and Interfaces

A `TreeModel` interface defines the methods that implement a generic interface for a tree hierarchy. The tree interface is used for organizing the meta data and the actual data for the JClass DataSource.

The `DataModel` is the data interface for the JClass DataSource. An implementation of this interface will be passed to the data source. All data for the DataSource is maintained and manipulated in this data model through its sub-interfaces. This data model requires the implementation of two tree models: one for describing the relationships of the hierarchical data (`MetaDataModel`) and one for the actual data (`DataTableModel`).

The `TreeNodeModel` is the interface for nodes of the `TreeModel`.

`BaseDataTable` provides a default implementation of `DataTableModel` and `DataTableAbstractionLayer` interfaces. Instances of this class provide basic storage, retrieval, and manipulation operations on data rows. This class can be used without extending it. In this case you must create and populate rows manually. For example,

```
BaseDataTable root = new BaseDataTable(rootMetaData);
data_tree.setRoot((TreeNode) root);
int row1 = root.addRow();
root.updateCell(row1, column1, value1);
root.updateCell(row1, column2, value2); // etc. ...
```

Extensions of this class, for example the JClass DataSource JDBC implementation, automatically create/populate data tables based on data returned from datasource queries. In the case of IDE-specific implementations, they extend this class and override the data storage and retrieval mechanisms. Users wishing to extend this class should look at overriding some or all of the methods defined in the `DataTableAbstractionLayer` interface. These are the methods most likely to need overriding. Each instance of this class has its own cursor which can be navigated independently of the `DataModel`'s global cursor. Only the global cursor (controlled by `DataModel.moveToRow`) causes commits to occur when the current row is changed. The navigation methods here do not cause the global cursor to change, only this table's private cursor.

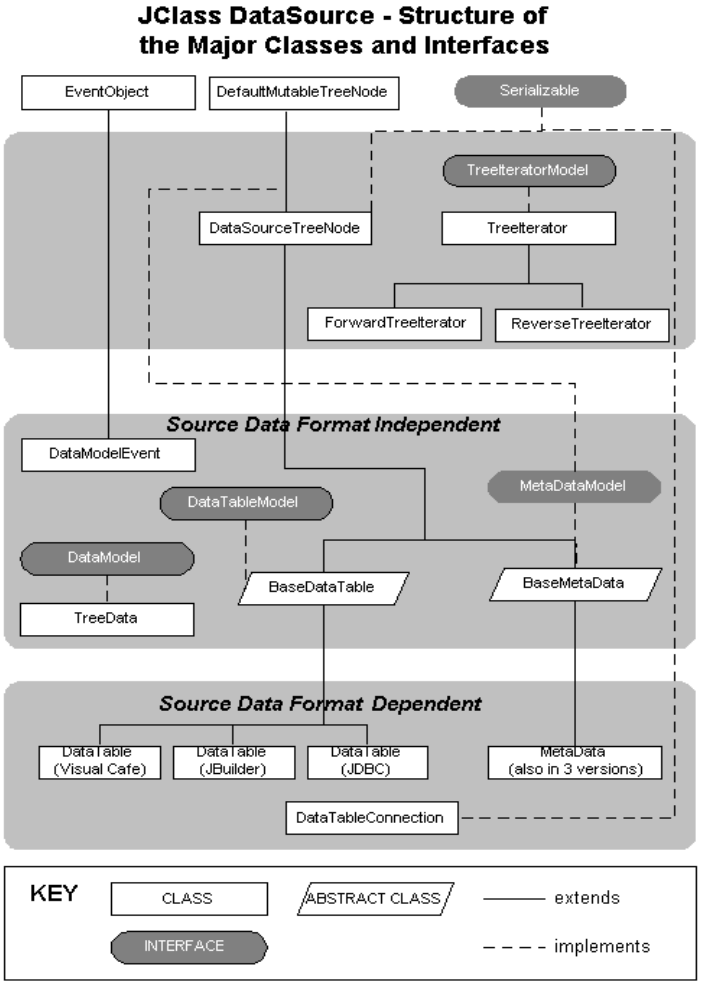


Figure 56 Major classes and interfaces for the Data Model .

Commit Policy

Updating a database is a two step process. First, a cell or group of cells is edited, then the edits are confirmed by committing them to the database.

COMMIT_MANUALLY – Requires a click on the pencil icon (or the X icon), or you can select any of *Update All, Update Current, Update Selected* from the pop-up menu.

COMMIT_LEAVING_RECORD – Commits changes to a row as soon as the cursor moves to a different bookmark.

COMMIT_LEAVING_ANCESTOR – Does not commit until a sub-tree is accessed which has a different parent-level bookmark than the previous one (see `DataTableModel.getMasterRow()`). By convention, setting the root-level `MetaData` object to `COMMIT_LEAVING_ANCESTOR` is equivalent to setting it to `COMMIT_MANUALLY`.

5.9 Examples

Row Nodes

It often helps in simplifying your code if you assign names to rows. Method `setDescription` assigns any name you choose to a row node. This name can then be retrieved with `getDescription`. Since `getDescription` requires an object of type `MetaDataModel`, a possible invocation would be:

```
String x = rowNode.getDataTableModel().getMetaData().getDescription();
```

You can find the row node associated with an event as follows:

```
ValidateEvent event = e.getValidateEvent();
   RowNode rowNode = e.getRowNode();
```

`getRowNode` returns the row node of the row on which the event occurred.

5.9.1 Useful Classes and Methods as Demonstrated by Code Snippets

The following sections demonstrate some common tasks by using code snippets.

For most applications, you will need to perform the following steps:

- Connect to a database
- Set commit policies
- Specify joins on tables using single or multiple keys
- Refresh data structures after the data has been modified (including the insertion of a new row or deletion of a row)
- Inspect bookmarks and column identifiers
- Sort data

- Programmatically move through the retrieved-record data structure and possibly calculate totals or other summary information
- Cancel some or all of a group of pending changes
- Inspect column identifiers
- Recover from operations that attempt to violate database integrity.

General

Connecting to a Database via a JDBC-ODBC Bridge and Setting the Top-level Query

Use the `DataTableConnection` constructor to instantiate a new connection, then use the root form of the `MetaData` constructor to set the top-level query.

```
DataTableConnection c = new DataTableConnection(
"sun.jdbc.odbc.JdbcOdbcDriver",     // driver
"jdbc:odbc:GridDemo",               // url
"Admin",                            // user
"",                                 // password
null);                              // database

String query_string = "SELECT * FROM myTable";
MetaData root_meta_data = MetaData(data_model,c, query_string);
```

Joining Tables

Joining tables involves creating a new node that names its parent using the second form of the `MetaData` constructor, building a query statement, and setting it on the node, then issuing the join command or commands. Here, two joins are indicated.

```
private MetaData createDetailChild(InsertData link, MetaData par,
DataTableConnection c)
    {
        try
        {
            // Link the customer table to the SalesOrder table
            MetaData node = new MetaData(link, par,c);
            node.setStatement("SELECT * FROM OrderDetail WHERE order_id
                = ? AND store_id = ?");
            node.joinOnParentColumn("id","order_id");
            node.joinOnParentColumn("store_id","store_id");
            node.open();
            node.setColumnTableRelations("OrderDetail", new String[]
                {"*"});

            return node;
        }
        catch (DataModelException e)
        {
            ExceptionDump.dump("Creating OrderDetail Child Table", e);
            System.exit(0);
        }
        return null;
    }
```

Refreshing Tables

This example shows how you might construct a method that refreshes a table. The data types of the variables can be inferred from the casts.

```
public void refreshStructure()
{
   this.meta_tree = (TreeModel) this.data_model.getMetaDataTree();
   this.meta_data_model = (MetaDataModel) this.meta_tree.getRoot();
   this.data_tree = (TreeModel) this.data_model.getDataTableTree();
   this.data_table_model = (DataTableModel) this.data_tree.getRoot();
}
```

Setting Permissions

This example shows how to set modification permissions programmatically.

```
public void setPermissions(String table,boolean ia,boolean da,
   boolean ua)
{
   this.meta_data_model.setInsertAllowed(table,ia);
   this.meta_data_model.setDeleteAllowed(table,da);
   this.meta_data_model.setUpdateAllowed(table,ua);
 }
```

5.10 Binding the data to the source via JDBC

In order to bind the data, you must first connect a database using the DataSource customizer. This is described in Making a Connection to a Database, in Chapter 7.

Accomplishing the same thing programmatically involves these steps:

1. Create an instance of a `DataModel` which will be passed to the connection method, so the class in which the connection parameters and the query are formed becomes the first parameter in the call to `MetaData`.
2. Next, form a `DataTableConnection` object, and a query String.
3. Once the connection is made, and the query is passed to the database, use the root constructor `MetaData(DataModel, DataTableConnection, String)`.
4. If sub-tables are required, they are constructed using `MetaData(DataModel, MetaData, DataTableConnection)`. In this form of the constructor, the `MetaData` object refers to the parent table.

Note: The query String must satisfy the database language requirements. Generally speaking, SQL-92 should be used.

5.10.1 Getting Table Names

Some databases have trouble sorting out the proper association when two or more tables are used at the same grid level. In these cases, `ColumnModel` method `getTableName` fails to

return the necessary information about column names. In this case, you must supply the proper join or update statement yourself. A helper method named `setColumnTableRelations` is available.

The `setColumnTableRelations` method explicitly sets the relationships between tables and columns. If introspection fails to determine the association between tables and their columns (when there is more than one table to a level), or you wish to override the associations, say to exclude a column in the update statement, use this method. This method must be called for each table. For example, if `SalesOrder` and `Store` are joined in a one-to-one relationship for a level, these would be the necessary calls. In this example the `MetaData` object is called `Orders`.

```
Orders.setColumnTableRelations("SalesOrder", new String[]
   {"id","store_id","cust_id","ship_via","purchase_order_number",
   "order_date","order_total"});
Orders.setColumnTableRelations("Store", new String[]
   {"store_store_id","address","phone_number","name"});
```

For a single table on a level the call would be,

```
Customer.setColumnTableRelations("Customer", new String[] {"*"});
```

Note: The "*" means all columns are from the Customer table.

5.10.2 Ambiguous Column Names

The JClass data model requires that if a column or field in one table has the same name as that in another table, the two must be capable of being meaningfully joined. That is, the two names must refer to the same logical property of some entity. Since you cannot always control the various field names in database tables, there is an alias mechanism that allows you to rename dissimilar fields that happen to have the same name. Assume that a *SalesOrder* table has an *id* field, and a table named *Store* also has a field called *id*. These keys respectively refer to a sales order reference number and the identification number for a store. If you wish to form a query in which both tables are mentioned, and the *id* field of both is to be selected, you provide an alias for one of the fields in the query statement. Its syntax goes like this:

SELECT SalesOrder.id, ... other SalesOrder field names ... , Store.id AS store_alias_id, ... other Store fields

Now that *Store.id* has an alias, it is used in place of the actual table name and causes no problem. Just remember the rule: you can't have identical column names if they mean different things.

5.11 The Data "Control" Components

The Data Navigator Bean. This GUI component comes in two flavors, `DSdbNavigator` for AWT and `DSdbJNavigator` for swing. They allow you to navigate through the

database records. Both have the same behavior, which is described in The Navigator and its Functions, in Chapter 8.

Figure 57 The Data Navigator.

5.12 Custom Implementations

5.12.1 Unbound Data

There may be times when you need to compute results that cannot be obtained through SQL queries. Typically these situations arise when the results depend on a computation that involves more than one column, or if it requires a function that is not supported by one of the Aggregate classes. It has become customary to refer to this type of generated data as "unbound data," and we will use the term this way. You can provide added functionality to your application with this flexible technique by adding a separate class that manages the required callbacks. An example follows.

Imagine that the requirement is for a column containing a calculation that requires extra verification logic, or some other calculation not covered by the existing aggregate types. You can extend `AggregateAll` and override its `calculate` method to provide the custom calculation.

See `SummaryUnboundDataExample` for the complete listing. It shows how to locate the node containing the fields you want to work with and add a new summary column containing the derived quantity. An outline of the procedure is given next.

In your main class, append a new summary column to the node's footer:

```
SummaryColumn column = new SummaryColumn("Order Total Less Tax: ");
orderDetailFooterMetaData.appendColumn(column);
```

Provide parameters in the summary column's constructor like these when you want unbound data:

```
column = new SummaryColumn(orderDetailMetaData,
    "jclass.higrid.examples.OrderDetailTotalAmount",
        SummaryColumn.COLUMN_TYPE_UNBOUND,
        SummaryColumn.AGGREGATE_TYPE_NONE,
        MetaDataModel.TYPE_DOUBLE);
orderDetailFooterMetaData.appendColumn(column);
orderDetailFooterFormat.setShowing(true);
```

The second parameter names the class that defines the new calculation, which is presented next. Note that its constructor calls the base class to provide the required

initialization. `OrderDetailTotalAmount` provides the logic for calculations on each row and `AggregateAll` sums them to a group total.

```
package jclass.higrid.examples;

import jclass.higrid.AggregateAll;
import jclass.higrid.RowNode;

/**
 * Calculates the order detail total amount.
 */
public class OrderDetailTotalAmount extends AggregateAll {

public OrderDetailTotalAmount() {
   super();
}

/**
 * Perform the aggregation.
 *
 * @param rowNode The row node.
 */
public void calculate(RowNode rowNode) {
   if (isSameMetaID(rowNode)) {
       Object quantity = getRowNodeResultData(rowNode, "Quantity");
       Object unitPrice = getRowNodeResultData(rowNode, "UnitPrice");
       if (quantity != null && unitPrice != null) {
           double amount = getDoubleValue(quantity) *
               getDoubleValue(unitPrice);
           addValue((Object) new Double(amount));
       }
   }
}

}
```

5.12.2 Batching HiGrid Updates

You can control the frequency at which updates occur. Normally, you want the grid to be repainted immediately after a change is made or a property is set. To make several changes to a grid before causing a repaint, set the `setBatched` property of `HiGrid` object to `true`. Property changes do not cause a repaint until a `setBatched(false)` command is issued.

Thus, when initializing an object, or performing a number of property changes at one time, you can begin and end the section as follows:

```
grid = new HiGrid();
grid.setBatched(true);
grid.setDataModel(new MyDataSource());
grid.setBatched(false);
```

Setting a new data model may cause the grid to request numerous repaints, but these are prevented by sandwiching the command between the two `setBatched` commands.

5.13 Use of Customizers to Specify the Connection to the JDBC

If you have previously made a connection to your chosen database and have saved the information in a serialization file, then all you have to do is launch the customizer and specify the serialization file to reload. The following two figures show the dialogs for the single data level (in `JCData`) and for the hierarchical `JCTreeData`. In either case, you type in the filename or use the **Load...** button to choose it in a file dialog.

5.14 Classes and Methods of JClass DataSource

The following sections describe many of the classes and methods that application programmers find useful.

5.14.1 MetaDataModel

Use the constants in the following table when you want to map a JDBC data type to a Java type. These are the types that are returned. These constants are defined in `com.klg.jclass.datasource.MetaDataModel`.

Java data types used to map JDBC data types
TYPE_BOOLEAN
TYPE_SQL_DATE
TYPE_DOUBLE
TYPE_FLOAT
TYPE_INTEGER
TYPE_STRING
TYPE_BIG_DECIMA
TYPE_LONG
TYPE_SQL_TIME
TYPE_SQL_TIMESTAMP
TYPE_OBJECT
TYPE_BYTE
TYPE_SHORT
TYPE_BYTE_ARRAY
TYPE_UTIL_DATE

5.14.2 Data Model

This is the data interface for the JClass DataSource. An implementation of this interface will be passed to the class using the data source. All data for the data source is maintained and manipulated in this data model through its sub-interfaces. This data model requires the implementation of two tree models, one for describing the relationships of the hierarchical data (`MetaDataModel`) and one for the actual data (`DataTableModel`).

`BaseDataTable` is an abstract implementation of the methods and properties common to various implementations of the `DataTableModel`. This class must be extended to concretely implement those methods not implemented here, which are all of the methods in the `DataTableAbstractionLayer`.

The object that actually holds the data retrieved from the database is `DataTable`. Its implementation is specific to the type of data binding that different sources provide, so its implementation is different in each of the supported IDEs. The following discussion assumes a direct connection to JDBC rather than via an IDE.

`DataTable` contains a copy of the data returned in a JDBC result table, which will be copied into one of these result tables so the data can be cached. Rows can then be added, deleted or updated through this `DataTable`. All operations can access data through row/column `idxToBookmarkMap` rather than indexes. This facilitates sorting of rows and/or columns.

```
public class DataTable extends BaseDataTable
```

Methods:

Method	Description
`buildDeleteStatement(String, Vector, DataTableRow)`	Builds the delete statement for a table.
`buildInsertStatement(String, Vector, Vector)`	Default method for building insert statement.
`buildUpdateStatement(String, Vector, int)`	Default method for building update statement.
`buildWhereClause(Vector, Vector)`	Builds a WHERE clause for update/delete statements.
`cloneRow(int)`	Returns a copy of this row.
`columnModified(int, String)`	Returns `true` if a column value has been modified.
`commit()`	Actually commits this row to the server.
`createNewRow()`	Creates a new row, called by `addRow()`.

Method	Description
`createTable(int, TreeNode)`	Creates and returns the `DataTable` which corresponds to the specified row of this parent for the indicated child `MetaData` object.
`getCell(int, String)`	Returns a value for a given row/column `idxToBookmarkMap`.
`getCombinedKeys(String)`	Returns a Vector of column names which are the join columns and the primary keys for the driver table.
`getOriginalRow(int)`	Given a bookmark, returns the original row as it was before any changes where made.
`getRowFromServer(String, String, int)`	Sends the query to the server, fetches and returns the row.
`originalCellWasNull(int, Vector, Vector)`	Returns `true` if the original cell value is null.
`refreshRow(int)`	Re-reads a row from the originating data source.
`requeryLevel()`	Repopulates this DataTable by re-reading rows from the originating data source.
`restoreRow(int)`	Restores a row's original values.
`saveRow(int)`	Saves row changes to originating data source.
`setParameter(int, Object)`	Sets parameters in the requery.
`setValueAt(int, String, Object)`	Changes the value of an existing cell.
`tablesColumnsModified(int, Vector)`	Returns `true` if at least one of this table's columns been modified.

6

The Data Model

6.1 Introduction

Creating an application with JClass DataSource normally involves these steps:

1. Establishing a database connection.
2. Creating the root meta data table.
3. Defining the meta data for sub-tables.
4. Setting properties, such as the commit policy.
5. Optionally adding generated fields in what are called “virtual columns”.
6. Connecting display objects, like JClass HiGrid.

This chapter illustrates some of the methods that accomplish the above mentioned steps programmatically.

JClass DataSource is structured around two `TreeModel`s, a meta data tree and a data table tree. The classes that make up the meta data model cooperate to define methods for describing the way that you want your data structured. You define the abstract relationship between data tables as a tree. This is the meta data structure, and after it has been designed, you query the database to fill data tables with result sets. The abstract model defines the structure for the specific data items that are to be retrieved and indexed using a dynamic bookmark mechanism. At the base level of the class hierarchy, class `MetaData` describes a node in the `MetaTree` structure and class `DataTable` holds the actual data for that node. There are different implementations of `MetaData` for differing data access technologies, therefore there will be a different `MetaData` defined for the JDBC and for various IDEs. Similarly, there will be different `DataTable` classes depending on the basic data access methodology.

`MetaData` and `DataTable` are concrete subclasses of the classes `BaseMetaData` and `BaseDataTable`. The latter is an implementation of the methods and properties common to various implementations of the `DataTable` model. This class must be extended to concretely implement those methods that it does not, which are all of the methods in the

data table abstraction layer. Both of these classes are derived from `TreeNode`, which contains a collection of methods for managing tree-structured objects.

Interface `MetaDataModel` defines the methods that `BaseMetaData` and its derived classes must implement. This is the interface for the objects that hold the meta data for `DataTable`s. There is one `MetaDataModel` for the root data table, and there can be zero, one, or more `DataTable` objects associated with one meta data object for all subsequent nodes in the meta data model. Thus it is more efficient to store this meta data only once, rather than repeating it as a part of every data table. In JClass DataSource, meta data objects are the nodes of the meta tree. The meta tree, through instances of the `MetaData` classes, describes the hierarchical relations between the meta data nodes. `DataTableModel` is the interface for data storage for the *JClass HiGrid* data model. It requests data from instances of this table and manipulates that data through this interface. That is, rows can be added, deleted or updated through this `DataTable`. To allow sorting of rows and columns, all operations access cell data using unique identifiers for rows and columns, rather than their indexes.

The `DataModel` has one "global" cursor. Commit policies rely on the position of this cursor. This cursor, which is closely related to the bookmark structure, can point anywhere in the "opened" data.

Additionally, each `DataTableModel` has its own "current bookmark." This cursor is used by the `getTable` method to point to a definite row in the named table. If another table is referenced, a new, likely different, bookmark is used as the current row cursor.

6.2 Accessing a Database

Because this product is desiged primarily to populate its data tables from SQL queries, it provides various ways to make the necessary connection to the database or databases that source the data.

6.2.1 Specifying the Database Connection

If you are working on a Windows platform, and wish to test your application using ODBC, register your database as shown in the section on "Setting Up the Data Source" in the *JClass DesktopViews Installation Guide*. Other platforms have similar mechanisms for registering the database with an appropriate driver – consult their documentation for details.

JClass DataSource provides a programmatic mechanism for making a database connection and one based on customizers for those using IDEs. As long as you are using the JDBC API, you may use the JARs that accompany this product in your development environment. If you are using a supported IDE, you may optionally use the IDE-specific data binding solution in the customizer.

6.2.2 Accessing a Database Via JDBC Type 1 Driver

The JDBC-ODBC bridge is part of the JDK. ODBC drivers are commonly available for many databases. Some ODBC binary code is required on each client machine, which means that the bridge and the driver are written in native code. For security reasons, Web browsers may not use the ODBC driver, and therefore Applets must use another approach.

The JDBC-ODBC bridge lets you use the wide range of existing ODBC drivers. Unfortunately, this is not a pure Java solution, and as such may impose an unacceptable limitation. Use of a Type 4 driver, described next, is highly recommended.

6.2.3 Accessing a Database Via JDBC Type 4 Driver

The JavaSoft Web page *http://java.sun.com/products/jdbc/driverdesc.html* has this to say about Type 4 drivers: "A native-protocol fully Java technology-enabled driver converts JDBC calls into the network protocol used by DBMSs directly. This allows a direct call from the client machine to the DBMS server and is a practical solution for Intranet access. Since many of these protocols are proprietary the database vendors themselves will be the primary source for this style of driver. Several database vendors have these in progress." As these become available, Web browsers can use this approach to allow applets access to databases.

```
// A sybase jConnect connection
   DataTableConnection connection=
   new DataTableConnection(
        "com.sybase.jdbc.SybDriver",           // driver
        "jdbc:sybase:Tds:localhost:1498",      // url
        "dba",                                 // user
        "sql",                                 // password
        "HiGridDemoSQLAnywhere");              // database
} catch (Exception e) {
        System.out.println(
        "Data connection attempt failed to initialize " + e.toString());
   }
```

Note: The connection object handles all four types of JDBC drivers, only the parameters names are different as one changes from one driver to another.

6.2.4 The JDBC-ODBC Bridge and Middleware products

You have seen that you can establish a database connection through code similar to this snippet.

```
try {
   // create the connection object which will be shared
   DataTableConnection connection= new DataTableConnection(
        "sun.jdbc.odbc.JdbcOdbcDriver",  // driver
        "jdbc:odbc:HiGridDemo",          // url
        "Admin",                         // user
        "",                              // password
        null);                           // database
} catch (Exception e) {
```

```
System.out.println(
"Data connection attempt failed to initialize " + e.toString());
}
```

There are many JDBC-ODBC bridge products, including one from JavaSoft. The JDBC-ODBC Bridge driver (`package sun.jdbc.odbc`) is included in JDK 1.2.

6.2.5 Instantiating the Data Model

The root level of the model is created by code similar to the following. Here, all the fields from a table named "*Orders*" have been chosen.

```
// Create the Orders MetaData for the root table
MetaData Orders = new MetaData(this, connection,
    " select * from Orders order by OrderID asc");
Orders.setDescription("Orders");
```

The root-level `MetaData` constructor is passed parameters naming the `TreeData` object, the JDBC connection, and the SQL query. The meta data for sub-level tables is instantiated through a command similar to the one shown directly below.

```
// Create the Customer MetaData
MetaData Customers = new MetaData(this, Orders, connection);
```

This constructor takes as parameters a `TreeData` object, the name of the parent meta data object, and the connection object.

If you wish to present fields from more than one table at the same level in the hierarchy, you use the same constructor and the same syntax. The difference only appears when you build the query statement. The next section creates an *OrderDetails* level that is joined to the *Orders* table but obtains data from two database tables, here having the anonymous names *a* and *b*.

```
// Create the Products MetaData
MetaData Products = new MetaData(this, OrderDetails, connection);
String query = "select a.ProductID, a.ProductDescription,
   a.ProductName,";
query += " a.CategoryID, a.UnitPrice, a.Picture, ";
query += " b.CategoryName";
query += " from Products a, Categories b";
query += " where a.ProductID = ?";
query += " and a.CategoryID = b.CategoryID";
```

In the previous code snippet, the two tables are joined to the parent table using the `ProductID` key. That a join with the parent is taking place is recognizable by the use of the *?* parameter that substitutes a particular single *ProductID* value in the parent table to match against ProductID values in table *a*. The two sub-tables themselves are joined on the `CategoryID` key. The next section discusses SQL queries in more detail.

6.2.6 Specifying the SQL Query

JClass HiGrid's customizer permits the point-and-click construction of SELECT statements as one of the essential operations along with naming a database and its tables,

and constructing the grid's meta data. Similarly, JClass DataSource's Beans have custom editors that facilitate building a query. These customizers and custom editors have text panels that permit you to edit the query.

If you do edit the SQL query statement, your more elaborate statement is passed on to the database with only the most rudimentary validation having been done, Therefore, please realize that you must take extra care when testing your code, especially with commands that potentially modify the host database.

6.3 Specifying Tables and Fields at Each Level

Specifying the tables and fields that comprise each level of the hierarchical structure of the grid is really more of a design issue that depends on your particular application rather than any requirement imposed by the data model. Once you have created your design, specify the top level's tables and fields with the command:

```
MetaData Orders = new MetaData(this, connection,
   " select * from Orders order by OrderID asc");
```

This is the constructor for the root level, and is distinguished by the fact that the constructor actually passes a query to the database. For dependent tables, use this form of the constructor:

```
MetaData Territory = new MetaData(this, Customers, connection);
```

As before `this` is a `DataModel` (or `TreeData`) object and `connection` is a `Connection` object, while `Customers` is the name of the parent level. The query is set up using the method `setStatement`:

```
String select = "SELECT TerritoryID, TerritoryName from Territories
   WHERE TerritoryID = ?";
Territory.setStatement(t);
```

Further setup is done with the commands:

```
Territory.joinOnParentColumn("TerritoryID","TerritoryID");
Territory.open();
```

Methods `joinOnParentColumn` and `open` cooperate to return the meta data for the query. The data itself is retrieved when some operation that opens sub-levels is performed.

There is a recurring pattern used to describe and construct the data binding at each level. The commands are:

- Create the level. A root level requires one form of the `MetaData` constructor; all others make use of a second form.
- Define the SQL query for the level as a Java String.

- Provide a descriptive word for the level and pass it to the `MetaData` object via `setDescription`. It's a good idea to ensure that you don't duplicate any of these descriptive words. If you do, you can't be sure which instance the `getDescription` method will return.
- Use `joinOnParentColumn` to name the join fields. This will be checked at run time (or in a custom editor if you are using an IDE or a customizer) against the WHERE clause of the query to confirm that they match.
- Use `MetaData`'s `open` method to load the `ResultSetMetaData` for the level. The retrieval of actual data is deferred until there is a need to display it.

6.4 Setting the Commit Policy

There are three commit policies defined in `MetaDataModel`:

Commit Policy	Description
`COMMIT_LEAVING_RECORD`	Modifications to a row will be written to the originating data source when the cursor moves to any other row.
`COMMIT_LEAVING_ANCESTOR`	Changes will be written to the originating data source when the cursor moves to a row which does not have the same parent as the current row.
`COMMIT_MANUALLY`	Any row changes will simply change the status of those rows (see DataTableModel.getRowStatus()). You must then call DataTableModel.commitRow(bookmark) or DataModel.updateAll() to make the changes permanent, or call DataTableModel.cancelRowChanges(bookmark) or DataModel.cancellAll() to undo the changes. If you are using JClass HiGrid, the end-user can click on the Edit Status column icon to commit edits, or use the popup menu to commit or cancel edits.

By default, edits to a row are committed upon leaving it (the record).

Note that you can find the commit policy currently in effect by calling `getCommitPolicy`, and you can cause all pending updates to be written to the database using `updateAll`. These methods are in classes `MetaDataModel` and `DataModel` respectively.

Also note that `commitAll` should not be used to update the database even though it is declared public. Use `updateAll` instead.

```
// override the default commit policy COMMIT_LEAVING_ANCESTOR
Orders.setCommitPolicy(MetaDataModel.COMMIT_LEAVING_RECORD);
OrderDetails.setCommitPolicy(
```

```
      MetaDataModel.COMMIT_LEAVING_ANCESTOR);
Customers.setCommitPolicy(
      MetaDataModel.COMMIT_LEAVING_ANCESTOR);
Products.setCommitPolicy(MetaDataModel.COMMIT_MANUALLY);
Territory.setCommitPolicy(
      MetaDataModel.COMMIT_LEAVING_ANCESTOR);
```

6.5 Methods for Traversing the Data

Interface `TreeNodeModel` specifies the methods that the nodes of a `TreeModel` must implement. `TreeModel` itself is an interface for the whole tree, including the root, while `TreeNodeModel` refers only to the nodes of a generic tree structure. Both these interfaces are used for meta data objects and for actual data tables. `TreeModel` includes many of the methods of `TreeNodeModel` merely as a convenience.

Method	Description
`append`	Adds a TreeNodeModel to the node upon which the method is invoked. The argument node is added as a child of this node.
`getChildren`	Returns the Vector that contains the child nodes of the node upon which the method is invoked.
`getData`	Returns the Object associated with a TreeNodeModel.
`getFirstChild`	The TreeNode of the first child node for the current data model.
`getIterator`	Given a starting node, a tree iterator is used to follow the links to the node's descendents.
`getLastChild`	Follows the link to the last child table for the current TreeNodeModel; that is, the last table of the group of tables at the meta data level directly beneath the object upon which the method is invoked.
`getNextChild`	Follows the link to the next child table for the current TreeNodeModel.
`getNextSibling`	Follows the link to the next sibling table for the current TreeNodeModel; that is, the next table of the group of tables at the same meta data level as the object upon which the method is invoked.
`getParent`	Returns the parent, as a TreeNodeModel, of the object upon which the method is invoked.

Method	Description
getPreviousChild	Follows the link to the last child table for the current TreeNodeModel; that is, the last table of the group of tables at the meta data level directly beneath the object upon which the method is invoked.
getPreviousSibling	Follows the link to the last child table for the current TreeNodeModel; that is, the last table of the group of tables at the meta data level directly beneath the object upon which the method is invoked.
hasChildren	Use this Boolean method to find out if the object upon which the method is invoked has children.
insert	Inserts a TreeNodeModel as a child of the object upon which this method is invoked.
isChildOf(TreeNode),	Use this boolean method to determine if the object upon which the method is invoked is a child of the TreeNodeModel parameter.
remove	Removes the specified TreeNodeModel from this node's array of children.
removeChildren	Removes the children of the object upon which the method is invoked.

TreeNodeModel defines:

Method	Description
append	Adds a TreeNode to this node.
getChildren	Returns the Vector that contains the child nodes of this node.
getFirstChild	Returns the first child of this node.
getIterator	Returns an iterator to traverse this node's children.
getLastChild	Returns the last child of this node.
getNextChild	Returns the child of this node which follows the node parameter.
getParent	Returns the parent node of this node.
getPreviousChild	Returns the child of this node which precedes the node parameter.
hasChildren	Returns a Boolean: true if this node has children.
insert	Inserts a TreeNode as a child node of this node.

Method	Description
remove	Removes a child node from the Tree.
removeChildren	Removes all children of this node.

TreeIteratorModel defines:

Method	Description
advance	Moves to the next element in this iterator's list.
advance	Moves ahead a specified number of elements in this iterator's list.
atBegin	Returns Boolean: true if iterator is positioned at the beginning of list, false otherwise.
atEnd	Returns Boolean: true if iterator is positioned at the end of list, false otherwise.
clone	Returns a copy of the current node.
get	Returns the current node.
hasMoreElements	Returns Boolean: true if this node has more children, false otherwise.
nextElement	Returns the next child of this node.

6.6 The Result Set

6.6.1 Performing Updates and Adding Rows Programmatically

Performing Updates

JClass DataSource implements all the standard Requery, Insert, Update, and Delete operations. The requery methods are DataModel requeryAll, DataTableModel requeryRow, and DataTableModel (and SummaryMetaData) requeryRowAndDetails.

After a user has modified a cell, call updateCell(rowNumber, columnName, value) to inform the data source of the change. This method will then fire a DataTableEvent to inform listeners about this change. getRowStatus will report this row as UPDATED.

Cancelling pending updates to the database is accomplished via the cancel methods called cancelAll (DataModel) and cancelAllRowChanges (BaseDataTable). See the API for cancelCellEditing in com.klg.jclass.cell.JCCellEditor and its overridden methods in com.klg.jclass.cell.editors for methods which cancel edits to cells.

Requerying the Database

`requeryAll` requeries the root-level of the database – all rows. Not only do the bookmarks get reset, the sub-tables need to be set up from scratch after a `requeryAll`.

Adding a Row

The `addRow` method adds a row and returns a bookmark to the row.

6.6.2 Accessing Rows and Columns

Rows and columns may be accessed in various ways, depending on what information is currently available.

Method	Description
`BaseMetaData.getColumnCount`	The number of columns in the result set.
`MetaDataModel.getColumnIdentifier`	Returns a String that uniquely identifies the column. Used to access data rather than a column index which can change when the columns are sorted.
`DataModelEvent.getColumn`	Returns a String indicating which column changed, or " " in the case where the column is not applicable.
`MetaDataModel.getCurrentBookmark` `DataTableModel.getCurrentBookmark`	Moves global cursor to a row, say by `first`, and return the bookmark.
`DataTableModel.getRowIdentifier(i)`	The index `i` is the row order within the result set. The method returns the bookmark for that row.

6.6.3 Column Properties

Most of these properties are derived from the JDBC class `ResultSetMetaData` in `java.sql`. They are declared in the `ColumnModel` interface.

Property	Description
`getCatalogName`	Returns the catalog name for the table containing this field.
`getDisplayWidth`	Returns the width in pixels of the column.
`getColumnName`	The column's name.
`getPrecision`	The number of decimal digits.
`getSchemaName`	The name of the schema for the table containing this column.

Property	Description
getTableName	The name of the table containing this column.
getColumnType	The Java type of the column.
isAutoIncrement	When a new row containing this column is created, its contents are assigned a sequence number. Some databases permit it to be overridden.
isCaseSensitive	Is upper case to be distinguished from lowercase?
isCurrency	Is the data a currency value?
isDefinitelyWritable	Is the field writable?
isNullable	Is null an allowable value?
isReadOnly	Is the column write protected?
isSearchable	Can this column's contents be used in a WHERE clause?
isSigned	Is the object signed?
isWritable	Is the field writable?

6.7 Virtual Columns

You can add columns whose contents are not retrieved from the data source. The class `BaseVirtualColumn` allows you to add columns which are derived from other columns on the row, including other virtual columns, by performing defined operations on one or more other columns in the row to arrive at a computed value.

Virtual columns are based on `VirtualColumnModel`, an interface with one method: `Object getResultData(DataTableModel, bookmark)`. This allows access to all the other cells in the row.

A base implementation of `VirtualColumnModel` called `BaseVirtualColumn` is provided. It handles the obvious operations you might want to perform on one or more cells in a row: `SUM, AVERAGE, MIN, MAX, PRODUCT, QUOTIENT`[1]. Whether a column is real or virtual, it is transparent to listeners (like `HiGrid`). They simply call `getResultData(bookmark)` as before. The `DataTable` will check the column type. If it is real the normal method is used. If virtual, the `VirtualColumnModel.getResultData(DataTableModel, bookmark)`

1. You can define your own operation by defining a new constant and subclassing BaseVirtualColumn's getResultData method.

method will be called. There can be zero, one, or more virtual columns for a row. Virtual columns will be added by calling a method on the `MetaDataModel`. For example,

```
String name = "LineTotal";
int type = MetaDataModel.TYPE_BIG_DECIMAL;
int operation = VirtualColumnModel.PRODUCT;
Orders.addVirtualColumn(new BaseVirtualColumn(name, type,
   operation, new String[] = {"col1","col2"});

UserDefinedVirtualColumn v = new UserDefinedVirtualColumn(....);
v.setSomeProperty( .. );
Orders.addVirtualColumn(v);
```

Columns are added to the end of the list of existing columns. `VirtualColumn`s cannot be removed.

Computation Order when using Virtual Columns

The implementation of virtual columns requires that the columns referenced by the virtual column must lie to the left of the summary column containing the result. This is usually not a problem because totals and other such summary data are normally placed to the right of the source columns. However, the rule admits of some flexibility because it is the order in which items are added to the meta data structure that determines the left-right relationship referred to above, but the visual layout may be different.

The following code snippet demonstrates the procedure.

```
// Create the OrderDetails MetaData
// Three virtual columns are used:
//
//      TotalLessTax      (Quantity * UnitPrice),
//      SalesTax           (TotalLessTax * TaxRate) and
//      LineTotal          (TotalLessTax + SalesTax).
//
// Thus, when Quantity and/or UnitPrice is changed, these derived
// values reflect the changes immediately.
// Note 1: TaxRate is not a real column either,
// it is a constant returned by the sql statement.
// Note 2: Virtual columns can themselves be used to derive other
// virtual columns. They are evaluated from left to right.
MetaData OrderDetails = new MetaData(this, Orders, c);
OrderDetails.setDescription("OrderDetails");
String detail_query = "select OrderDetailID, OrderID, ProductID, ";
detail_query += " DateSold, Quantity, UnitPrice, ";
detail_query += " '0.15' AS TaxRate ";
detail_query += " from OrderDetails where OrderID = ?";
OrderDetails.setStatement(detail_query);
OrderDetails.joinOnParentColumn("OrderID","OrderID");
OrderDetails.open();
BaseVirtualColumn TotalLessTax  = new BaseVirtualColumn(
   "TotalLessTax",
   java.sql.Types.FLOAT,
   BaseVirtualColumn.PRODUCT,
   new String[] {"Quantity",
       "UnitPrice"});
```

```
BaseVirtualColumn SalesTax      = new BaseVirtualColumn(
    "SalesTax",
    java.sql.Types.FLOAT,
    BaseVirtualColumn.PRODUCT,
    new String[] {"TotalLessTax",
        "TaxRate"});
BaseVirtualColumn LineTotal     = new BaseVirtualColumn(
    "LineTotal",
    java.sql.Types.FLOAT,
    BaseVirtualColumn.SUM,
    new String[] {"TotalLessTax",
        "SalesTax"});
OrderDetails.addColumn(TotalLessTax);
OrderDetails.addColumn(SalesTax);
OrderDetails.addColumn(LineTotal);
```

The `BaseVirtualColumn` constructor is given a column label, the column's data type, the arithmetic operation (one of the supported types), and an array of column names on which the operation is to be applied.

6.7.1 Excluding Columns from Update Operations

Because the `setColumnTableRelations` method explicitly sets the relationships between tables and columns, it can be used to exclude a column from update operations. This is useful in the case of a column containing bitmapped graphics. The database may think that some pixels have changed in the displayed data and the cell should be updated even though the picture has not been edited at all. In cases like this, you can list only those columns that really should be updated, and save the cost of updating a read-only column.

```
// override the table-column associations for the Products table
// to exclude the Picture column so it is not included as part of
// the update. Precision problems cause the server to think it's
// changed.
Products.setColumnTableRelations("Products",
    new String[] {"ProductID","ProductDescription",
    "ProductName","CategoryID","UnitPrice"});
```

Now the column containing the pictures will not be updated.

6.8 Handling Data Integrity Violations

6.8.1 Exceptions

Many files, including the `JCData`, `NodeProperties`, `JCTreeData`, `DataTableModel`, `TreeData`, and `VirtualColumnModel` throw exceptions to alert the environment that actions need to be taken as a result of changes, both planned and unplanned, that have happened as data is retrieved, manipulated and stored to the underlying database.

Since many of these exceptions are specific to the way that data is handled internally, and because extra information is often needed about the details of the exception, a special

class extending `java.lang.Exception` called `DataModelException` is available to supply the extra necessary information.

`DataModelException` adds information about the context of the exception. From it you can determine the bookmark, the column identifier (`columnID`), the action that caused the exception, the `DataTableModel` related to the exception, and the exception itself. There are overridden `toString` and `getMessage` methods that allow you access to the exceptions in readable form.

The following code snippet is just one example of the numerous situations where you might wish to catch a `DataModelException` object. Here, a new `MetaData` object is being created. If the table names are incorrect, or there is a problem accessing the database, the catch block will inform you of the problem.

```
try
{
    String query = new String("");
    query = query + "SELECT * FROM OrderDetail ";
    query = query +
        "ORDER BY order_id,store_id,prod_id,qty_ordered ASC";
    MetaData node = new MetaData(link, connection, query);

    node.setColumnTableRelations("OrderDetail", new String[] {"*"});
    return node;
}
catch (DataModelException e)
{
    ExceptionProcess(); //Print diagnostic and exit
    System.exit(0);
}
```

7

JClass DataSource Beans

7.1 Introduction

JClass DataSource includes nine JavaBeans. Their custom editors simplify the task of making a connection to a database, specifying the master-detail relationships, and binding data-aware components to any level. For designs of the hierarchical or master-detail type, `JCTreeData` is the one to use.

Use `JCData` to bind to one or more database tables at a single level.

Use `DSdbNavigator` (or `DSdbJNavigator` for Swing) as a way of signalling a change to a data pointer. A `DSdbNavigator` can be associated with any level in the hierarchical design that you have defined using a `JCTreeData`. Its buttons are used primarily to request movement to another row in the level to which it is bound, but it has many more capabilities. These are discussed in the following chapter.

The six data bound components `DSdbCheckbox`[1], `DSdbImage`, `DSdbLabel`, `DSdbList`, `DSdbTextArea`, and `DSdbTextField` are used to display information in a column or a field at the level to which they are bound. Other components which are also bound to the same source of data, such as `DSdbNavigator` or JClass HiGrid, are used to move from one row to another, causing the data bound components to update their displays with the new information. `DSdbCheckbox`, `DSdbTextArea`, and `DSdbTextField` are editable components. Changes in their contents are propagated back to the database under the commit policy currently in effect.

`DSdbList` displays a column in the table defined by the current row pointer. It also functions as a navigator. Clicking on one of the items in the list sends a request that the current row pointer be updated. The items in `DSdbList` are not editable.

This chapter describes `JCData` and `JCTreeData`. The navigator and data bound components are described in the following chapter.

1. Swing components are designated DSdbJCheckbox, and so on.

The JClass DataSource Beans connect to database drivers. If you are using Windows and you have `ODBC` drivers installed (perhaps as a result of installing your database software), you can set up an `ODBC` data source and use the JDBC-ODBC bridge. If you haven't done it before, here are the details on setting up a user data source.

1. Double-click on `ODBC` in the *Control Panel*. If it isn't already on top, click on the *User DSN* tab. A list (possibly empty) of *User Data Sources* is visible.
2. Click *Add...* and a setup dialog window appears.
3. Type in a `Data Source Name` and a `Description`. These names may be anything you choose. The `Data Source Name` field is what will be displayed in the *Name* field of the `ODBC` window when the setup is complete.
4. In the *Database* button group, click on *Select...* A file dialog appears, allowing you to type in the full pathname of your chosen database, or navigate to it.
5. If you need to set a *Login Name* and *Password* for your database, click the *Advanced...* button.
6. Click *OK* on the *ODBC Data Source Administrator* window to complete the setup.

If the name you chose was `HiGridDB`, the URL for your data source is:

```
jdbc:odbc:HiGridDB
```

To load the JDBC-ODBC bridge, you use a driver whose name is:

```
sun.jdbc.odbc.JdbcOdbcDriver
```

7.2 Installing JClass DataSource's JAR files

Before getting into the details of the DataSource's JavaBeans, it is important to be able to add these objects to a builder tool. The example chosen is SunSoft's BeanBox. Begin by ensuring that you have installed your *JAR* files in the proper directory for your development environment. In the case of the BeanBox, this would normally be */bdk/jars*. If you prefer to keep your JAR files in another directory, you will have to load them by choosing **File > LoadJar...** A file dialog appears, permitting you to specify your *JAR*'s directory.

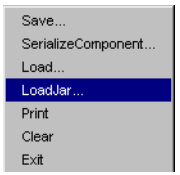


Figure 58 Choosing LoadJar... from the BeanBox's File menu.

The ToolBox displays the Beans contained in the JAR file once they are loaded. Note that these same files are contained in JClass HiGrid, so if you place a HiGrid Bean in the same

environment, you may see a duplicate list of DataSource Beans. It's a good idea to use only one of the DataSource or HiGrid JARs at a time because of their tendency to interact.

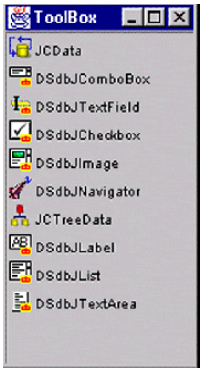


Figure 59 JClass DataSource's Beans, displayed in the ToolBox.

At this point you are ready to add these components to the BeanBox and begin setting them up.

7.3 The Data Bean

Use `JCData` to bind one or more tables at a single level. This Bean is non-hierarchical and is suitable for any application where the data is to be presented all within a non-expandable grid. The fields in the grid may be chosen from more than one database table.

7.3.1 Setting a Data Bean's Properties and Saving Them to a Serialization File

The first step in using this Bean is to add it to your IDE. It is important that you use the JAR corresponding to the data source connection mechanism you intend to use. Use `jcdatasource.jar` if you are going to use a connection based on JDBC. If you intend to use an IDE-specific connection, install the JAR whose name includes the initials matching the IDE.

We'll use the ToolBox to illustrate the general method. After placing `JCData` in the BeanBox, a window opens reminding you to name and save the serialization file.

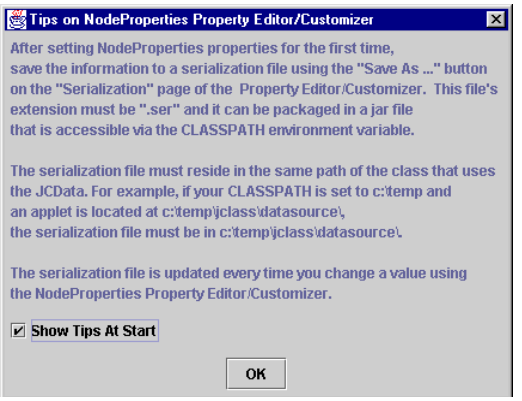


Figure 60 The Data Bean tip window: a reminder to save the serialization file.

For more information on saving the serialization file, see Section 7.3.3, Saving a Serialization File.

7.3.2 The Data Bean Editor

Observe the *Property Sheet* which opens when `JCData` is placed on the BeanBox. The data Bean's *Properties* sheet is shown in Figure 61.

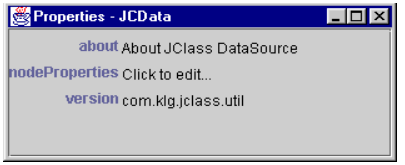


Figure 61 The BeanBox's Properties sheet, showing the properties of the Data Bean.

The middle line has a property called `nodeProperties`, whose pseudo-value is **Click to edit...**. Clicking on this item brings up a custom editor, the `JCData` Bean Component Editor.

The `JCData` Bean Component Editor contains an array of tabbed dialogs that permit you to set a large number of properties. These are discussed in the following sections.

7.3.3 Saving a Serialization File

The `JCData NodePropertiesEditor` initially shows its **Serialization** tab. Choose a name for the serialization file or use the **Save As...** button to create the file.

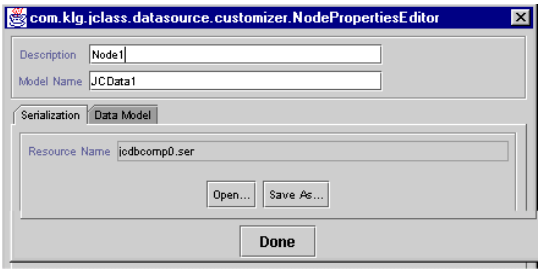


Figure 62 The Serialization tab of the NodePropertiesEditor.

Once a serialization file has been saved, the property editor updates it as you make changes to any of the properties in the data Bean. When subsequent design changes are made, begin by loading the serialization file. This can save time because it stores all the settings that have already been made.

7.3.4 Making a Connection to a Database

Follow these steps to directly connect to a JDBC driver supported database, or to use an IDE-specific data binding mechanism.

1. Click the **Data Model** tab, exposing another level of tabbed choices.
2. Fill in the fields in the **Connection** pane.
3. Type the URL and the driver name in the property editor. Leave the other fields blank unless you are connecting to a database or middleware server over a network.

4. Click on **Connect**. There is a message area just below the **Connect** button that informs you whether the connection attempt was successful or whether it failed and you are in for a troubleshooting exercise.

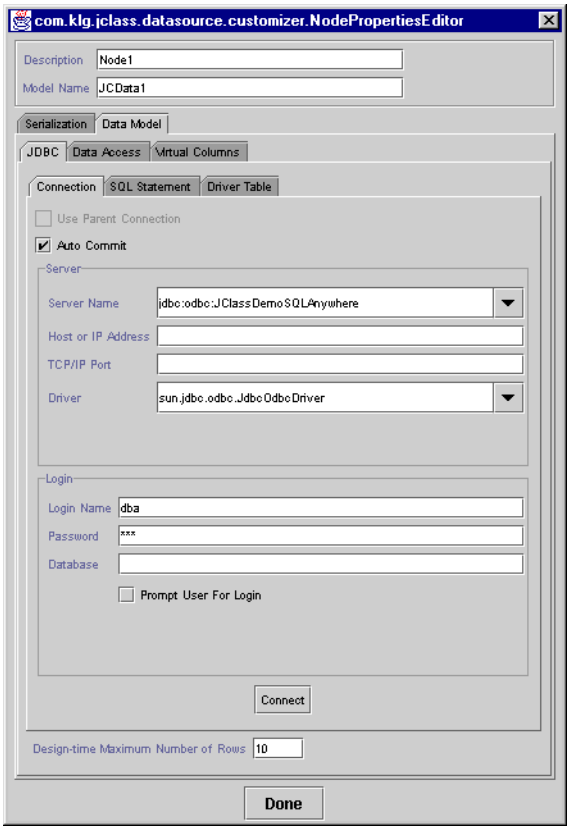


Figure 63 The Connection page of the data Bean's custom property editor.

Note that you can set the *Design-time Maximum Number of Rows* to limit the amount of data that the database must furnish at design-time. This saves time and memory if the query normally returns a large amount of data that is quite unnecessary at design-time.

Using a non-JDBC-ODBC driver

In the case of drivers that require a host address and a TCP/IP port specification, the database name must be given separately, rather than associating it with an alias as is the case with an ODBC setup. Use the following example as a guide when configuring this

type of database connection. In the example, the name of the host is *gonzo*. (The driver is a FastForward type 4 driver for Sybase Sql Server.)

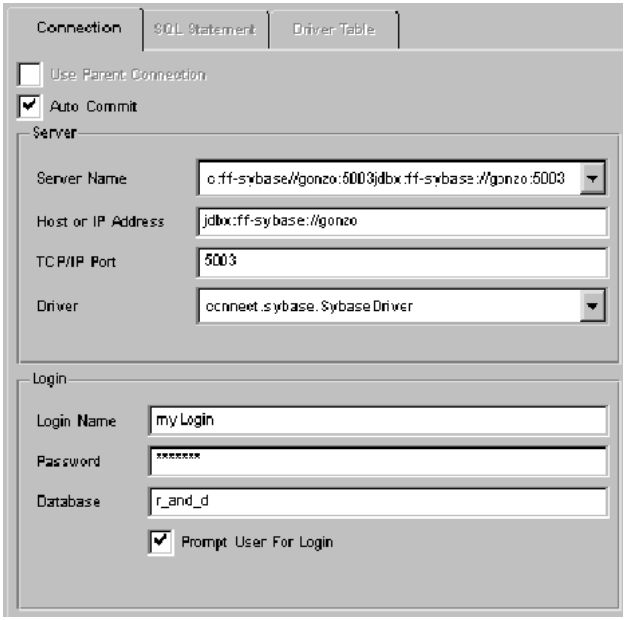


Figure 64 A database connection that requires every field to be filled in.

7.3.5 Choosing Tables

This section applies if you are using the JDBC connection. After a connection is established, the **SQL Statement** tab is accessible, as shown in Figure 66.

1. Click on the **SQL Statement** tab. There are two scrollable panes. The table selector area is the space reserved for the table or tables that you are going to refer to in the chosen database. After being chosen, the table appears as a scrollable pane containing a list of all its fields. The *SQL Statement* area is directly below the table selector area.
2. Click on **Add Table...** or right-click in the table selector area and a *Table Chooser* pop-up menu appears. Select the database table you want and click the **Add** button in the *Table Chooser* window. Notice that a FROM clause naming the table appears in the *SQL Statement* area.
3. Use the customizer to generate SQL statements from mouse actions. See the next section for details.
4. To choose more than one table, repeat the process for selecting tables. Click **Add** for each.

7.3.6 Choosing a Query

The customizer gives you two ways to form a query. You can type the query directly in the SQL Statement area, or you can use the mouse. See Figure 66 to see how a table and its associated query appear in the customizer.

Simple queries for selecting which fields of the table to display are usually accomplished automatically using mouse actions on the **SQL Statement** tab. Choose the fields by double-clicking on them, or by clicking on the **Add Selected Column(s)** button. You'll notice that the text for the query appears in the *SQL Statement* text area as you use the mouse to choose fields. For more elaborate queries, directly type it in the *SQL Statement* text area. Whatever text appears in this area is used as the SQL statement for retrieving the data from its source.

7.3.7 Joining Two Tables: Driver Table

If your application needs to present information that is stored in more than one table, you can perform a database join on the tables. One simple way is to use the *Auto Join* feature.

1. Click on **Add Join** in the **SQL Statement** group of panels. A *Join* window appears.

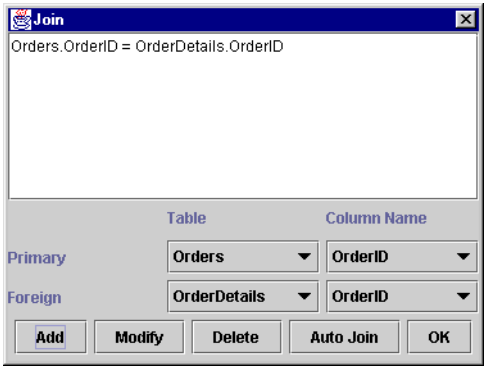


Figure 65 The Join window, where you can perform a database join.

2. Select the "Primary" table and the "Foreign" table from drop-down lists.
3. Click **Auto Join**. The customizer looks for a foreign key in the "Foreign" table that matches the primary key in the "Primary" table. If it finds a match it places a WHERE clause fragment in the text area of the *Join* window.
4. Click OK and observe that the complete SQL statement, including the WHERE clause, appears in the *SQL Statement* panel.

5. Save your SQL statement by clicking on the **Set/Modify** button. Your window will look something like that in the next figure.

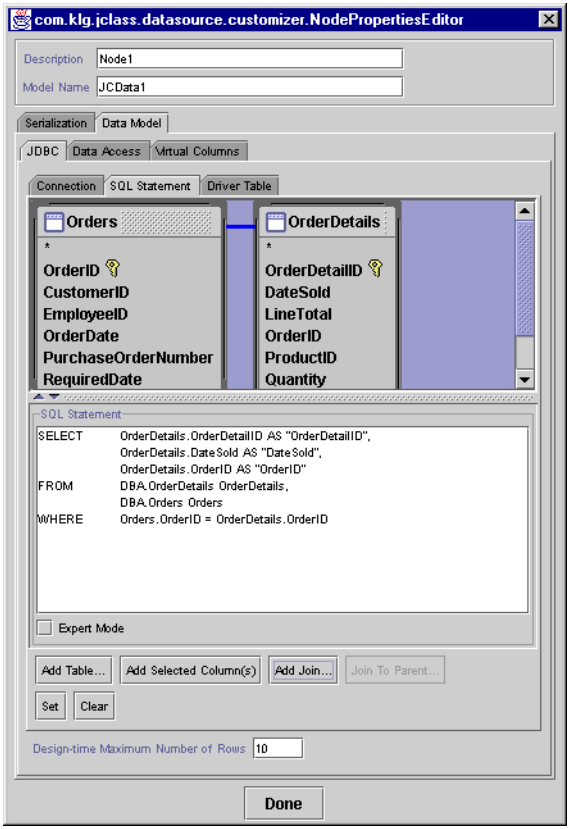


Figure 66 The SQL Statement page, showing a completed query.

Click **Done** to close the window. The serialization file has captured all the changes so long as the **Set** button was clicked to save any changes made to the tables, such as which fields are selected and how tables are joined.

7.3.8 The Driver Table Tab

One of the possible operations on data tables, once they have been retrieved from the database, is the requery of a single row. If a row is formed from the fields of more than one table, the *Driver Table* is the one whose primary key can be used to drive this type of requery. This is best illustrated with an example.

For the requery to succeed, there needs to be some way of uniquely specifying the row when it contains data from two tables. A specific case, drawn from a sample database, is a join of the *Customers* table with the *Salespeople* table. The query that returns the desired result set is:

```
SELECT Customers.CustomerID AS "CustomerID",
       Customers.CompanyName AS "CompanyName",
       Salespeople.SalepersonID AS "SalepersonID",
       Salespeople.Name AS "Name"
FROM   DBA.Salespeople AS Salespeople,
       DBA.Customers AS Customers
WHERE  Customers.SalespersonID = Salespeople.SalepersonID
```

For the requery of a single row to work correctly, we must know that the most restrictive of the two tables is *Customers*, in the sense that the result set contains rows that have unique *CustomerID* values. On the other hand, there may be duplicate or repeated values for *SalespersonID*, so the way to uniquely specify a row is to refine the original query by adding to the WHERE clause the value for the row's *CustomerID* field.

The *Driver Table* panel lets you specify which table, and which key, to use when JClass DataSource needs to requery a single row.

1. Click on the **Driver Table** tab.
2. Choose a table from the *Table* drop-down list.
3. Choose the key from the *Column Name* drop-down list.

If no driver table is chosen, JClass DataSource uses the first table named in the FROM clause whenever it needs to requery a single row.

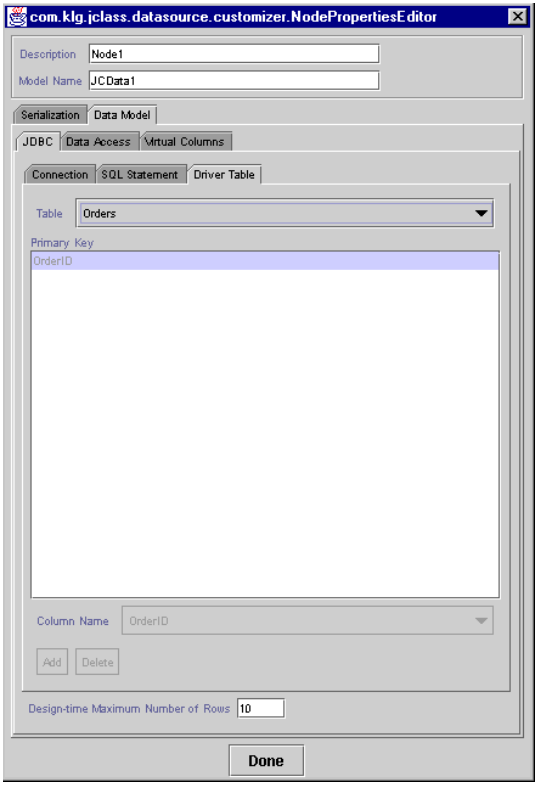


Figure 67 The Driver Table tab.

7.3.9 The Data Access Tab

Use this panel to set the overall commit policy and the access rights.

The *Commit Policy* group has these options:

- The Commit Policy itself has these choices: `COMMIT_LEAVING_RECORD` and `COMMIT_MANUALLY`. See the discussion on commit polices in Commit Policy, in Chapter 5, and `MetaDataModel.setCommitPolicy` in the API.
- The *Show Deleted Rows* checkbox is unused.

The *Table Access* group simplifies the task of setting access permissions for all three types of SQL update operations. The permissions are set on a per-table basis.

1. Choose a table from the *Table* drop-down list.

2. Use the *Insert Allowed*, *Update Allowed*, and *Delete Allowed* checkboxes to set access permissions for the table.
3. Click **Add** to record the settings in the *Table Access* text area and apply the permissions you have set.
4. Choose another table, set permissions using the checkboxes, and click **Add**.
5. To edit any of the settings you have made, select a table, adjust the access permissions and click **Modify**.

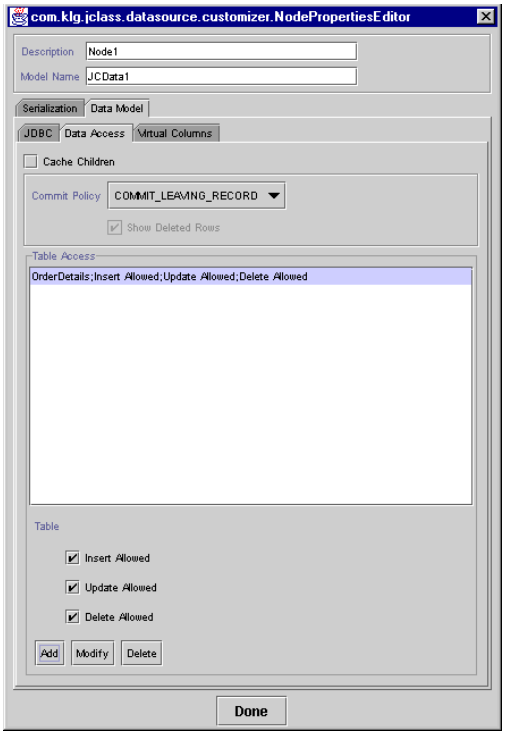


Figure 68 The Data Access tab.

7.3.10 The Virtual Columns Tab

JClass DataSource supports the use of computed fields as well as fields retrieved from a database. Use the *Virtual Columns* panel to define a computed field that occupies the same position in every row of the chosen table. Use this tab to define additional columns that

perform one of the supported types of aggregation: *Average*, *Count*, *First*, *Last*, *Max*, *Min*, and *Sum*.

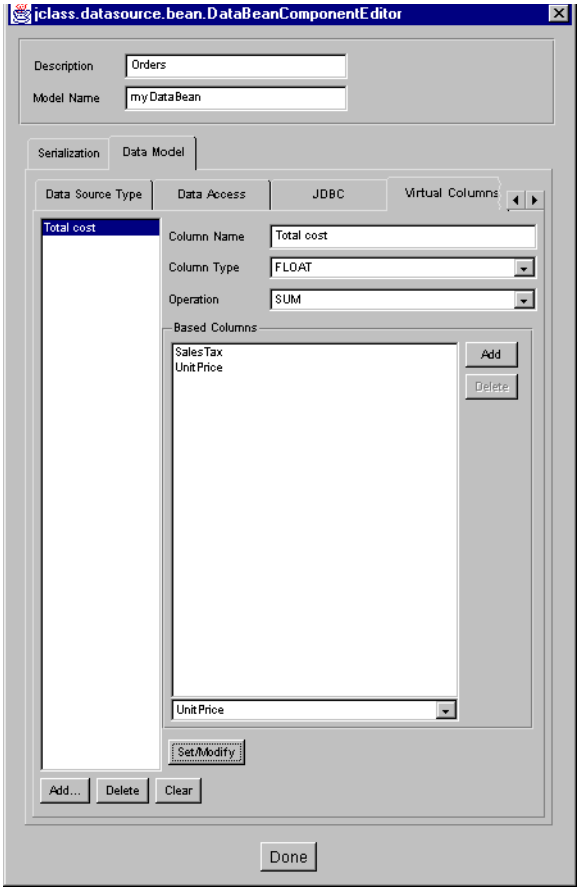
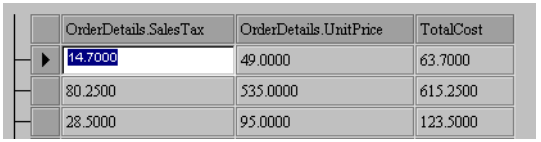


Figure 69 The Virtual Columns tab.

To define a virtual column:

1. Decide on a name for your virtual column and type it in the *Column Name* text area.
2. Select the data type from the *Column Type* drop-down list.
3. Select the type of aggregation from the *Operation* drop-down list.
4. Use the drop-down list in the *Related Columns* area to choose the fields upon which the aggregation will be based. Click **Add** to place the column in the text area.
5. Click **Set/Modify** to complete the operation.

The example shown in Figure 69 adds a virtual column called *TotalCost* which is the sum of database fields *SalesTax* and *UnitPrice*. The result is shown in the next figure.



	OrderDetails.SalesTax	OrderDetails.UnitPrice	TotalCost
▶	14.7000	49.0000	63.7000
	80.2500	535.0000	615.2500
	28.5000	95.0000	123.5000

Figure 70 An example of a virtual column whose aggregate type is SUM.

Note that all the fields used by a Virtual Column to generate a value must lie to its left.

7.3.11 Displaying the Result Set

`JCData` is ready to execute your query. The result can be displayed using a `HiGridBean` or a `LiveTable` Bean. The following steps show how to connect a `HiGridBean` to a `JCData` in the BeanBox.

1. Place a HiGrid JavaBean on the BeanBox. Select the `JCData` and choose **Edit > Events > dataModel > dataModelCreated.**
2. Join the BeanBox's rubber band to the HiGrid Bean. Select `dataModelChanged` from the popup menu. If you don't see this choice it probably means you have selected some other object besides the HiGrid - you have to select its outline, and since the outline isn't visible while you are trying to find it, the operation reduces to a challenge in precise pointing. An event is fired, and the grid is updated. After resizing, you should see the result set from your query displayed in the HiGrid.

7.4 The Tree Data Bean

A `JCTreeData` Bean is capable of displaying master-detail relationships in indented tabular form. Its customizer uses the full power of JClass DataSource while making it easy to transfer your hierarchical design to Java code.

Placing a `JCTree` data JavaBean on a form is the same as using a `JCData` Bean. As in the discussion for the `JCData` Bean, you begin by clicking on the filename at the right of the *treeDataBeanComponent* label on the *JCTreeData Bean*'s property sheet.

This invokes the *Tree Properties Editor*, the custom editor for this component. The **Serialization** tab is the same as in the case of the *JCData Bean Properties Editor*, but the left

hand panel has a different appearance. The outliner for the hierarchical design occupies this area. See Figure 71.

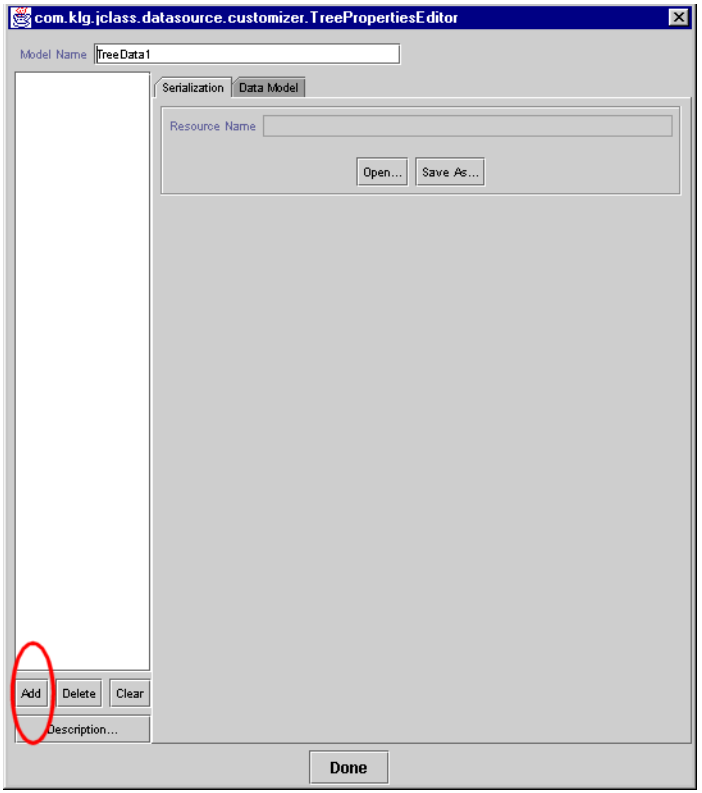


Figure 71 Before a table is added, you are asked to supply a descriptive name.

The database connection is accomplished just as it is in the case of the *JCData Bean* component. The way that tables are installed is different because you are able to use this Bean to design a hierarchical data model.

Important: To add the parent table to the form, click the **Add** button at the lower left of the outliner panel. A warning dialog like that shown in Figure 72 appears reminding you

to save a serialization file. Type in the name of your root data table (in place of the name *Node0*) in the upper left-hand portion of the text pane.

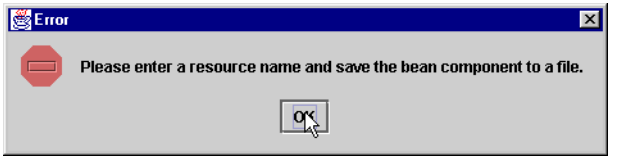


Figure 72 Name the root data table after clicking the Add button in Figure 71.

You have the beginnings of your data design, as shown in Figure 73.

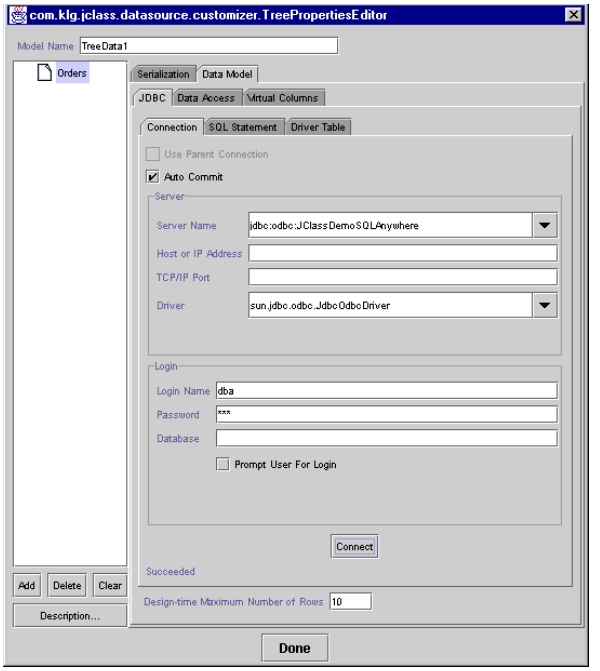


Figure 73 The Connection tab for the Tree Data Bean component.

At this point the **SQL Statement** tab becomes active. Click on it and add the *OrderDetails* table to your form with the aid of the *Table Chooser* menu. Tables may be chosen by double-clicking on an item, or by highlighting the item and clicking the **Add** button.

Use the **Table Chooser** dialog to review a list of all available tables. You can choose more than one table at the parent level, but one of these should be selected as the *Driver Table*. See Section 7.3.8, The Driver Table Tab, for the step-by-step process.

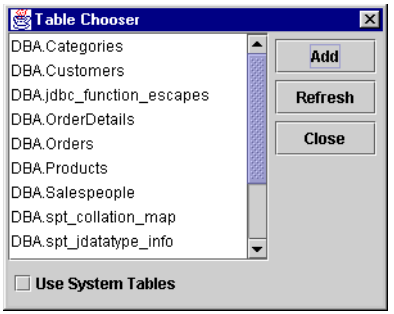


Figure 74 The Table Chooser window.

The completed form is shown in Figure 76. A hierarchical data design has been defined and is now ready for connection to an object that can display the results. As in the case with `JCData`, a `JCHiGrid` Bean is used to display the data.

7.4.1 The Driver Table Tab

If there is more than one table at a given level, a *DriverTable* should be declared. This is accomplished with the **Driver Table** tab as shown in the next figure.

The driver table is the one that the database uses to drive the query. If a driver table is not specified in this dialog the database will choose one, but it is not easy to tell which of the candidate tables it will be.

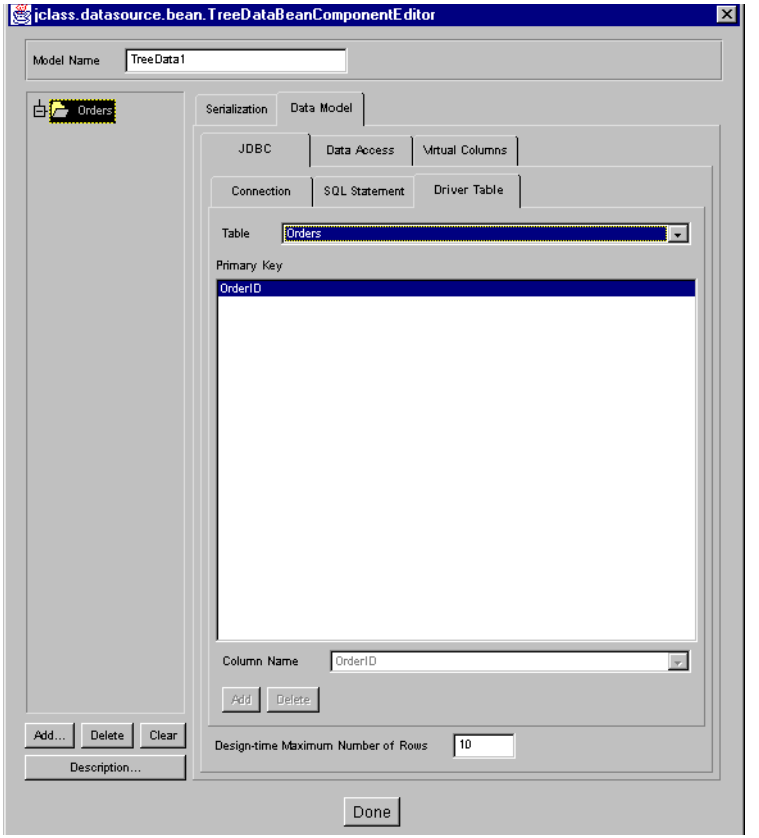


Figure 75 The Driver Table tab.

Here is a completed *SQL Statement* panel for a sub-table called *OrderDetails*.

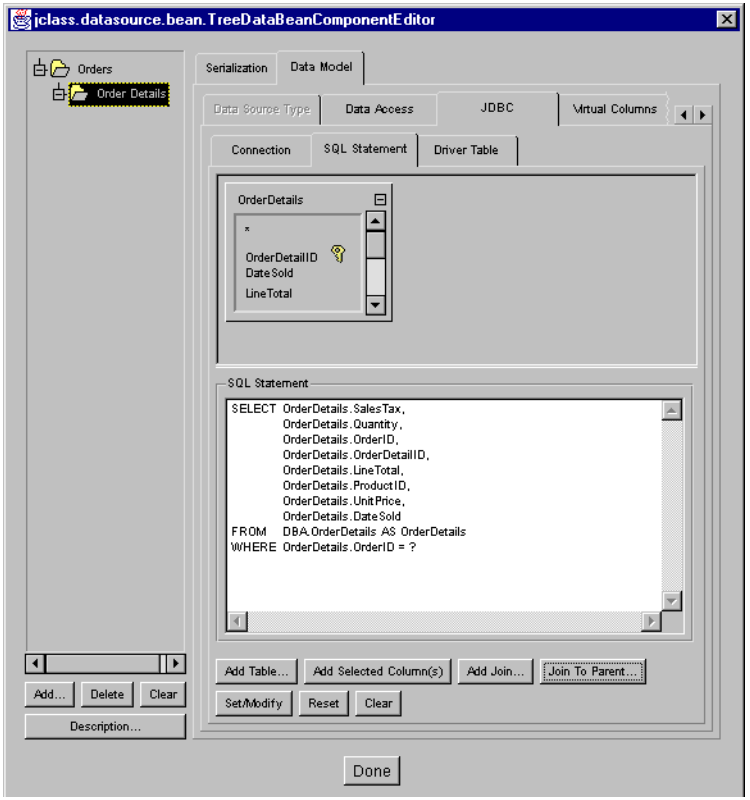


Figure 76 Adding a detail-level table and selecting a query statement.

The Tree Data Bean may be attached to any component capable of displaying a hierarchical grid, such as JClass HiGrid. It is possible to attach data bound components to any level in the hierarchy.

7.5 The Data Navigator and Data Bound Components

JClass DataSource has several beans, including a data navigator and a group of data bound components. The data navigator can be bound to any level in a master-detail hierarchy. Through its row-positioning mechanism, it fires events that notify the other data bound controls that they need to update themselves with the data from the new row.

The navigator and the data bound controls are discussed in detail in DataSource's Data Bound Components, in Chapter 8.

7.6 Custom Implementations

7.6.1 Using the DataSource Bean in an IDE

JClass DataSource is designed to be used in an IDE. Use the DataSource Beans' powerful customizers to set up the database connection, build a query in a point-and-click fashion, and bind the retrieved data to a grid, or other data bound component for display. The upcoming sections demonstrate the use of such a customizer. The next section shows how to add the JAR file to a specific IDE so that you can begin using the JClass DataSource's JavaBeans.

7.6.2 Data Binding in Borland JBuilder

If you intend to use Borland JBuilder's own method of forming a database connection, follow these steps *before adding JClass DataSource components to your form*:

1. After beginning your applet or application, click on the **Data Express** tab in the *Component Palette*, select the component labeled *borland.sql.dataset.Database* and add it to your form.
2. A *connection* window appears. Choose the URL for your database connection or type it in the *Connection URL* text field. Also supply information for the *Username*, *Password*, and *Driver class* text fields.
3. Place a *borland.sql.dataset.QueryDataSet* on the form. A *query* window appears. Choose the database connection object from the *Database* drop-down list and type the query in the *SQL Statement* text area.
4. Now add a JClass data Bean to the form. On the **IDE** tab, choose **Borland JBuilder** and type the name of the `queryDataSet` object in the *Data Source Name* text field.

The JClass data Bean is now ready for use within the Borland JBuilder data binding scheme.

8

DataSource's Data Bound Components

8.1 Introduction

JClass DataSource and JClass HiGrid work as a team to provide a flexible data binding solution for those applications that need to present hierarchically organized data in an integrated package. JClass DataSource by itself is able provide your application with a number of SWING-like components grouped on a form and bound to a hierarchical source of data. It contains a versatile set of components that can be bound to any source of data that JClass DataSource can access, and it provides the navigation tool for choosing any of the records in the data set to which it is bound. The same data binding mechanism is available for use in JClass Chart, JClass Field, and JClass LiveTable as long as all products have matching version numbers.

8.2 The Types of Data Bound Components

JClass DataSource contains SWING-type components. If you are using the JClass DesktopViews product suite, you are able to bind JClass Chart, JClass Field, and JClass LiveTable objects in addition to the set of components included in JClass DataSource.

The "standard" components and their associated data bound component names are given in the table.

Swing		Types
JCheckBox	DSdbJCheckBox	String, Numeric
	DSdbJImage	java.awt.Image
JLabel	DSdbJLabel	String, Numeric
JList	DSdbJList	String, Numeric
	DSdbJNavigator	void

Swing		Types
JTextArea	DSdbJTextArea	String, Numeric
JTextField	DSdbJTextField	String, Numeric

Editable components are: DSdbJTextField, DSdbJTextArea, DSdbJCheckBox. The non-editable components are DSdbJLabel, DSdbJList, DSdbJImage, and DSdbNavigator.

The Navigator is derived from either SWING Panel classes, and it is included in the table because it functions much the same way as the other components.

JClass DataSource's API makes it possible for you to bind SWING, or even components of your own making, to a data source. The next sections illustrate how this is done.

Binding a Component to a Meta Data-Level

Each component Bean has a setDataBinding property to simplify the task of specifying the data connection. This method is called automatically by the component's property editor in an IDE environment. The next section discusses the programmatic method.

Binding the Component Programmatically

Programmatically, data binding is accomplished by calling the setDataBinding constructor in one of two ways. The "standard" method is to provide handles to the DataModel and the MetaDataModel themselves. A second way of representing the MetaDataModel is by a "path" of MetaDataModel descriptions separated by "|" (for example, Orders|Customers).

Binding the navigator component to a data source requires only references to a DataModel and a MetaDataModel. For example:

```
DSdbNavigator nav = new DSdbNavigator();
nav.setDataBinding(dataModel, metaDataModel);
```

To bind a component that displays a single database field, such as a text field, requires a column name as a third parameter in the call to the setDataBinding method:

```
DSdbTextField dbCustomerID = new DSdbTextField();
dbCustomerID.setDataBinding( dataModel, metaDataModel, "CustomerID");
```

Binding the Component through an IDE

There are more choices when you effect data binding using an IDE. The recommended way is to use JClass DataSource's JCData or JCTreeData and JDBC to specify the connection to the data source. Alternatively, you provide the instance of the DataModel and path, just as in the case of programmatic data binding. Finally, you can provide a single String containing the name of the DataModel, separated by a colon, from the path to the chosen MetaDataModel. For example:

```
setDataBinding("DataModel0:Orders|OrderDetails").
```

Using the JClass DataSource Data Bound Components

The data bound components have been made especially easy to use in an IDE by providing a customizer that communicates with any `JCData` or `JCTreeData` that has already been created and connected to a source of data. Use this customizer to select the `DataModel` and `MetaDataLevel`. Once these have been selected, a list of column names is presented. Once a name has been selected, the data bound component is ready for use.

If you decide to use the programmatic API, the data bound component's constructor takes three parameters whether it binds to the entire column, in the case of `DSdbList`, or to a single cell for all the rest. Taking `DSdbTextField` as an example, its constructor is:

```
public DSdbTextField(DataModel dataModel,
   MetaDataModel metaDataModel,
   String column_name)
```

There is also a parameterless constructor that requires data binding to be set using `setDataBinding`, which takes the same three parameters. Use this form of the constructor when you need to instantiate the component first and set the data binding later.

8.3 The Navigator and its Functions

8.3.1 Introduction

`DSdbNavigator` is a visual component that fires events to JClass DataSource, requesting a move to another row in the table to which it is bound. In addition to buttons for movement to the first, last, next, and previous rows, it is able to request the insertion of a new row or the deletion of the row to which it is currently pointing.

It is bound to a data source by giving its constructor references to the `DataModel` and a particular `MetaDataModel` in the hierarchy. Thus, it can be bound to any level in the master-detail structure.

Swing Support

Since data bound components have been defined in JClass DataSource for SWING, a navigator exists for this environment. The Swing navigator is based on `JComponent` and is called `DSdbJNavigator`.

The navigators are in the same packages as the other JClass DataSource data bound components. The Swing navigator is called `com.klg.jclass.datasource.swing.DSdbJNavigator`.

8.3.2 The Navigator Binds to any MetaData Level

The navigator binds to any `MetaData` level, just like the other `DataSource` data bound components. It uses `com.klg.jclass.datasource.TreeData` to bind to a particular node in the hierarchical data source. The property is called `DataBinding`, just like all the other

data bound components in JClass Chart, JClass Field, JClass LiveTable, and JClass DataSource.

A `DSdbNavigator` constructor is parameterless; therefore, the newly instantiated component is not initially bound to a data source. Binding occurs in various ways, depending on whether the IDE or programmatic approach is taken.

Binding the Navigator Programmatically

Programmatically, data binding is accomplished by calling the `setDataBinding` constructor in one of two ways. The "standard" method is to provide handles to the `DataModel` and the `MetaDataModel` themselves. A second way of representing the `MetaDataModel` is by a "path" of `MetaDataModel` descriptions separated by "|" (for example, Orders|Customers).

Binding the Navigator through an IDE

There are more choices when you effect data binding using an IDE. The recommended way is to use JClass DataSource's `JCData` or `JCTreeData` and JDBC to specify the connection to the data source. Alternatively, you provide the instance of the `DataModel` and path, just as in the case of programmatic data binding. Finally, you can provide a single `String` containing the name of the `DataModel`, separated by a colon (:), from the path to the chosen `MetaDataModel`. An example is `setDataBinding("DataModel0:Orders|OrderDetails")`.

8.3.3 DSdbNavigator's Functions

The JClass DSdbNavigator component displays the current row of the data table to which it is bound. Four of its buttons, `First`, `Previous`, `Next`, and `Last`, signal the data source to adjust its current row pointer. The `Insert` button requests the insertion of a new row and the `Delete` button requests the deletion of the current row from the data source. The `Command` button pops up a sub-menu of additional choices. The layout of the navigator's buttons is shown below:

Figure 77 The DSdbNavigator component.

The central `Status` field displays the description for the meta data level, the current record number, and the total number of records in the data table to which it is bound. The navigator's buttons are described below, beginning at the right and preceding in order to the left:

Command	Description
`First`	Moves to the first row in the current DataTable.

Command	Description
Previous	Moves to the previous row in the current DataTable. If already at the start, no move occurs.
Delete	Deletes the current record.
Command	Pops up a menu of commands that can be executed. The menu is similar to the one in JClass HiGrid that pops up by right-clicking on one of the grid's rows.
Status	Displays the name given to the meta data level to which it is bound, the Data Table record number, and total number of records in that Data Table.
Next	Moves to the next row in the current DataTable. If already at the end, no move occurs
Last	Moves to the last row in the current DataTable
Insert	Adds a new record in the current table at the end of the table.

The Swing version of the navigator uses tool tips to show what each of the buttons does. The tool tip's text is derived from the text in the table above.

The **Command** menu pops up a sub-menu that allows operations on a table similar to those allowed by HiGrid. A list of **Command** menu commands is shown after the figure that illustrates it:

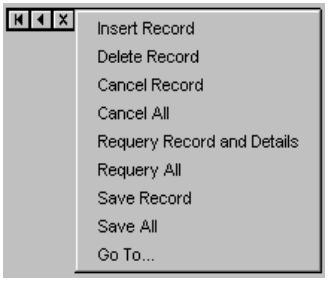


Figure 78 DSdbNavigator, showing the Command menu.

Command	Description
Insert Record	Adds a new record in the current table. Same as the add button in the navigator.
Delete Record	Removes the current record in the current table.

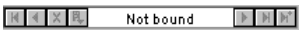
Command	Description
Cancel Record	Cancels the current edit.
Cancel All	Cancels all edits made.
Requery Record and Details	Requeries the table from the database.
Requery All	Updates the current row in the database.
Save Record	Saves changes made to the current record.
Save All	Updates all changes made.
Go To	Pops up a dialog that allows specification of a new row number.

8.3.4 Exploring DSdbNavigator's Bean Properties

Binding a navigator to a data source in an IDE is accomplished through the use of its data binding editor. The editor is aware of the data sources that you have pre-configured, so it's important to add a `JCData` or a `JCTreeData` to your form before you use the navigator's data binding editor.

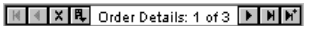
Here are the steps to bind a navigator to a data source:

1. Place a `JCData` or a `JCTreeData` on your form.
2. Use the Data Bean's customizer to specify the connection to the database.
3. Place a `DSdbNavigator` (or a `DSdbJNavigator`) on your form. Its display area (called the *status* area) indicates that it is not bound to a data source.



4. Click **Select a Data Source** to launch its data binding editor.
5. A diagram of the meta data structure appears. If necessary, expand the diagram to show all the nodes. Click on a node to select it. Press **Done** to bind the navigator to the chosen level.

6. Check that the navigator confirms that it is bound to a data source by reporting the meta data level to which it is bound in its display area.



When a DSdbNavigator is placed in the BeanBox or an IDE, you'll see the properties listed in Figure 79.

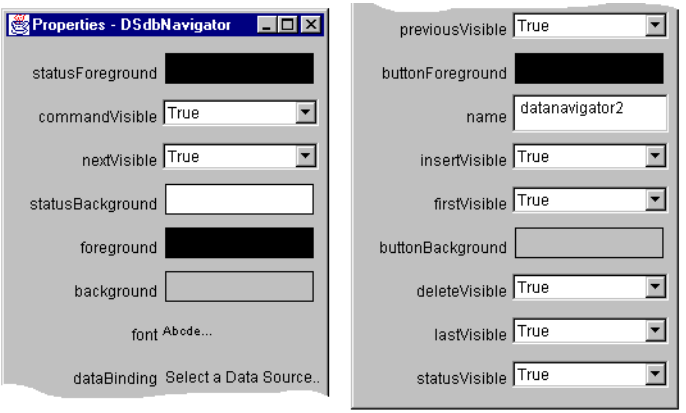


Figure 79 The properties of DSdbNavigator.

Each region has the following properties:

Property	Description
`Visible`	Determines whether the region is shown.
`Foreground`	Foreground color.
`Background`	Background color.

Note that all of the buttons must have the same color, but the color of the status area can be set independently of the button color. Set the background and foreground colors for the buttons using `setButtonBackground` and `setButtonForeground`. The color is applied to all the buttons as a group. Set the colors for the status area using `setStatusBackground` and `setStatusForeground`.

Each button has its own get and set methods for reading and controlling its visibility. For example, use `setCommandVisible(false)` to hide the `Command` button.

The property names for controlling visibility are based on the region names as shown below:

Button or Status Area	Visible Get/Set Method
First	FirstVisible
Previous	PreviousVisible
Delete	DeleteVisible
Status	StatusVisible
Command	CommandVisible
Next	NextVisible
Last	LastVisible
Insert	InsertVisible

The following figure shows the data binding editor window which appears as a result of clicking on **Select a Data Source...** in the properties list.

It shows an example of an expanded view of the data model that was created to accompany the steps in the data binding procedure given above. The navigator was bound to the *Order Details* level by clicking on its name, then clicking **Done**.

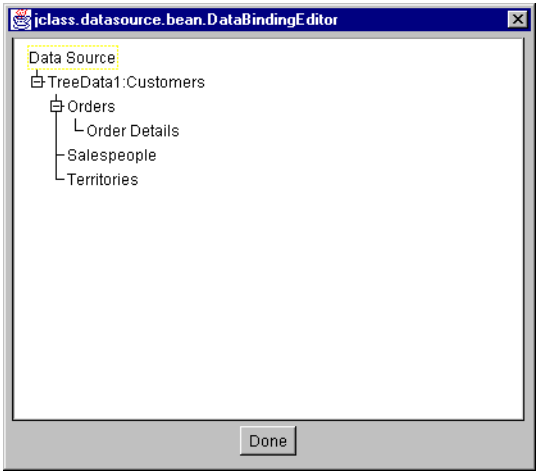


Figure 80 DSdbNavigator's data binding editor.

8.4 Data Binding the Other Components

A component that binds to a single database field requires a column name in addition to the data model and meta data level. In an IDE, the binding is done following the same steps as is the case for `DSdbNavigator`. The next figure shows a cutout of a `DSdbTextField`, its exposed properties, and its `ColumnDataBindingEditor`.

The text field is bound to a column called *Territory Name*, which is part of the meta data level called *Territories*. All the database field names (that is, the column names) appear in the editor. The name appears highlighted after it has been chosen with a mouse click.

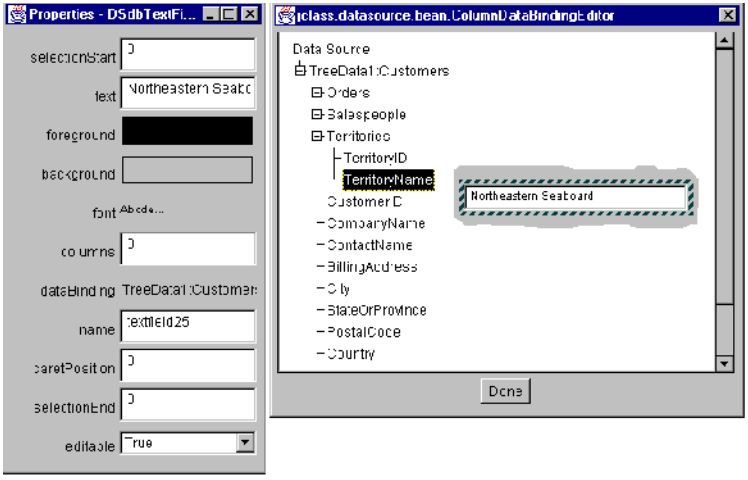


Figure 81 A DSdbTextField, its Properties, and its ColumnDataBindingEditor.

9

Sample Programs

9.1 The Sample Database

The sample database that is included with JClass DataSource has the structure shown in Figure 92. This diagram shows the table names, column (field) names, and data types. Many-one relationships are shown by terminating the dotted lines connecting two tables with a black circle at the "many" end of the relationship. The key fields are shown at the top of each table and the foreign keys are designated by placing the tag "(FK)" after the data type.

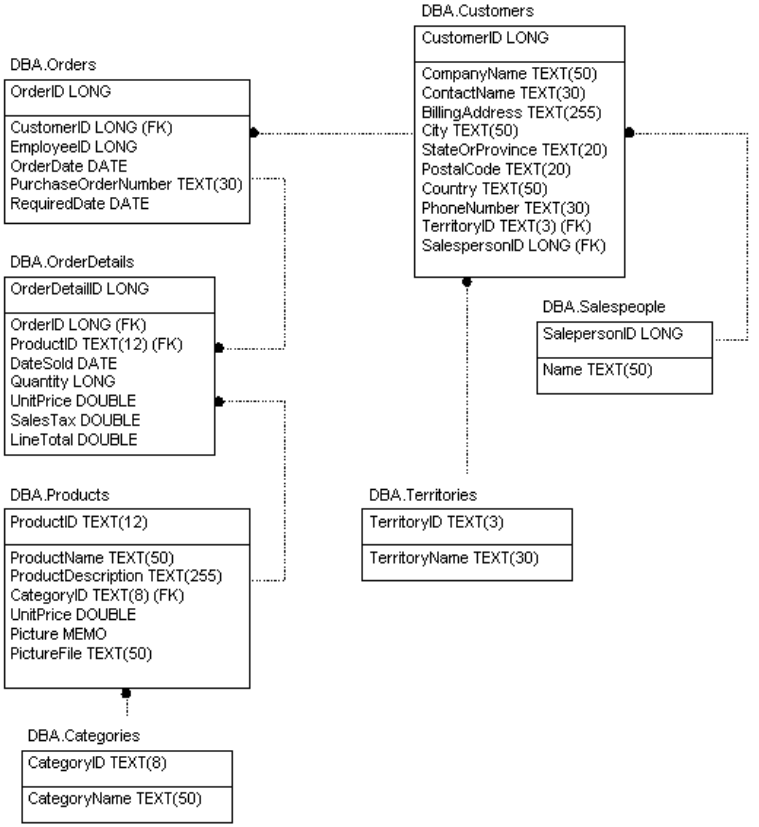


Figure 82 Entity-Relationship diagram for the sample database.

9.2 The DemoData Program

We'll begin with an example of using a class called `DemoData`, used to retrieve data from a database, and then show how a `HiGrid` is used to display selected columns from the database.

The class performs these functions:

1. Establishes the connection to the demo database
2. Constructs the meta data levels and defines the queries that are used to populate the levels
3. Adds some virtual columns to the ones based on database fields
4. Sets the commit policies for all levels

What follows is a line-by-line breakdown of the code. Lines 1–20 are the standard copyright notice that accompanies all JClass examples. They should be assumed as the beginning lines of every other example given in this chapter. Lines 22–29 list the package name and the libraries that `DemoData` imports. This package identifies itself as part of the examples that accompany the product. Package `datasource` forms the data source part of JClass HiGrid's code. As this example shows, it is responsible for setting up the connection to the chosen database and then passing the appropriate SQL query to the database. Actually, the `com.klg.jclass.datasource.jdbc` package is where the code to connect via JDBC resides (or to an ODBC, through a JDBC-ODBC bridge).

Lines 54–57 define the constants that are used to specify which is the desired database connection. Line 59 states that the Microsoft Access database is currently selected.

Line 62 is the beginning of the code for the constructor. It sets up a String for the JDBC–ODBC driver, then embeds the database connection attempt in a `try` block. Line 74 sets up a new `DataTableConnection`. The JDBC URL structure is defined generally as follows:

```
jdbc:<subprotocol>:<subname>
```

In this line, `jdbc` is the standard base, `subprotocol` is the particular data source type, and `subname` is an additional specification that the subprotocol uses. In our example, the subprotocol is `odbc`. The **Driver Manager** uses the subprotocol to match the proper driver to a specific subprotocol. The subname identifies the name of the data source.

Line 74 begins the process of instantiating a new connection. Line 75 declares the driver. In fact, lines 74–79 are a concrete instance of a constructor call whose general form is `com.klg.jclass.datasource.jdbc.DataTableConnection(String driver, String url, String user, String password, String database)`. Parameter `driver` is a String indicating which driver to load, `url` is the URL String described above, `user` is the String for the user's name, `password` is a String for the user's password, if required, and `database` is the String for the database name, which may be null. This class defines various ways of connecting to databases, such as using a host name and port, or an `odbc` style connection, in addition to the one used in our example. Once the connection is established, a query sets up the structure for the data that will be retrieved.

In line 108 of our example, the top-level table of our grid is declared in a query specifying that the database table, `Orders`, is to be used. We wish to include, as sub-tables, information contained in tables *Customers*, *Territories*, *OrderDetails* and *Products–Categories*. The last-mentioned is a detail level consisting of a join of two tables.

Line 108 shows that the `MetaData` class holds the structure of the query. Two constructors are used. First, the "root" constructor is called to set up and execute the query to bootstrap root levels of the `DataModel` and the `MetaDataModel`. This constructor executes the query and sets the resulting `DataTable` as the root of the `DataTableTree`. Call this constructor first, then call the `MetaData(DataModel dataModel, DataTableConnection ds_connection, MetaData parent)` constructor to build the meta data tree hierarchy. Next, the second form of the constructor is called to add master-detail relationships. All of this is accomplished in lines 113–125.

Note that the class' constructor does all the work, and a try block encloses all of the code. If the class can't be instantiated, the exception will print an error message on the monitor.

Once an instance of this class is successfully created, we have established a connection to the named database and the query will return a result set.

Joins are accomplished programmatically by code such as is seen in lines 116 and 124. They may be specified by using Bean customizers if you are using an IDE.

Lines 127 and following show how to attach virtual columns to a grid. These use the `BaseVirtualColumn` class as illustrated in line 151, 156, and 160. The type of aggregation to be done is specified using `BaseVirtualColumn` constants, as shown in lines 154, 158, and 162.

Finally, commit policies for each level are set, beginning at line 188. All three commit policies are illustrated.

```
 1 /**
 2 * Copyright (c) 2002, QUEST SOFTWARE. All Rights Reserved.
 3 * http://www.quest.com
 4 *
 5 * This file is provided for demonstration and educational uses only.
 6 * Permission to use, copy, modify and distribute this file for
 7 * any purpose and without fee is hereby granted, provided that the
 8 * above copyright notice and this permission notice appear in all
 9 * copies, and that the name of Quest not be used in
10 * advertising or publicity pertaining to this material without the
11 * specific, prior written permission of an authorized representative
12 * of Quest.
13 *
14 * QUEST SOFTWARE MAKES NO REPRESENTATIONS OR WARRANTIES ABOUT THE
15 * SUITABILITY OF THE SOFTWARE, EITHER EXPRESS OR IMPLIED, INCLUDING
16 * BUT NOT LIMITED TO THE IMPLIED WARRANTIES OF MERCHANTABILITY,
17 * FITNESS FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. QUEST
18 * SOFTWARE WILL NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY USERS AS A
19 * RESULT OF USING MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS
20 * DERIVATIVES.
21 */
22 package examples.datasource.jdbc;
23
24 import com.klg.jclass.datasource.BaseVirtualColumn;
25 import com.klg.jclass.datasource.MetaDataModel;
26
27 import com.klg.jclass.datasource.TreeData;
28 import com.klg.jclass.datasource.jdbc.DataTableConnection;
29 import com.klg.JClass DataSource.jdbc.MetaData;
30
31 /**
32 * This is an implementation of the JClass DataSource DataModel which
33 * relies on the our own JDBC wrappers (rather than IDE-specific data
34 * binding).
35 *
36 * It models a database for a fictitious bicycle company. The same
37 * schema has been implemented using an MS Access database
```

```
38 * and a SQLAnywhere database (demo.mdb and demo.db respectively).
39 * They contain the same table structures and data.
40 *
41 * The default is to use the jdbc-odbc bridge to connect to the Access
42 * implementation of the data base. You can change which data base is
43 * accessed by changing the dataBase variable to either SA or SYB below.
44 *
45 * This is the tree hierarchy for the data:
46 *   Orders
47 *       Customers
48 *           Territory
49 *       OrderDetails
50 *               Products-Categories
51 *
52 */
53 public class DemoData extends TreeData {
54
55 public static final int MS = 1;
56 public static final int SA = 2;
57 public static final int SYB = 3;
58 //Change the definition of database to any of the above constants.
59 int dataBase = MS;
60 DataTableConnection c;
61
62 public DemoData() {
63     String driver = "sun.jdbc.odbc.JdbcOdbcDriver";
64     if (System.getProperty("java.vendor").indexOf("Microsoft") !=
                                                                  -1) {
65         // use the driver that Microsoft Internet Explorer wants
66         driver = "com.ms.jdbc.odbc.JdbcOdbcDriver";
67     }
68     try {
69         switch (dataBase) {
70         case MS:
71         // This connection uses the jdbc-odbc bridge to
72         // connect to the Access implementation of the
73         // data base.
74         c = new DataTableConnection(
75             driver,                          // driver
76             "jdbc:odbc:JClassDemo",          // url
77             "Admin",                         // user
78             "",                              // password
79             null);                           // database
80         break;
81
82         // This connection uses the jdbc-odbc bridge to connect
83         // to the SQLAnywhere implementation of the data base.
84         case SA:
85         c = new DataTableConnection(
86             "sun.jdbc.odbc.JdbcOdbcDriver",  // driver
87             "jdbc:odbc:JClassDemoSQLAnywhere",// url
88             "dba",                           // user
89             "sql",                           // password
90             null);                           // database
91         break;
92
```

```
93          // This connection uses Sybase's jConnect type 4
94          // driver to connect to the SQLAnywhere implementation
95          // of the data base.
96          case SYB:
97          c = new DataTableConnection(
98              "com.sybase.jdbc.SybDriver",            // driver
99              "jdbc:sybase:Tds:localhost:1498",       // url
100             "dba",                                  // user
101             "sql",                                  // password
102             "HiGridDemoSQLAnywhere");               // database
103         break;
104         default:
105             System.out.println("No database chosen");
106         }
107
108         // Create the Orders MetaData
109         MetaData Orders = new MetaData(this, c,
                " select * from Orders order by OrderID asc");
110         Orders.setDescription("Orders");
111
112         // Create the Customer MetaData
113         MetaData Customers = new MetaData(this, Orders, c);
114         Customers.setDescription("Customers");
115         Customers.setStatement(
                "select * from Customers where CustomerID = ?");
116         Customers.joinOnParentColumn(
                "CustomerID","CustomerID");
117         Customers.open();
118
119         // Create the Territory MetaData
120         MetaData Territory = new MetaData(this, Customers, c);
121         Territory.setDescription("Territory");
122         String t = "select TerritoryID,
                TerritoryName from Territories
                    where TerritoryID = ?";
123         Territory.setStatement(t);
124         Territory.joinOnParentColumn("TerritoryID","TerritoryID");
125         Territory.open();
126
127         // Create the OrderDetails MetaData
128         // Three virtual columns are used:
129         //
130         //      TotalLessTax      (Quantity * UnitPrice)
131         //      SalesTax          (TotalLessTax * TaxRate) and
132         //      LineTotal         (TotalLessTax + SalesTax).
133         //
134         // Thus, when Quantity and/or UnitPrice is changed, these
                derived
135         // values reflect the changes immediately.
136         // Note 1: TaxRate is not a real column either, it is a
137         // constant returned by the sql statement.
138         // Note 2: Virtual columns can themselves be used to derive
                other
139         // virtual columns. They are evaluated from left to right.
140         MetaData OrderDetails = new MetaData(this, Orders, c);
141         OrderDetails.setDescription("OrderDetails");
```

```
142         String detail_query =
                "select OrderDetailID, OrderID, ProductID, ";
143         detail_query += " DateSold, Quantity, UnitPrice, ";
144         detail_query += " '0.15' AS TaxRate ";
145         detail_query += " from OrderDetails where OrderID = ?";
146         OrderDetails.setStatement(detail_query);
147         OrderDetails.joinOnParentColumn("OrderID","OrderID");
148         OrderDetails.open();
149
150         //Extend the row with some calculated values.
151         BaseVirtualColumn TotalLessTax  = new BaseVirtualColumn(
152             "TotalLessTax",
153             java.sql.Types.FLOAT,
154             BaseVirtualColumn.PRODUCT,
155             new String[] {"Quantity", "UnitPrice"});
156         BaseVirtualColumn SalesTax      = new BaseVirtualColumn(
157             "SalesTax",java.sql.Types.FLOAT,
158             BaseVirtualColumn.PRODUCT,
159             new String[] {"TotalLessTax", "TaxRate"});
160         BaseVirtualColumn LineTotal     = new BaseVirtualColumn(
161             "LineTotal",java.sql.Types.FLOAT,
162             BaseVirtualColumn.SUM,
163             new String[] {"TotalLessTax", "SalesTax"});
164
165         OrderDetails.addColumn(TotalLessTax);
166         OrderDetails.addColumn(SalesTax);
167         OrderDetails.addColumn(LineTotal);
168
169         // Create the Products MetaData
170         MetaData Products = new MetaData(this, OrderDetails, c);
171         Products.setDescription("Products");
172         String query = "select a.ProductID,
                    a.ProductDescription,a.ProductName,";
173             query += " a.CategoryID, a.UnitPrice, a.Picture, ";
174             query += " b.CategoryName";
175             query += " from Products a, Categories b";
176             query += " where a.ProductID = ?";
177             query += " and a.CategoryID = b.CategoryID";
178         Products.setStatement(query);
179         Products.joinOnParentColumn("ProductID","ProductID");
180         Products.open();
181
182         // Override the table-column associations for the Products
                table
183         // to exclude the Picture column so it is not included as
                part of
184         // the update. Precision problems cause the server to think
                it's
185         // changed.
186         Products.setColumnTableRelations("Products",
                new String[] {"ProductID",
                    "ProductDescription",
                    "ProductName",
                    "CategoryID",
                    "UnitPrice"});
187
```

```
188         // Override the default commit policy
                COMMIT_LEAVING_ANCESTOR
189         Orders.setCommitPolicy(
                MetaDataModel.COMMIT_LEAVING_RECORD);
190         OrderDetails.setCommitPolicy(
                MetaDataModel.COMMIT_LEAVING_ANCESTOR);
191         Customers.setCommitPolicy(
                MetaDataModel.COMMIT_LEAVING_ANCESTOR);
192         Products.setCommitPolicy(MetaDataModel.COMMIT_MANUALLY);
193         Territory.setCommitPolicy(
                MetaDataModel.COMMIT_LEAVING_ANCESTOR);
194
195     } catch (Exception e) {
196         System.out.println(
                "DemoData failed to initialize " + e.toString());
197     }
198 }
199
200 }
```

9.3 Base Example

The `DemoData` database connection is used for many of the examples that illustrate the use of JClass HiGrid. The first example is called `BaseExample`. It shows a basic grid and sets up a general framework of displaying messages above the grid to describe the particular operation currently being demonstrated in the example. It is worthwhile to describe the structure of the messaging framework, since it is reused in other examples.

`BaseExample` performs the following tasks:

1. Sets up panels in a `JCExitFrame`.
2. Defines methods for setting a title and a prompt.
3. Defines a method called `countChars()` that is used to count the number of new lines in the passed-in prompt. The method is used in the many examples that are subclassed from `BaseExample`.
4. Instantiates a grid and sets the data model with a meta data structure defined by `DemoData`.
5. Uses `HiGridFormatNodeListener` to randomly color all cells.

Messages have a title, a prompt, and a message area that contains text found in the variable standardText. The size of the message area is determined with the help of a method called `countChars`. After defining the Strings for the title, prompt, and standard text, `setPrompt` is called. It uses `countChars` to determine if the text area should be reduced from its maximum size, then displays the text.

Pre-JClass 4.0 technique for traversing the format tree:

Methods `setRandomRowBackgroundColor` set the colors of the different types of rows in the grid, but more generally, illustrate how the grid's `FormatTree` is accessed and used to navigate from the root on down. Note the use of the `TreeIterator` class, which acts as a specialized `Enumeration` type to allow traversal of the format tree.

```
void setRandomRowBackgroundColor(boolean recordFormat) {
FormatTree formatTree = grid.getFormatTree();

    FormatNode node = (FormatNode) formatTree.getRoot();
    setRandomRowBackgroundColor(recordFormat, node);
}

void setRandomRowBackgroundColor(boolean recordFormat, FormatNode node) {
    if (node == null) {
        return;
    }
    Color randomColor = new Color(lightColor(), lightColor(),
        lightColor());
    if (recordFormat) {
        setRowBackgroundColor(node.getRecordFormat(), randomColor);
    }
    else {
        setRowBackgroundColor(node.getFooterFormat(), randomColor);
        setRowBackgroundColor(node.getBeforeDetailsFormat(),
            randomColor);
        setRowBackgroundColor(node.getAfterDetailsFormat(),
            randomColor);
    }
    TreeIterator ti = node.getIterator();
    while (ti.hasMoreElements()) {
        node = (FormatNode)ti.get();
        setRandomRowBackgroundColor(recordFormat, node);
        ti.nextElement();
    }
}
```

The call to `lightColor` returns a random value that is used to specify the RGB color value of the row.

Simpler JClass 6.0 and higher technique:

`HiGrid's` event delegation model lets you catch the creation of each format node as it is about to happen so that you can change the default *plaf* color to one that is randomly chosen.

```
class BaseExampleHiGridFormatNodeListener extends
    HiGridFormatNodeAdapter {
    public void beforeCreateFormatNodeContents(
                HiGridFormatNodeEvent event) {
        CellStyleModel cellStyle = grid.getRecordCellStyle();
        Color randomColor = new Color(lightColor(),
            lightColor(), lightColor());
        cellStyle.setBackground(randomColor);
    }
```

Turning off repainting

Whenever a grid is first loaded, it's a good idea to turn off repainting. Therefore, a common code idiom is:

```
grid.setBatched(true);
   if (applet == null) {
   grid.setDataModel(new jclass.datasource.examples.jdbc.DemoData());
   if (grid.getDataModel() == null) {
       grid.setDataModel(
           new jclass.datasource.examples.vector.VectorData());
   }
   }
   else {
       grid.setDataModel(
           new jclass.datasource.examples.vector.VectorData());
   }
   setRandomRowBackgroundColor(true);
grid.setBatched(false);
```

The operations that could cause repaint flickers are bracketed by `setBatched(true)` ... `setBatched(false)`.

9.4 BaseButton Example

This example merely adds a button to the bottom of the `BaseExample`'s frame. It extends `BaseExample` and its button responds to `actionPerformed` events.

```
public class BaseButtonExample extends BaseExample
implements ActionListener {
```

The button is used in subsequent examples to initiate changes to the grid.

9.5 Cell Validation Example

The interface for doing cell-level validation is illustrated here.

```
public class CellValidationExample extends BaseExample implements
   HiGridValidateListener
```

Because this class implements the `HiGridListener` interface, it must define methods `stateIsInvalid`, `valueChangedBegin`, and `valueChangedEnd`. The class also uses the validation capabilities of `jclass.cell` to fire a `stateIsInvalid` message if the validation criteria are not met. If the change meets the validation criteria, a message on the standard output reports the changed information. If not, the application refuses to change the current row until valid data is placed in the cell being edited.

Here is how `valueChangedBegin` validates the cell's contents:

```
/*
 * HiGridValidateListener implementation
 */
```

```
/**
 * Invoked just before the data source value of the field is updated.
 */
public void valueChangedBegin(HiGridValidateEvent e) {
   JCValidateEvent event = e.getValidateEvent();
   RowNode rowNode = e.getRowNode();
   // find out the user-defined name for this level
   //(assume it is unique)
   String levelName = rowNode.getDataTableModel().
       getMetaData().getDescription();
   String oldValue = getStringValue(event.getOldValue());
   String newValue = getStringValue(event.getValue());
   // reject any changes where the new entry is empty
   if (newValue.length() == 0) {
       event.setValid(false);
       return;
   }
   // only do validation for top-level PurchaseOrderNumber column
   if (levelName.compareTo("Orders") == 0 &&
       e.getColumn().compareTo("PurchaseOrderNumber") == 0) {
       // reject any changes where the first character changes
       boolean valid = (oldValue.length() > 0) &&
           (newValue.length() > 0) &&
           (oldValue.charAt(0) == newValue.charAt(0));
       event.setValid(valid);
   }
}
```

Method `setValid` determines whether `stateIsInvalid` will be fired or not.

9.6 Row Validation Example

There are times when you must simultaneously validate more than a single cell. There may be additional dependencies between the cells in a row that must be checked before committing changes to the database. The current example is based on the premise that an item cannot be delivered before it is ordered.

```
package examples.higrid.validation;

import java.awt.*;
import java.awt.event.*;
import java.util.Date;

import javax.swing.*;
import com.klg.jclass.datasource.*;
import com.klg.jclass.higrid.*;
import com.klg.jclass.util.swing.JCMessageHelper;
import com.klg.jclass.util.swing.JCExitFrame;

import examples.higrid.BaseExample;

/**
 * Do row level validation in HiGrid.
```

```
 */
public class RowValidationExample extends BaseExample {

public RowValidationExample() {
    super();
    grid.getDataModel().addDataModelListener(
            new RowValidateDataModelListener());
    setTitle("Row Validation");
    setPrompt(usage);
}

private Frame myFrame;

public void setFrame(Frame f) {
    myFrame = f;
}

static String usage = "This example shows how to do row level
    validation.\n" +
        "It will reject changes where the RequiredDate comes before
            the OrderDate.";
void validateRow(DataModelEvent event, RowNode rowNode) {
    // find out the user-defined name for this level (assume it is
    //   unique)
    DataTableModel model = rowNode.getDataTableModel();
    String levelName = model.getMetaData().getDescription();
    // only do validation for top-level columns
    if (levelName.compareTo("Orders") == 0) {
        long bookmark = rowNode.getBookmark();
        // ensure that RequiredDate is after OrderDate
        try {
            Date orderDate = (Date)model.getResultData(bookmark,
                 "OrderDate");
            Date requiredDate = (Date)model.getResultData(bookmark,
                 "RequiredDate");
            if (orderDate.after(requiredDate)) {
                event.cancelProposedAction();
                JCMessageHelper.showError("Row Validation",
                    "OrderDate must come before
                         RequiredDate");
            }
        }
        catch (Exception e) {
        }
    }
}

class RowValidateDataModelListener extends DataModelAdapter {
    public void beforeMoveToCurrentRow(DataModelEvent e) {
        validateRow(e, grid.getCurrentRowNode());
    }

    public void beforeCommitRow(DataModelEvent e) {
        validateRow(e, grid.getRowTree().findRecordRowNode(null,
            e.getBookmark()));
    }
```

```
}

public static void main(String args[]) throws InterruptedException {
    JCExitFrame f = new JCExitFrame("Row Validation Example");
     RowValidationExample app = new RowValidationExample();
    app.setFrame(f);
    f.getContentPane().add(app);
    f.pack();
    f.show();
    javax.swing.FocusManager.getCurrentManager().focusNextComponent(
        app.getGrid());
}

}
```

9.7 Exception Message Example

The `examples.higrid.exception.ExceptionMessageExample.java` lists all the HiGrid and DataSource event constants defined in `com.klg.jclass.higrid.HiGridEvent` and `com.klg.jclass.datasource.DataModelEvent`. It presents two ways of handling events. By default, HiGrid presents event and exception messages in a dialog window. You can provide your own mechanism by implementing the HiGrid listener interface and providing code there to deal with the event. Similarly, to deal with `DataModel` events, you implement the `DataModelListener` interface.

To use your own event handling routine, have your class implement the `HiGridListener` interface and register itself as a `HiGridListener`:

```
grid.addHiGridListener(this);
grid.getErrorHandler().setShowErrorDialog(false);
```

Customized event handling is illustrated by displaying the type of event in the text area using the messaging mechanism defined in Section 9.3, Base Example.

9.8 Popup Menu Example

JClass HiGrid's popup menu can be changed, or completely replaced by one of your own design. The example shows that the short and long forms of the popup menu can be selected using the constants `EditPopupMenu.DEFAULT_SHORT_POPUPMENU_LIST` and `EditPopupMenu.DEFAULT_LONG_POPUPMENU_LIST`. It goes on to show how to install additional choices; specifically, an "about" choice that, when clicked, presents a message in the applet's text area.

To view the code, see *examples.higrid.menu.PopupMenuExample.java*.

Part II

Reference Appendices

Appendix A

Bean Properties Reference

The following is a listing of the JClass HiGrid Bean properties and their default values. The properties are arranged alphabetically by property name. The second entry on any given row names the data type returned by the method. Note that a small number of properties are really read-only variables, and therefore only have a *get* method. These properties are marked with a "(G)" following the property name.

A.1 HiGridBean

Property	Type	Default Value
about (G)	java.lang.String	About JClass HiGrid
allowPopupMenu	Boolean	True
allowRowSelection	Boolean	True
allowSorting	Boolean	True
batched	Boolean	False
beepOnInvalid	Boolean	True
connectionsVisible	Boolean	True
editStatusWidth	int	20
editable	Boolean	True
folderIconStyle	int	FOLDER_ICON_STYLE_SHORTCUT
hiGridBeanComponent	jclass.higrid.HiGridBeanComponent	Click to edit.

Property	Type	Default Value
horizontalScrollbar Display	int	DISPLAY_AS_NEEDED
levelIndent	int	25
printFoldersAnd Connections	Boolean	False
printFormat	int	PRINT_AS_DISPLAYED
rowSelectionMode	int	ROW_SELECT_ANY
rowtipVisible	Boolean	True
sortIconsVisible	Boolean	True
version	java.lang.String	JClass HiGrid version number for <platform>
verticalScrollbar Display	int	DISPLAY_AS_NEEDED

A.2 HiGridBeanComponent

Property	Type	Default Value
class	java.lang.Class	class jclass.higrid. HiGridBeanComponent
dataSourceNames	java.lang.String[]	(null)
dataSources	java.lang.Object[]	(null)
resourceName	java.lang.String	(null)
root	jclass.datasource. treemodel.TreeNode	(null)
serializationFile	java.lang.String	jchigrid0.ser
serializationRequired	Boolean	False
structureOnly	Boolean	False

A.3 HiGridBeanCustomizer

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	java.awt.Color	(null)
component	(null)	null
componentCount	int	1
components	java.awt.Component[]	dynamic
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	java.awt.Color	(null)
insets	java.awt.Insets	top=0,left=0,bottom=0,right=0
layout	java.awt.LayoutManager	java.awt.FlowLayout[hgap=5, vgap=5, align=center]
maximumSize	java.awt.Dimension	width=32767,height=32767
minimumSize	java.awt.Dimension	width=144,height=184
name	java.lang.String	panel0
object	java.lang.Object	(null)
preferredSize	java.awt.Dimension	(null)
visible	Boolean	True

A.4 DataBean

Property	Type	Default Value
about	java.lang.String	About JClass DataSource
class	java.lang.Class	class jclass.datasource.DataBean
commitPolicy	int	COMMIT_LEAVING_RECORD
currentGlobalBookmark	long	-1

Property	Type	Default Value
currentGlobalTable	jclass.datasource.DataTableModel	(null)
dataBeanComponent	jclass.datasource.DataBeanComponent	Click to edit.
dataTableTree	jclass.datasource.treemodel.TreeModel	dynamic
description	java.lang.String	Node1
eventsEnabled	Boolean	True
listeners	java.lang.Object	(null)
metaDataTree	jclass.datasource.treemodel.TreeModel	dynamic
modelName	java.lang.String	DataBean1
modified	Boolean	False
version	java.lang.String	JClass HiGrid version number for <platform>

A.5 DataBeanComponent

Property	Type	Default Value
class	java.lang.Class	class jclass.higrid.HiGridBeanComponent
dataSourceNames	java.lang.String[]	(null)
dataSources	java.lang.Object[]	(null)
resourceName	java.lang.String	(null)
root	jclass.datasource.treemodel.TreeNode	(null)
serializationFile	java.lang.String	jchigrid0.ser
serializationRequired	Boolean	False
structureOnly	Boolean	False

A.6 DataBeanCustomizer

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	java.awt.Color	(null)
component	(null)	null
componentCount	int	1
components	java.awt.Component[]	dynamic
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	java.awt.Color	(null)
insets	java.awt.Insets	top=0,left=0,bottom=0,right=0
layout	java.awt.LayoutManager	java.awt.FlowLayout[hgap=5, vgap=5, align=center]
maximumSize	java.awt.Dimension	width=32767, height=32767
minimumSize	java.awt.Dimension	width=90, height=140
name	java.lang.String	panel0
object	java.lang.Object	(null)
preferredSize	java.awt.Dimension	(null)
visible	Boolean	True

A.7 TreeDataBean

Property	Type	Default Value
about (G)	java.lang.String	About JClass DataSource
class	java.lang.Class	class jclass.datasource.TreeDataBean
currentGlobalBookmark	long	-1

Property	Type	Default Value
currentGlobalTable	jclass.datasource.DataTableModel	(null)
dataTableTree	jclass.datasource.treemodel.TreeModel	dynamic
eventsEnabled	Boolean	True
listeners	java.lang.Object	(null)
metaDataTree	jclass.datasource.treemodel.TreeModel	jclass.datasource.treemodel.Tree@1f0bc2
modelName	java.lang.String	TreeData0
modified	Boolean	(null)
treeDataBeanComponent	jclass.datasource.TreeDataBeanComponent	Click to edit.
version (G)	java.lang.String	JClass HiGrid version number for <platform>

A.8 TreeDataBeanComponent

Property	Type	Default Value
class	java.lang.Class	class jclass.datasource.TreeDataBeanComponent
dataSourceNames	java.lang.String[]	(null)
dataSources	java.lang.Object[]	(null)
resourceName	java.lang.String	(null)
root	jclass.datasource.treemodel.TreeNode	(null)
serializationFile	java.lang.String	jcdbtree0.ser
serializationRequired	Boolean	False
structureOnly	Boolean	False

A.9 TreeDataBeanCustomizer

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	java.awt.Color	(null)
component	(null)	null
componentCount	int	1
components	java.awt.Component []	dynamic
enabled	Boolean	True
font	java.awt.Font	(null)
foreground	java.awt.Color	(null)
insets	java.awt.Insets	top=0,left=0,bottom=0,right=0
layout	java.awt.Layout Manager	java.awt.FlowLayout[hgap=5, vgap=5, align=center]
maximumSize	java.awt.Dimension	width=32767,height=32767
minimumSize	java.awt.Dimension	width=105,height=140
name	java.lang.String	panel0
object	java.lang.Object	null
preferredSize	java.awt.Dimension	(null)
visible	Boolean	True

A.10 DSdbJNavigator

Property	Type	Default Value
UIClassID	java.lang.String	not a pluggable look and feel class
accessibleContext	com.sun.java. accessibility. AccessibleContext	dynamic
alignmentX	float	0.5

Property	Type	Default Value
alignmentY	float	0.5
autoscrolls	Boolean	False
background	java.awt.Color	204,204,204
border	com.sun.java.swing.border.Border	null
bounds	java.awt.Rectangle	null
buttonBackground	java.awt.Color	192,192,192
buttonForeground	java.awt.Color	0,0,0
commandVisible	Boolean	True
component	(null)	null
componentCount	int	7
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
deleteVisible	Boolean	True
doubleBuffered	Boolean	True
enabled	Boolean	False
firstVisible	Boolean	True
focusCycleRoot	Boolean	False
focusTraversable	Boolean	False
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null
height	int	0
insertVisible	Boolean	True
insets	java.awt.Insets	0, 0, 0, 0
lastVisible	Boolean	True
layout	java.awt.LayoutManager	null

Property	Type	Default Value
managingFocus	Boolean	False
maximumSize	java.awt.Dimension	32767, 32767
minimumSize	java.awt.Dimension	width=108,height=17
name	java.lang.String	datanavigator0
nextFocusableComponent	java.awt.Component	null
nextVisible	Boolean	True
opaque	Boolean	True
optimizedDrawingEnabled	Boolean	True
paintingTile	Boolean	False
preferredSize	java.awt.Dimension	width=239,height=17
previousVisible	Boolean	True
registeredKeyStrokes	com.sun.java.swing.KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rootPane	com.sun.java.swing.JRootPane	null
statusBackground	java.awt.Color	255,255,255
statusForeground	java.awt.Color	0,0,0
statusVisible	Boolean	True
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	False
visible	Boolean	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	int	0
x	int	0
y	int	0

A.11 DSdbJTextField

Property	Type	Default Value
UI	com.sun.java.swing.plaf.TextUI	dynamic
UIClassID	java.lang.String	TextFieldUI
accessibleContext	com.sun.java.accessibility.AccessibleContext	dynamic
actionCommand	java.lang.String	null
actions	com.sun.java.swing.Action[]	dynamic
alignmentX	float	0.5
alignmentY	float	0.5
autoscrolls	Boolean	True
background	java.awt.Color	dynamic
border	com.sun.java.swing.border.Border	dynamic
bounds	java.awt.Rectangle	null
caret	com.sun.java.swing.text.Caret	dynamic
caretColor	java.awt.Color	0,0,0
caretPosition	int	0
columns	int	0
component	(null)	null
componentCount	int	0
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
disabledTextColor	java.awt.Color	153,153,153
document	com.sun.java.swing.text.Document	dynamic

Property	Type	Default Value
doubleBuffered	Boolean	False
editable	Boolean	True
enabled	Boolean	True
focusAccelerator	char	
focusCycleRoot	Boolean	False
focusTraversable	Boolean	True
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null
height	int	0
highlighter	com.sun.java.swing.text.Highlighter	dynamic
horizontalAlignment	int	2
horizontalVisibility	com.sun.java.swing.BoundedRangeModel	[value=0, extent=0, min=0, max=100, adj=false]
insets	java.awt.Insets	2, 2, 2, 2
keymap	com.sun.java.swing.text.Keymap	dynamic
layout	java.awt.LayoutManager	null
managingFocus	Boolean	False
margin	java.awt.Insets	0, 0, 0, 0
maximumSize	java.awt.Dimension	width=2147483647,height=19
minimumSize	java.awt.Dimension	width=4,height=19
name	java.lang.String	null
nextFocusableComponent	java.awt.Component	null
opaque	Boolean	True
optimizedDrawingEnabled	Boolean	True

Property	Type	Default Value
paintingTile	Boolean	False
preferredScrollable ViewportSize	java.awt.Dimension	width=4,height=19
preferredSize	java.awt.Dimension	width=4,height=19
registeredKeyStrokes	com.sun.java.swing. KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rootPane	com.sun.java.swing. JRootPane	null
scrollOffset	int	0
scrollableTracksViewport Height	Boolean	False
scrollableTracksViewport Width	Boolean	False
selectedText	java.lang.String	null
selectedTextColor	java.awt.Color	0,0,0
selectionColor	java.awt.Color	204,204,255
selectionEnd	int	0
selectionStart	int	0
text	java.lang.String	
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	True
visible	Boolean	True
visibleRect	java.awt.Rectangle	java.awt.Rectangle[x=0,y=0,width=0, height=0]
width	int	0
x	int	0
y	int	0

A.12 DSdbJImage

Property	Type	Default Value
UIClassID	java.lang.String	not a pluggable look and feel class
accessibleContext	com.sun.java.accessibility.AccessibleContext	null
alignmentX	float	0.5
alignmentY	float	0.5
autoscrolls	Boolean	False
background	java.awt.Color	null
border	com.sun.java.swing.border.Border	null
bounds	java.awt.Rectangle	null
component	(null)	null
componentCount	int	0
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
focusCycleRoot	Boolean	False
focusTraversable	Boolean	False
font	java.awt.Font	null
foreground	java.awt.Color	null
graphics	java.awt.Graphics	null
height	int	0
insets	java.awt.Insets	0, 0, 0, 0
layout	java.awt.LayoutManager	null

Property	Type	Default Value
managingFocus	Boolean	False
maximumSize	java.awt.Dimension	width=32767,height=32767
minimumSize	java.awt.Dimension	0, 0
name	java.lang.String	null
nextFocusableComponent	java.awt.Component	null
opaque	Boolean	False
optimizedDrawingEnabled	Boolean	True
paintingTile	Boolean	False
preferredSize	java.awt.Dimension	0, 0
registeredKeyStrokes	com.sun.java.swing.KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rootPane	com.sun.java.swing.JRootPane	null
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	False
visible	Boolean	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	int	0
x	int	0
y	int	0

A.13 DSdbJCheckbox

Property	Type	Default Value
UI	com.sun.java.swing.plaf.ButtonUI	dynamic
UIClassID	java.lang.String	CheckBoxUI

Property	Type	Default Value
accessibleContext	com.sun.java.accessibility.AccessibleContext	dynamic
actionCommand	java.lang.String	
alignmentX	float	0.0
alignmentY	float	0.0
autoscrolls	Boolean	False
background	java.awt.Color	204,204,204
border	com.sun.java.swing.border.Border	dynamic
borderPainted	Boolean	False
bounds	java.awt.Rectangle	null
component	(null)	null
componentCount	int	0
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
disabledIcon	com.sun.java.swing.Icon	null
disabledSelectedIcon	com.sun.java.swing.Icon	null
doubleBuffered	Boolean	False
enabled	Boolean	True
focusCycleRoot	Boolean	False
focusPainted	Boolean	True
focusTraversable	Boolean	True
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null
height	int	0
horizontalAlignment	int	2

Property	Type	Default Value
horizontalTextPosition	int	4
icon	com.sun.java.swing.Icon	null
insets	java.awt.Insets	5, 5, 5, 5
label	java.lang.String	
layout	java.awt.LayoutManager	dynamic
managingFocus	Boolean	False
margin	java.awt.Insets	2, 2, 2, 2
maximumSize	java.awt.Dimension	width=23,height=23
minimumSize	java.awt.Dimension	width=23,height=23
mnemonic	char	null
model	com.sun.java.swing.ButtonModel	dynamic
name	java.lang.String	null
nextFocusableComponent	java.awt.Component	null
opaque	Boolean	False
optimizedDrawingEnabled	Boolean	True
paintingTile	Boolean	False
preferredSize	java.awt.Dimension	width=23,height=23
pressedIcon	com.sun.java.swing.Icon	null
registeredKeyStrokes	com.sun.java.swing.KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rolloverEnabled	Boolean	False
rolloverIcon	com.sun.java.swing.Icon	null
rolloverSelectedIcon	com.sun.java.swing.Icon	null

Property	Type	Default Value
rootPane	com.sun.java.swing.JRootPane	null
selected	Boolean	False
selectedIcon	com.sun.java.swing.Icon	null
selectedObjects	java.lang.Object[]	null
text	java.lang.String	
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	False
verticalAlignment	int	0
verticalTextPosition	int	0
visible	Boolean	True
visibleRect	java.awt.Rectangle	x=0,y=0,width=0,height=0
width	int	0
x	int	0
y	int	0

A.14 DSdbJList

Property	Type	Default Value
UI	com.sun.java.swing.plaf.ListUI	dynamic
UIClassID	java.lang.String	ListUI
accessibleContext	com.sun.java.accessibility.AccessibleContext	dynamic
alignmentX	float	0.5
alignmentY	float	0.5

Property	Type	Default Value
anchorSelectionIndex	int	-1
autoscrolls	Boolean	True
background	java.awt.Color	dynamic
border	com.sun.java.swing.border.Border	null
bounds	java.awt.Rectangle	null
cellRenderer	com.sun.java.swing.ListCellRenderer	dynamic
component	(null)	null
componentCount	int	1
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
firstVisibleIndex	int	-1
fixedCellHeight	int	-1
fixedCellWidth	int	-1
focusCycleRoot	Boolean	False
focusTraversable	Boolean	True
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null
height	int	0
insets	java.awt.Insets	0, 0, 0, 0
lastVisibleIndex	int	-1
layout	java.awt.LayoutManager	null
leadSelectionIndex	int	-1

Property	Type	Default Value
listData	java.util.Vector	null
managingFocus	Boolean	False
maxSelectionIndex	int	-1
maximumSize	java.awt.Dimension	0, 0
minSelectionIndex	int	-1
minimumSize	java.awt.Dimension	0, 0
model	com.sun.java.swing.ListModel	dynamic
name	java.lang.String	null
nextFocusableComponent	java.awt.Component	null
opaque	Boolean	True
optimizedDrawingEnabled	Boolean	True
paintingTile	Boolean	False
preferredScrollableViewportSize	java.awt.Dimension	width=256, height=128
preferredSize	java.awt.Dimension	0, 0
prototypeCellValue	java.lang.Object	null
registeredKeyStrokes	com.sun.java.swing.KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rootPane	com.sun.java.swing.JRootPane	null
scrollableTracksViewportHeight	Boolean	False
scrollableTracksViewportWidth	Boolean	False
selectedIndex	int	-1
selectedIndices	int[]	[I@1f3bf5
selectedValue	java.lang.Object	null

Property	Type	Default Value
selectedValues	java.lang.Object[]	dynamic
selectionBackground	java.awt.Color	204,204,255
selectionEmpty	Boolean	True
selectionForeground	java.awt.Color	0,0,0
selectionInterval	(null)	null
selectionMode	int	2
selectionModel	com.sun.java.swing.ListSelectionModel	dynamic
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	False
valueIsAdjusting	Boolean	False
visible	Boolean	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
visibleRowCount	int	8
width	int	0
x	int	0
y	int	0

A.15 DSdbJTextArea

Property	Type	Default Value
UI	com.sun.java.swing.plaf.TextUI	dynamic
UIClassID	java.lang.String	TextAreaUI
accessibleContext	com.sun.java.accessibility.AccessibleContext	dynamic
actions	com.sun.java.swing.Action[]	dynamic

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
autoscrolls	Boolean	True
background	java.awt.Color	255,255,255
border	com.sun.java.swing.border.Border	dynamic
bounds	java.awt.Rectangle	null
caret	com.sun.java.swing.text.Caret	dynamic
caretColor	java.awt.Color	0,0,0
caretPosition	int	0
columns	int	0
component	(null)	null
componentCount	int	0
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
disabledTextColor	java.awt.Color	153,153,153
document	com.sun.java.swing.text.Document	dynamic
doubleBuffered	Boolean	False
editable	Boolean	True
enabled	Boolean	True
focusAccelerator	char	
focusCycleRoot	Boolean	False
focusTraversable	Boolean	True
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null

Property	Type	Default Value
height	int	0
highlighter	com.sun.java.swing.text.Highlighter	dynamic
insets	java.awt.Insets	2, 2, 2, 2
keymap	com.sun.java.swing.text.Keymap	dynamic
layout	java.awt.LayoutManager	null
lineCount	int	1
lineEndOffset	(null)	null
lineOfOffset	(null)	null
lineStartOffset	(null)	null
lineWrap	Boolean	False
managingFocus	Boolean	True
margin	java.awt.Insets	0, 0, 0, 0
maximumSize	java.awt.Dimension	width=15,height=19
minimumSize	java.awt.Dimension	width=15,height=19
name	java.lang.String	null
nextFocusableComponent	java.awt.Component	null
opaque	Boolean	True
optimizedDrawingEnabled	Boolean	True
paintingTile	Boolean	False
preferredScrollableViewportSize	java.awt.Dimension	width=15,height=19
preferredSize	java.awt.Dimension	width=15,height=19
registeredKeyStrokes	com.sun.java.swing.KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rootPane	com.sun.java.swing.JRootPane	null

Property	Type	Default Value
rows	int	0
scrollableTracksViewport Height	Boolean	False
scrollableTracksViewportWidth	Boolean	False
selectedText	java.lang.String	null
selectedTextColor	java.awt.Color	0,0,0
selectionColor	java.awt.Color	204,204,255
selectionEnd	int	0
selectionStart	int	0
tabSize	int	8
text	java.lang.String	
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	False
visible	Boolean	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	int	0
wrapStyleWord	Boolean	False
x	int	0
y	int	0

A.16 DSdbJLabel

Property	Type	Default Value
UI	com.sun.java.swing.plaf.LabelUI	dynamic
UIClassID	java.lang.String	LabelUI
accessibleContext	com.sun.java.accessibility.AccessibleContext	dynamic

Property	Type	Default Value
alignmentX	float	0.0
alignmentY	float	0.5
autoscrolls	Boolean	False
background	java.awt.Color	204,204,204
border	com.sun.java.swing.border.Border	null
bounds	java.awt.Rectangle	null
component	(null)	null
componentCount	int	0
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	int	0
disabledIcon	com.sun.java.swing.Icon	null
displayedMnemonic	int	0
doubleBuffered	Boolean	False
enabled	Boolean	True
focusCycleRoot	Boolean	False
focusTraversable	Boolean	False
font	java.awt.Font	null
foreground	java.awt.Color	102,102,153
graphics	java.awt.Graphics	null
height	int	0
horizontalAlignment	int	2
horizontalTextPosition	int	4
icon	com.sun.java.swing.Icon	null
iconTextGap	int	4
insets	java.awt.Insets	0, 0, 0, 0

Property	Type	Default Value
labelFor	java.awt.Component	null
layout	java.awt.LayoutManager	null
managingFocus	Boolean	False
maximumSize	java.awt.Dimension	0, 0
minimumSize	java.awt.Dimension	0, 0
name	java.lang.String	null
nextFocusableComponent	java.awt.Component	null
opaque	Boolean	False
optimizedDrawingEnabled	Boolean	True
paintingTile	Boolean	False
preferredSize	java.awt.Dimension	0, 0
registeredKeyStrokes	com.sun.java.swing.KeyStroke[]	dynamic
requestFocusEnabled	Boolean	True
rootPane	com.sun.java.swing.JRootPane	null
text	java.lang.String	
toolTipText	java.lang.String	null
topLevelAncestor	java.awt.Container	null
validateRoot	Boolean	False
verticalAlignment	int	0
verticalTextPosition	int	0
visible	Boolean	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	int	0
x	int	0
y	int	0

Appendix B

Distributing Applets and Applications

B.1 Using JarMaster to Customize the Deployment Archive

The size of the archive and its related download time are important factors to consider when deploying your applet or application.

When you create an applet or an application using third-party classes such as JClass components, your deployment archive will contain many unused class files unless you customize your JAR. Optimally, the deployment JAR should contain only your classes and the third-party classes you actually use. For example, the *jchigrid.jar*, which you used to develop your applet or application, contains classes and packages that are only useful during the development process and that are not referenced by your application. These classes include the Property Editors and BeanInfo classes. JClass JarMaster helps you create a deployment JAR that contains only the class files required to run your application.

JClass JarMaster is a robust utility that allows you to customize and reduce the size of the deployment archive quickly and easily. Using JClass JarMaster you can select the classes you know must belong in your JAR, and JarMaster will automatically search for all of the direct and indirect dependencies (supporting classes).

When you optimize the size of the deployment JAR with JClass JarMaster, you save yourself the time and trouble of building a JAR manually and determining the necessity of each class or package. Your deployment JAR will take less time to load and will use less space on your server as a direct result of excluding all of the classes that are never used by your applet or application.

For more information about using JarMaster to create and edit JARs, please consult its online documentation.

JClass JarMaster is included with JClass DesktopViews. For more details please refer to *Quest Software's Web site*.

Appendix C

Colors and Fonts

This section provides information on common colorname values, specific RGB color values, and fonts applicable to all Java programs. You may find it useful as a guide for choosing colors for cells.

C.1 Colorname Values

The following lists all the colornames that can be used within Java programs. The majority of these colors will appear the same (or similar) across different computing platforms.

- black
- blue
- cyan
- darkGray
- darkGrey
- gray
- grey
- green
- lightGray
- lightGray
- lightBlue
- magenta
- orange
- pink
- red
- white
- yellow

C.2 RGB Color Values

The following lists all the main RGB color values that can be used within JClass HiGrid. RGB color values are specified as three numeric values representing the red, green, and blue color components; these values are separated by dashes ("-").

The following RGB color values describe the colors available to Unix systems. It is recommended that you test these color values in a JClass program on a Windows or Macintosh system before utilizing them.

The list begins with all of the variations of white, then blacks and grays, and then describes the full color spectrum ranging from reds to violets.

Example code from an HTML file:

```
<PARAM NAME=backgroundList VALUE="(4, 5 255-255-0)">
```

RGB Value	Description
255-250-250	Snow
248-248-255	Ghost White
245-245-245	White Smoke
220-220-220	Gainsboro
255-250-240	Floral White
253-245-230	Old Lace
250-240-230	Linen
250-235-215	Antique White
255-239-213	Papaya Whip
255-235-205	Blanched Almond
255-228-196	Bisque
255-218-185	Peach Puff
255-222-173	Navajo White
255-228-181	Moccasin
255 248-220	Cornsilk
255-255-240	Ivory
255-250-205	Lemon Chiffon
255-245-238	Seashell
240-255-240	Honeydew
245-255-250	Mint Cream
240-255-255	Azure
240-248-255	Alice Blue
230-230-250	Lavender
255-240-245	Lavender Blush

RGB Value	Description
255-228-225	Misty Rose
255-255-255	White
0-0-0	Black
47-79-79	Dark Slate Grey
105-105-105	Dim Gray
112- 128-144	Slate Grey
119- 136-153	Light Slate Grey
190- 190-190	Grey
211- 211-211	Light Gray
25-25-112	Midnight Blue
0-0-128	Navy Blue
100- 149 237	Cornflower Blue
72-61-139	Dark Slate Blue
106-90-205	Slate Blue
123- 104 238	Medium Slate Blue
132-112- 255	Light Slate Blue
0-0-205	Medium Blue
65-105-225	Royal Blue
0-0-255	Blue
30-144-255	Dodger Blue
0-19 -255	Deep Sky Blue
135-206-235	Sky Blue
135-206-250	Light Sky Blue
70-130-180	Steel Blue
176-196- 222	Light Steel Blue
173-216-230	Light Blue
176-224-230	Powder Blue
175-238-238	Pale Turquoise
0-206-209	Dark Turquoise
72-209-204	Medium Turquoise
64-224-208	Turquoise
0-255-255	Cyan

RGB Value	Description
224-255-255	Light Cyan
95-158-160	Cadet Blue
102-205-170	Medium Aquamarine
127-255-212	Aquamarine
0-100-0	Dark Green
85-107-47	Dark Olive Green
143-188-143	Dark Sea Green
46-139-87	Sea Green
60-179-113	Medium Sea Green
32-178-170	Light Sea Green
152-251-152	Pale Green
0-255-127	Spring Green
124-252- 0	Lawn Green
0-255-0	Green
127-255- 0	Chartreuse
0-250-154	Medium Spring Green
173-255-47	Green Yellow
50-205-50	Lime Green
154-205-50	Yellow Green
34-139-34	Forest Green
107-142-35	Olive Drab
189-183-107	Dark Khaki
240-230-140	Khaki
238-232-170	Pale Goldenrod
250-250-210	Light Goldenrod Yellow
255-255-224	Light Yellow
255-255-0	Yellow
255-215-0	Gold
238-221-130	Light Goldenrod
218-165-32	Goldenrod
184-134-11	Dark Goldenrod
188-143-143	Rosy Brown

RGB Value	Description
205-92-92	Indian Red
139-69-19	Saddle Brown
160-82-45	Sienna
205-133-63	Peru
222-184- 135	Burlywood
245-245-220	Beige
245-222-179	Wheat
244-164-96	SandyBrown
210-180-140	Tan
210-105-30	Chocolate
178-34-34	Firebrick
165-42-42	Brown
233-150-122	Dark Salmon
250-128-114	Salmon
255-160-122	Light Salmon
255-165- 0	Orange
255-140-0	Dark Orange
255-127-80	Coral
240-128-128	Light Coral
255-99-71	Tomato
255-69-0	Orange Red
255-0-0	Red
255-105-180	Hot Pink
255-20-147	Deep Pink
255-192-203	Pink
255-182-193	Light Pink
219-112-147	Pale Violet Red
176-48-96	Maroon
199-21-133	Medium Violet Red
208-32-144	Violet Red
255-0-255	Magenta
238-130-238	Violet

RGB Value	Description
221-160-221	Plum
218-112-214	Orchid
186-85-211	Medium Orchid
153-50-204	Dark Orchid
148-0-211	Dark Violet
138-43-226	Blue Violet
160- 32-240	Purple
147-112-219	Medium Purple
216-191-216	Thistle

C.3 Fonts

There are five different font names that can be specified in any Java program. They are:

- `Courier`
- `Dialog`
- `DialogInput`
- `Helvetica`
- `TimesRoman`

Note: Font names are case-sensitive.

There are also four standard font style constants that can be used. The valid Java font style constants are:

- `bold`
- `bold+italic`
- `italic`
- `plain`

These values are strung together with dashes ("`-`") when used with the `VALUE` attribute. You must also specify a point size by adding it to other font elements. To display a text using a 12-point italic Helvetica font, use the following:

```
Helvetica-italic-12
```

All three elements (font name, font style and point size) must be used to specify a particular font display; otherwise, the default font is used instead.

Note: Font display may vary from system to system. If a font does not exist on a system, the default font is displayed instead.

Index

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W