

# **JClass DataSource™**

## **Programmer's Guide**

Version 6.3

for Java 2 (JDK 1.3.1 and higher)

***A Robust, Hierarchical, Multiple-platform Data Source***



8001 Irvine Center Drive  
Irvine, CA 92618  
949-754-8000  
[www.quest.com](http://www.quest.com)

**© Copyright Quest Software, Inc. 2004. All rights reserved.**

This guide contains proprietary information, which is protected by copyright. The software described in this guide is furnished under a software license or nondisclosure agreement. This software may be used or copied only in accordance with the terms of the applicable agreement. No part of this guide may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording for any purpose other than the purchaser's personal use without the written permission of Quest Software, Inc.

**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](http://www.quest.com)  
e-mail: [info@quest.com](mailto: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](mailto: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](mailto: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

<b>Preface</b> .....	<b>1</b>
Introducing JClass DataSource . . . . .	1
Assumptions . . . . .	2
Typographical Conventions in this Manual . . . . .	2
Overview of the Manual . . . . .	2
API Reference . . . . .	3
Licensing . . . . .	3
Related Documents . . . . .	3
About Quest . . . . .	4
Contacting Quest Software . . . . .	4
Customer Support . . . . .	5
Product Feedback and Announcements . . . . .	6

## Part I: Using JClass DataSource

<b>1 Using JClass DataSource</b> .....	<b>9</b>
1.1 The Two Ways of Managing Data Binding in JClass DataSource . . . . .	9
1.2 Using JClass DataSource with Visual Components . . . . .	10
1.3 JClass DataSource and the JClass Data Bound Components . . . . .	10
1.4 Internationalization . . . . .	11
<b>2 JClass DataSource Overview</b> .....	<b>13</b>
2.1 Introduction . . . . .	13
2.2 The Data Model's Highlights . . . . .	16
2.3 The Meta Data Model . . . . .	17
2.4 Setting the Data Model . . . . .	24
2.5 JClass DataSource's Main Classes and Interfaces . . . . .	34
2.6 Examples . . . . .	37
2.7 Binding the data to the source via JDBC . . . . .	39
2.8 The Data "Control" Components . . . . .	40
2.9 Custom Implementations . . . . .	41
2.10 Use of Customizers to Specify the Connection to the JDBC . . . . .	43
2.11 Classes and Methods of JClass DataSource . . . . .	43

<b>3</b>	<b>The Data Model</b>	<b>47</b>
3.1	Introduction	47
3.2	Accessing a Database	48
3.3	Specifying Tables and Fields at Each Level	51
3.4	Setting the Commit Policy	53
3.5	Methods for Traversing the Data	54
3.6	The Result Set	56
3.7	Virtual Columns	58
3.8	JClass DataSource Events and Listeners	60
3.9	Handling Data Integrity Violations	67
<b>4</b>	<b>The JClass DataSource Beans</b>	<b>69</b>
4.1	Introduction	69
4.2	Installing JClass DataSource's JAR files	70
4.3	The Data Bean	71
4.4	The Tree Data Bean	82
4.5	The Data Navigator and Data Bound Components	87
4.6	Custom Implementations	88
<b>5</b>	<b>DataSource's Data Bound Components</b>	<b>89</b>
5.1	Introduction	89
5.2	The Types of Data Bound Components	89
5.3	The Navigator and its Functions	92
5.4	Data Binding the Other Components	97
<b>6</b>	<b>Sample Programs</b>	<b>99</b>
6.1	The Sample Database	99
6.2	The DemoData Program	100
6.3	Custom Data Binding	105

## Part II: Reference Appendices

<b>A</b>	<b>Bean Properties Reference</b>	<b>109</b>
A.1	DataBean	109
A.2	DataBeanComponent	110
A.3	DataBeanCustomizer	110
A.4	JCTreeData	111
A.5	TreeDataBeanComponent	112

A.6	TreeDataBeanCustomizer . . . . .	112
A.7	DSdbJNavigator . . . . .	113
A.8	DSdbJTextField . . . . .	115
A.9	DSdbJImage . . . . .	118
A.10	DSdbJCheckbox . . . . .	120
A.11	DSdbJList . . . . .	123
A.12	DSdbJTextArea . . . . .	126
A.13	DSdbJLabel . . . . .	129
<b>B</b>	<b>Distributing Applets and Applications . . . . .</b>	<b>133</b>
B.1	Using JarMaster to Customize the Deployment Archive . . .	133
	<b>Index . . . . .</b>	<b>135</b>





# Preface

[Introducing JClass DataSource](#) ■ [Assumptions](#) ■ [Typographical Conventions in this Manual](#)  
[Overview of the Manual](#) ■ [API Reference](#) ■ [Licensing](#) ■ [Related Documents](#) ■ [About Quest](#)  
[Contacting Quest Software](#) ■ [Customer Support](#) ■ [Product Feedback and Announcements](#)

## Introducing JClass DataSource

JClass DataSource includes a number of data-bound components of its own and is designed to be used with a wide class of GUI components to manage the display of master-detail relational data. JClass DataSource provides enhanced Java components that add data binding to their equivalent AWT and Swing components. JClass DataSource may be used in conjunction with any data-bound component, such as Quest's JClass LiveTable and JClass Field, and it can be tied to a grid or multiple tables to display hierarchical data.

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

### Feature Overview

JClass DataSource is composed of 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. The default behavior 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 DataSource 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.
- Present fields that contain various database types, including pictures.
- 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.

- Use JClass Field components in cells to validate data entry operations.

JClass DataSource 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.

## 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    | <ul style="list-style-type: none"><li>■ Java language source code and examples of file contents.</li><li>■ JClass DataSource and Java classes, objects, methods, properties, constants, and events.</li><li>■ HTML documents, tags, and attributes.</li><li>■ Commands that you enter on the screen.</li></ul>                       |
| <i>Italic Text</i> | <ul style="list-style-type: none"><li>■ Pathnames, filenames, URLs, programs, and method parameters.</li><li>■ New terms as they are introduced, and to emphasize important words.</li><li>■ Figure and table titles.</li><li>■ The names of other documents referenced in this manual, such as <i>Java in a Nutshell</i>.</li></ul> |
| <b>Bold</b>        | <ul style="list-style-type: none"><li>■ Keyboard key names and menu references.</li></ul>  |

## Overview of the Manual

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

Chapter 1, [Using JClass DataSource](#), presents a general overview of JClass DataSource’s general structure and use.

Chapter 2, [JClass DataSource Overview](#), provides additional information on using JClass DataSource.

Chapter 3, [The Data Model](#), describes how a connection to a database is established.

Chapter 4, [The JClass DataSource Beans](#), discusses JClass DataSource’s Bean properties and shows how to use the custom property editor.

Chapter 5, [DataSource’s Data Bound Components](#), presents the suite of data bound components that accompany the product.

Chapter 6, [Sample Programs](#), illustrates some selected techniques for accessing data sources and displaying them in hierarchical grids.

**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 DataSource’s JavaBeans.

Appendix B, [Distributing Applets and Applications](#), illustrates how to use JClass JarMaster to help you combine only those JClass JARs that are required for deploying your application.

## API Reference

The [API](#) reference documentation (Javadoc) is installed automatically when you install JClass DataSource and is found in the `JCLASS_HOME/docs/api/` directory.

## Licensing

In order to use JClass DataSource, 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 DataSource.

## Related Documents

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

- “[Java Platform Documentation](http://java.sun.com/docs/index.html)” at <http://java.sun.com/docs/index.html> and the “[Java Tutorial](http://www.java.sun.com/docs/books/tutorial/index.html)” 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](http://java.sun.com/docs/books/tutorial/uiswing/index.html)” 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 DataSource, 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<sup>sm</sup> 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](http://www.quest.com).

## Contacting Quest Software

<b>E-mail</b>	<a href="mailto:sales@quest.com">sales@quest.com</a>
<b>Address</b>	Quest Software, Inc. World Headquarters 8001 Irvine Center Drive Irvine, CA 92618 USA
<b>Web site</b>	<a href="http://www.quest.com">www.quest.com</a>
<b>Phone</b>	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](http://www.quest.com/support)  
**E-mail**                                [support@quest.com](mailto: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 Desktop Views 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	
<b>JClass Support Email</b>	<a href="mailto:support@quest.com">support@quest.com</a>
<b>Telephone</b>	949-754-8000
<b>Fax</b>	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 DataSource, 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: 949-752-8999

*Part I*

*Using JClass  
DataSource*





# Using JClass DataSource

*The Two Ways of Managing Data Binding in JClass DataSource*

*Using JClass DataSource with Visual Components*

*JClass DataSource and the JClass Data Bound Components* ■ *Internationalization*

## 1.1 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, and other similar components, to a particular database field without having to pay attention to 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 = new MetaDataModel(dm, "select * from orders", c)
table.setDataBinding(dm, mdm, column);
```

Using Binding:

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

## 1.2 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.

## 1.3 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 JClass JCData or JClass JCTreeData and use their customizers to configure their properties, which include connecting to a database and, in the case of JClass 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 JClass JCData or JClass JCTreeData. Custom property editors turn this operation into a sequence of choices – no writing of code is required.

The following chapters discuss the use of JClass DataSource and the data bound components in detail.

## 1.4 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>.



# JClass DataSource Overview

*Introduction ■ The Data Model's Highlights ■ The Meta Data Model ■ Setting the Data Model  
JClass DataSource's Main Classes and Interfaces ■ Examples ■ Binding the data to the source via JDBC  
The Data "Control" Components ■ Custom Implementations  
Use of Customizers to Specify the Connection to the JDBC ■ Classes and Methods of JClass DataSource*

## 2.1 Introduction

JClass DataSource has classes and methods that retrieve, organize, and store data items. 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 have JDBC-ODBC driver support, and even to computed data produced by another application. This 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.

A group of data bound components are included with JClass DataSource. 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. The information on this form might correspond to one of the sub-levels in the grid – you do not have to bind the components to the root level.

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, JClass LiveTable, and JClass HiGrid, thereby multiplying your options for an elegantly designed form.

### 2.1.1 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.

### 2.1.2 `JClass DataSource`'s Organization

`JClass DataSource` is really the “Model” in the Model-View-Controller (MVC) architecture. In fact, it is comprised of two more basic models, a meta data model and a data table model. The classes that make up the meta data model cooperate to define methods for describing the way that you want your data structured. The meta data model describes the hierarchical structure of your design and provides a convenient place for storing all the static information about the actual data tables, such as column names and data types. 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 mirrored cells in the other views to update themselves, thereby maintaining a consistent view of the data everywhere.

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, 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 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.

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 `DataTables`. 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 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 DataSource` 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.

Interface `DataModel` is the data interface for the `JClass DataSource`. An implementation of this interface will be passed to `JClass HiGrid` or its equivalent. 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`.

## 2.2 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.

### 2.2.1 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 platform, the needed database drivers must be installed.

1. On Windows NT/95/98: choose **Start > Settings > Control Panel > ODBC** to launch the ODBC data source administrator. On Windows 2000 and XP: 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.

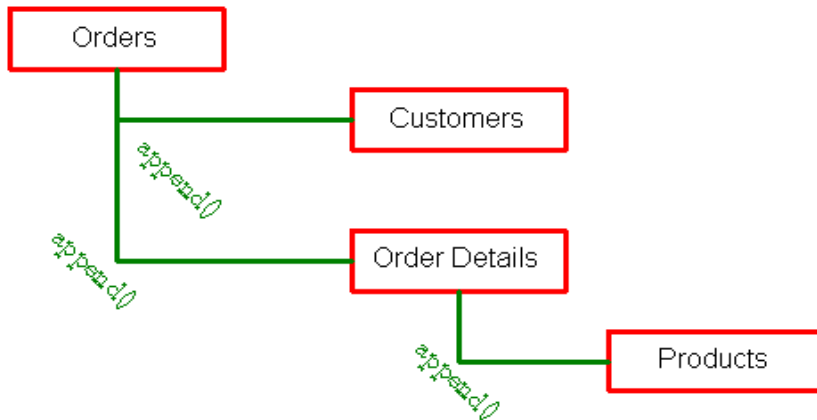
Other environments define different methods for making the low-level connection to a database. Consult the system documentation for your environment for its recommended connection method.



## 2.3 The Meta Data Model

Consider a master-detail design such as that shown in Figure 1. You can create a class that captures this model programmatically or you can describe it using the `JCTreeData`'s customizer in an IDE.

## An example of a Meta-Data Model and Its Meta-Data Tree



### Legend

**Color Coding:**

Line Color	Interface
Red	MetaDataModel
Green	TreeModel
Blue	DataModel
Black	DataTableModel

**Notes:**

**Blue line:**  
There is a "current path" threading through the rows of the data tree. Objects in the path are considered to be "current." The "global cursor" identifies a row in the current path. The only way to change the path is by invoking `DataTable.moveToRow`. The following pages illustrate the other information that can be garnered after setting the path.

→ The "reference" symbol. It indicates that the method returns a reference. For instance, the diagram to the right signifies a reference to the Orders table.

Orders

Products

Dotted lines indicate that this table has not yet been materialized.

Figure 1 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.

### 2.3.1 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.

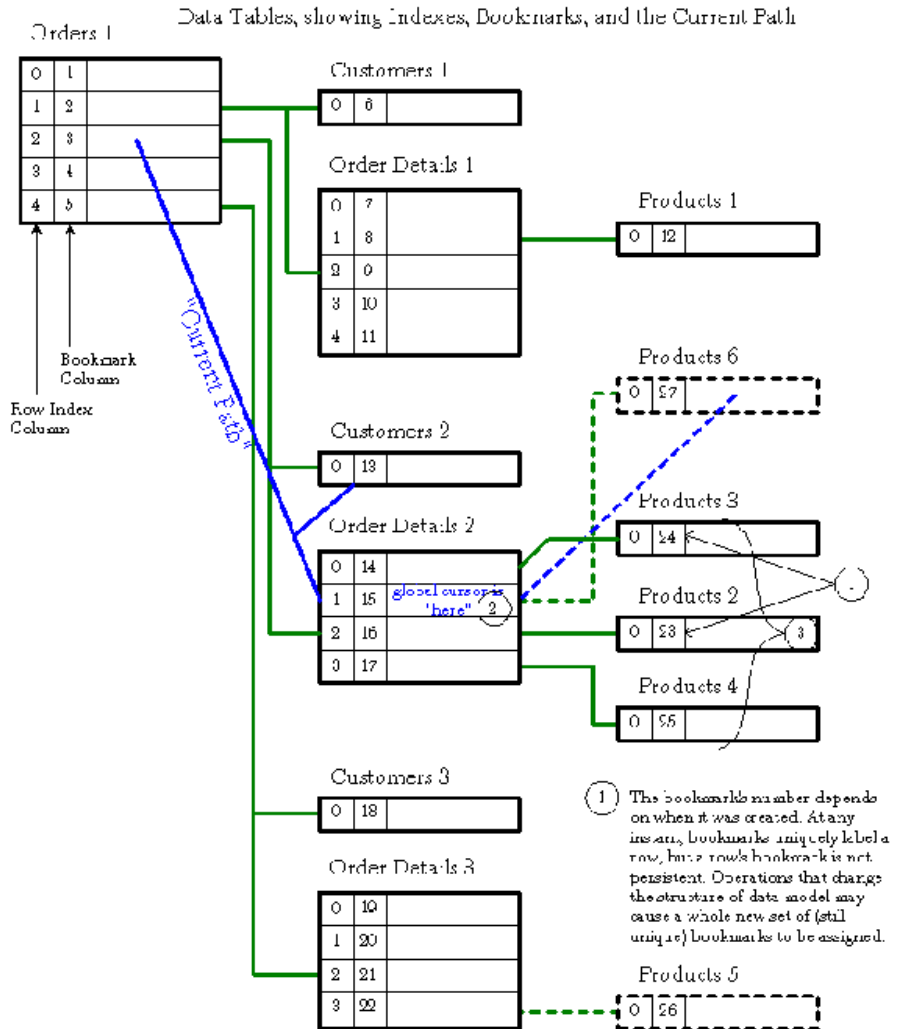
The most important parameter that labels a row is called the *bookmark*. This `long` integer is guaranteed to uniquely label a row at any given time, but it is not guaranteed to be invariant. A row's bookmark most probably will change over time as a result of insert and delete operations on other rows. 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 2.



The data structure for the current path includes the row in the root table where the path begins and continues to the row containing the global cursor. The current path has branches. It includes the first row of any sibling table, as shown in the diagram.

- ② `((TreeNodeModel)at.getTable()).getChildren#` → Vector of references to tables marked "3"
- ③ `((TreeNodeModel)at.getTable()).getParent#` → Orders

*Figure 2 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.

Figure 3 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 1. Some of the methods return references to tables, others return bookmarks and row indexes.

#### EXAMPLES of METHOD USAGE, SHOWING RESULTS

Create children for Bookmark 0:

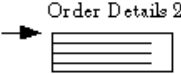
```
Orders.createTable(0, Customers )
Orders.createTable(0, Order Details)
```

Methods for traversing the tree and sample results - assume that the global cursor points to the row whose bookmark is 24:

```
DataTable.getAncestors()=16, 2
DataTable.getParentBookmark()=16
DataTable.getAncestorBookmark()=16, 2
DataTable.getRowIdentifier()=24
DataTable.getRowIndex(24)=0
DataTable.getRows()=1 Products 4
DataTable.getMetaData()=> 
```

Move to a row, then get various data items:

```
DataModel
.moveToRow(OrderDetails 2, 14)
.getCurrentGlobalBookmark()= 14
.getCurrentGlobalTable()=
```



```
.getCurrentDataTable(Customers) =
```



```
.getCurrentDataTable( Products ) =
```

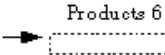


Table "Products 6" will be created because it did not exist before the call.

```
.getCurrentDataTable( Order Details ) =
```



```
.getCurrentDataItem( Order Details, column ) = "The data in this cell"
```

Figure 3 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.

Method `createTable` 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.

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

## 2.4 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 class `VectorData` 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 of 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 columnar 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.setColumnTypes(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.

## 2.4.1 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.

## 2.4.2 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.

## 2.4.3 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.

#### 2.4.4 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.

#### 2.4.5 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`, `JC-TreeData`, and the Swing-based data bound components `DSdbJCompoBox`, and so on, as well as `DSdbJNavigator`, `JClass DataSource`'s data navigator.

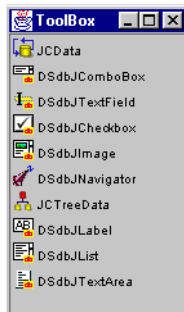


Figure 4 The Bean Development Kit's Toolbox, containing HiGrid's Beans.

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 Bean has a property sheet like the one shown next.

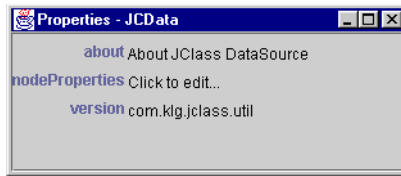


Figure 5 JCData's property sheet.

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.

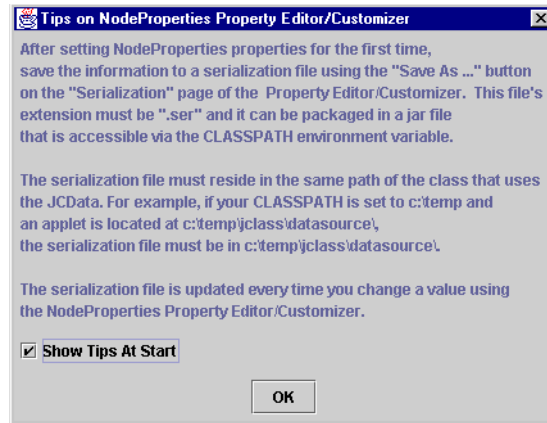


Figure 6 A reminder about creating a serialization file.

4. On the *Node Properties Editor*, click **Open** if you have a previously-saved serialization file that you want to use, otherwise click **Save As**. Type a filename in the 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. Fill in as many of these as are required for your particular situation.

The screenshot shows a Java Swing dialog box titled "com.klg.class.datasource.customizer.NodePropertiesEditor". It has two tabs: "Serialization" and "Data Model". The "Data Model" tab is active and contains three sub-tabs: "JDBC", "Data Access", and "Virtual Columns". The "JDBC" sub-tab is selected and contains several sections:

- Connection**: Includes a "Use Parent Connection" checkbox (unchecked) and an "Auto Commit" checkbox (checked).
- Server**: Contains four fields: "Server Name" (dropdown menu with "jdbc:odbc:JClass Demo SQLAnywhere" selected), "Host or IP Address" (text field), "TCP/IP Port" (text field), and "Driver" (dropdown menu with "sun.jdbc.odbc.JdbcOdbcDriver" selected).
- Login**: Contains three fields: "Login Name" (text field with "dba"), "Password" (text field with "\*\*\*"), and "Database" (text field). Below these is a "Prompt User For Login" checkbox (unchecked).

At the bottom of the dialog, there is a "Connect" button, a "Succeeded" status message, a "Design-time Maximum Number of Rows" field with the value "10", and a "Done" button.

Figure 7 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.

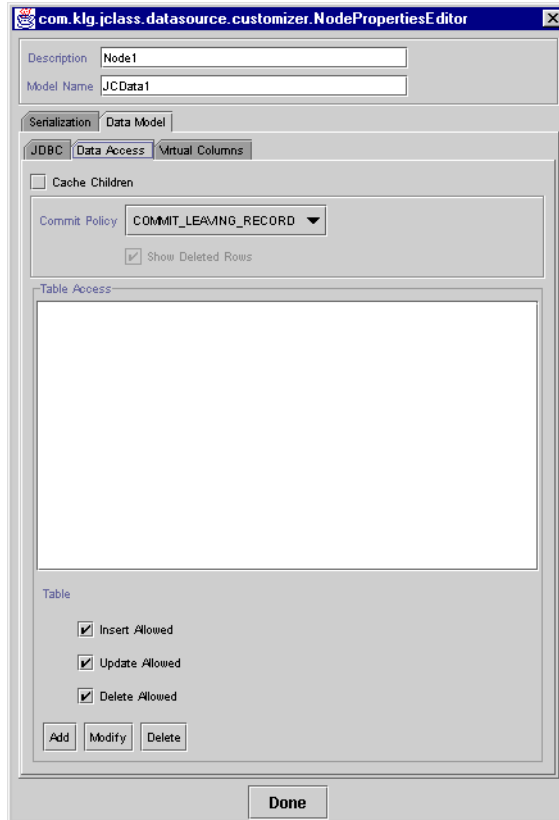


Figure 8 The Data Access tab.

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 also choose one of three commit policies from a drop-down menu: `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, as shown in Figure 9. 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.

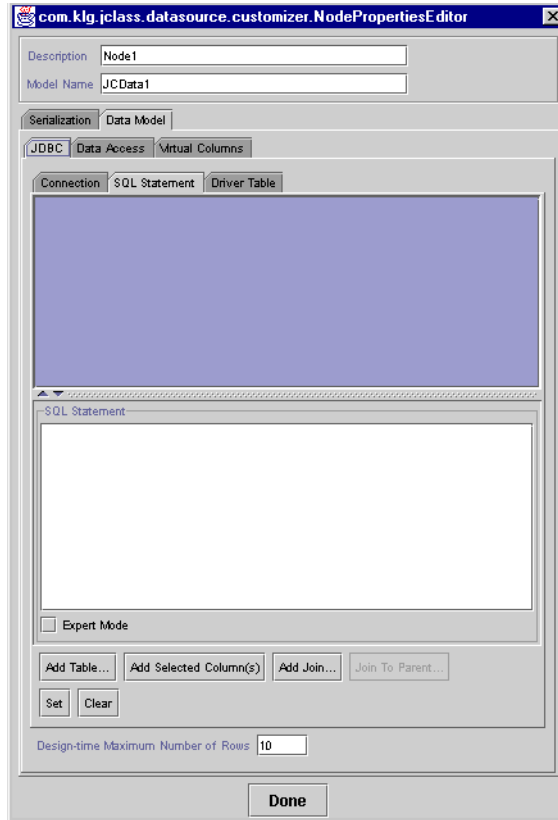


Figure 9 The SQL Statement tab.

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

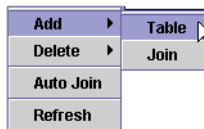


Figure 10 The popup menu for adding tables.

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**.

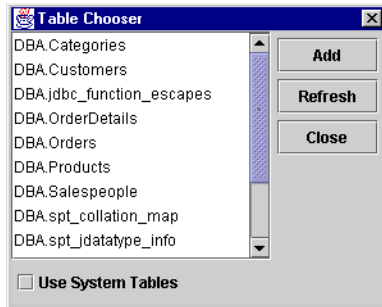


Figure 11 The Table Chooser dialog.

You can choose more than one table if you wish. The result will be a grouping of the two tables, but as of yet no columns or joins have been specified. For this operation to be meaningful, it is likely that you will have to 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 12.

**Important:** 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.

It's also important to 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.

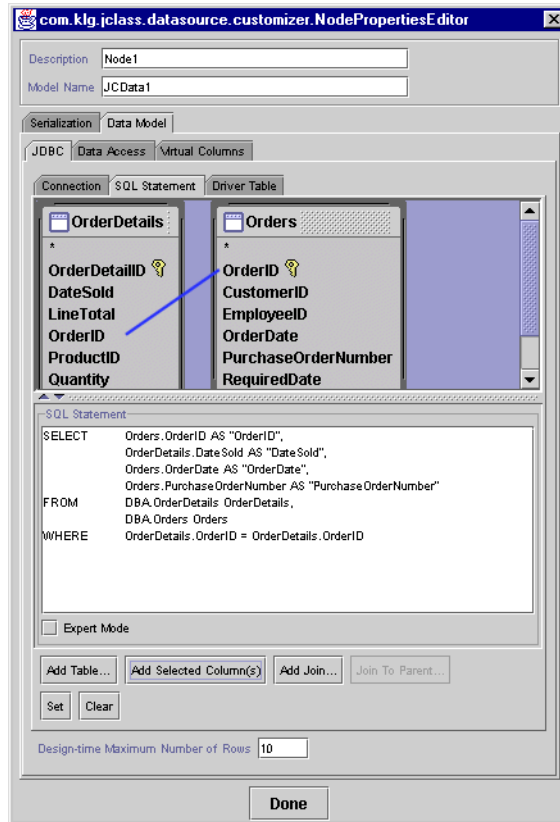


Figure 12 The SQL Statement panel.

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.

10. Select *JCHiGrid* in the *ToolBox* and place an instance on the *BeanBox*.
11. Resize it so that it is big enough to hold most of the cells in five or six rows.
12. 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.

13. 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.
14. Your grid fills with the retrieved data, as shown in Figure 13.

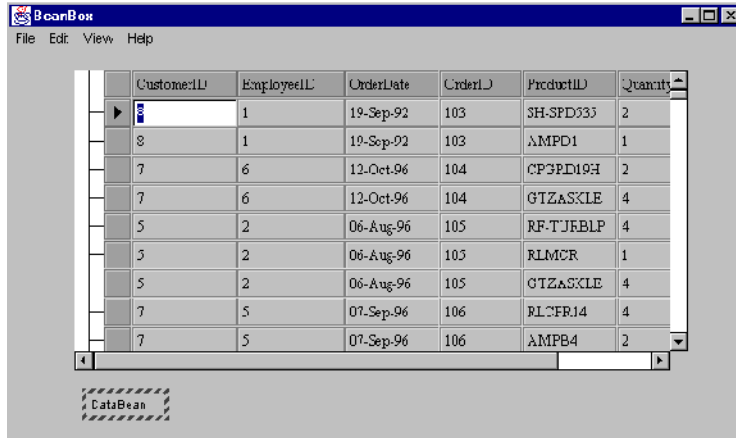


Figure 13 A database table as it appears in the BeanBox.

## 2.5 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.

## JClass DataSource - Structure of the Major Classes and Interfaces

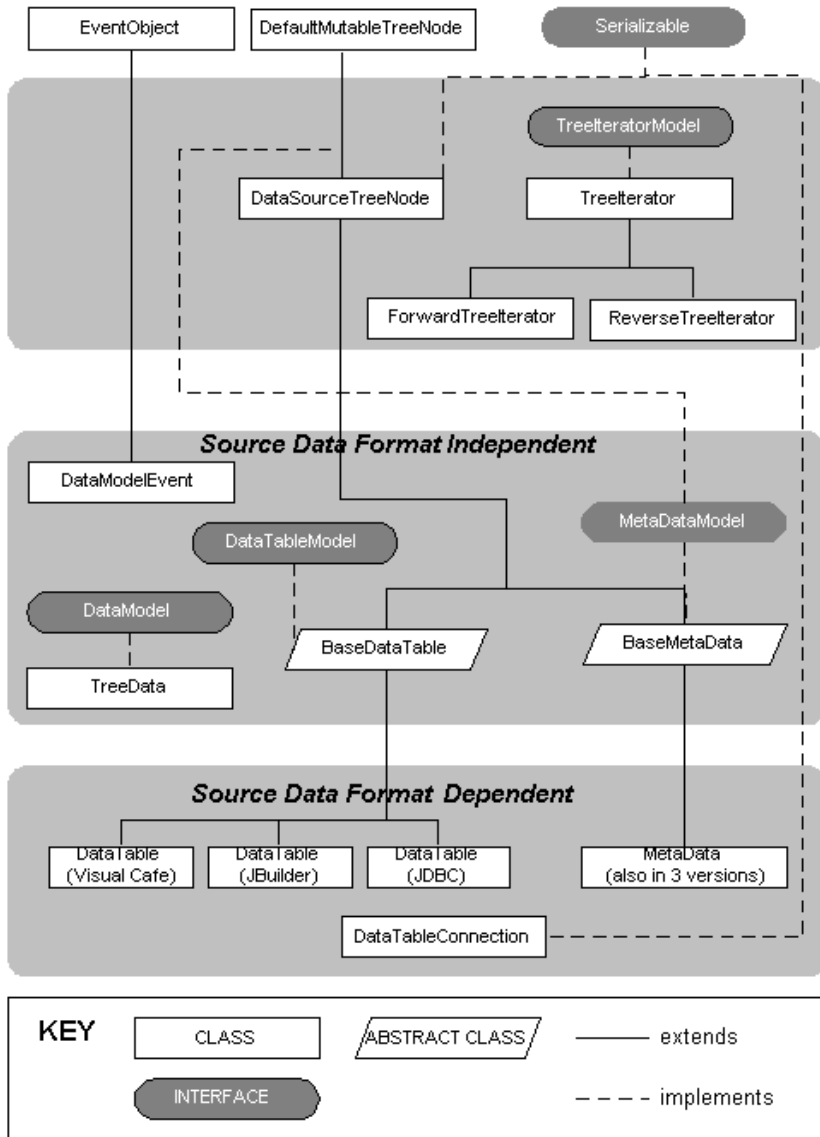


Figure 14 Major classes and interfaces for the Data Model.

### 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`.

## 2.6 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.

### 2.6.1 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.

### 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);
}
```

## 2.7 Binding the data to the source via JDBC

In order to bind the data, you must first connect to a database using the `DataSource` customizer. This is described in [Making a Connection to a Database](#), in Chapter 4.

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.

### 2.7.1 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.

Method `setColumnTableRelations` 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.

## 2.7.2 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.

## 2.8 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 5.



Figure 15 The Data Navigator.

## 2.9 Custom Implementations

### 2.9.1 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.higrd.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));
            }
        }
    }
}
```

## 2.9.2 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.

## 2.10 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.

## 2.11 Classes and Methods of JClass DataSource

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

### 2.11.1 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 `jdbc.class.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\_DECIMAL

---

TYPE\_LONG

---

TYPE\_SQL\_TIME

---

TYPE\_SQL\_TIMESTAMP

---

TYPE\_OBJECT

---

TYPE\_BYTE

---

TYPE\_SHORT

---

TYPE\_BYTE\_ARRAY

---

TYPE\_UTIL\_DATE

---

## 2.11.2 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().

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



# The Data Model

*Introduction* ■ *Accessing a Database* ■ *Specifying Tables and Fields at Each Level*  
*Setting the Commit Policy* ■ *Methods for Traversing the Data* ■ *The Result Set*  
*Virtual Columns* ■ *JClass DataSource Events and Listeners* ■ *Handling Data Integrity Violations*

## 3.1 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 or data bound components.

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

JClass DataSource is structured around two `TreeModels`, 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 `DataTables`. 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 JClass `DataSource`'s 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.

## 3.2 Accessing a Database

Because this product is designed 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.

### 3.2.1 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 [JClass Desktop Views 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.



### 3.2.2 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.

### 3.2.3 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
try {
    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.

### 3.2.4 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 and above.

### 3.2.5 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.

### 3.2.6 Specifying the SQL Query

JClass DataSource'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.

## 3.3 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.

### 3.4 Setting the Commit Policy

There are three commit policies defined in `MetaDataModel`:

Commit Policy	Description
<code>COMMIT_LEAVING_RECORD</code>	Modifications to a row will be written to the originating data source when the cursor moves to any other row.
<code>COMMIT_LEAVING_ANCESTOR</code>	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.
<code>COMMIT_MANUALLY</code>	<p>Any row changes will simply change the status of those rows (see <code>DataTableModel.getRowStatus()</code>). You must then call <code>DataTableModel.commitRow(bookmark)</code> or <code>DataModel.updateAll()</code> to make the changes permanent, or call <code>DataTableModel.cancelRowChanges(bookmark)</code> or <code>DataModel.cancelAll()</code> to undo the changes.</p> <p>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.</p>

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);
```

### 3.5 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
<code>append</code>	Adds a <code>TreeNodeModel</code> to the node upon which the method is invoked. The argument node is added as a child of this node.
<code>getChildren</code>	Returns the <code>Vector</code> that contains the child nodes of the node upon which the method is invoked.
<code>getData</code>	Returns the <code>Object</code> associated with a <code>TreeNodeModel</code> .
<code>getFirstChild</code>	The <code>TreeNode</code> of the first child node for the current data model.
<code>getIterator</code>	Given a starting node, a tree iterator is used to follow the links to the node's descendents.
<code>getLastChild</code>	Follows the link to the last child table for the current <code>TreeNodeModel</code> ; 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.
<code>getNextChild</code>	Follows the link to the next child table for the current <code>TreeNodeModel</code> .
<code>getNextSibling</code>	Follows the link to the next sibling table for the current <code>TreeNodeModel</code> , 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.
<code>getParent</code>	Returns the parent, as a <code>TreeNodeModel</code> ; of the object upon which the method is invoked.
<code>getPreviousChild</code>	Follows the link to the last child table for the current <code>TreeNodeModel</code> , 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.

Method	Description
getPreviousSibling	Follows the link to the last child table for the current <code>TreeNodeModel</code> , 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 <code>TreeNodeModel</code> as a child of the object upon which this method is invoked.
isChildOf( <code>TreeNode</code> )	Use this boolean method to determine if the object upon which the method is invoked is a child of the <code>TreeNodeModel</code> parameter.
remove	Removes the specified <code>TreeNodeModel</code> 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 <code>TreeNode</code> to this node.
getChildren	Returns the <code>Vector</code> 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 <code>TreeNode</code> as a child node of this node.
remove	Removes a child node from the <code>Tree</code> .
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.

## 3.6 The Result Set

### 3.6.1 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`.

Canceling pending updates to the database is accomplished via the cancel methods called `cancelAll (DataModel)` and `cancelAllRowChanges (BaseDataTable)`. See the API for `cancelCellEditing` in `jclass.cell.CellEditor` and its overridden methods in `jclass.cell.editors` for methods which cancel edits to cells.

#### Requerying the Database

`requeryAll` queries 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.

## 3.6.2 Accessing Rows and Columns

Rows and columns may be accessed in various ways, depending on what information is currently available.

Method	Description
<code>BaseMetaData.getColumnCount</code>	The number of columns in the result set.
<code>MetaDataModel.getColumnIdentifier</code>	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.
<code>DataModelEvent.getColumn</code>	Returns a String indicating which column changed, or “ ” in the case where the column is not applicable.
<code>MetaDataModel.getCurrentBookmark</code> <code>DataTableModel.getCurrentBookmark</code>	Moves global cursor to a row, say by <i>first</i> , and return the bookmark.
<code>DataTableModel.getRowIdentifier(i)</code>	The index <i>i</i> is the row order within the result set. The method returns the bookmark for that row.

## 3.6.3 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
<code>getCatalogName</code>	Returns the catalog name for the table containing this field.
<code>getDisplayWidth</code>	Returns the width in pixels of the column
<code>getColumnName</code>	The column's name.
<code>getPrecision</code>	The number of decimal digits.
<code>getSchemaName</code>	The name of the schema for the table containing this column.
<code>getTableName</code>	The name of the table containing this column.
<code>getColumnType</code>	The Java type of the column.

Property	Description
<code>isAutoIncrement</code>	When a new row containing this column is created, its contents are assigned a sequence number. Some databases permit it to be overridden.
<code>isCaseSensitive</code>	Is upper case to be distinguished from lowercase?
<code>isCurrency</code>	Is the data a currency value?
<code>isDefinitelyWritable</code>	Is the field writable?
<code>isNullable</code>	Is null an allowable value?
<code>isReadOnly</code>	Is the column write protected?
<code>isSearchable</code>	Can this column's contents be used in a WHERE clause?
<code>isSigned</code>	Is the object signed?
<code>isWritable</code>	Is the field writable?

### 3.7 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. (You can define your own operation by defining a new constant and subclassing `BaseVirtualColumn`'s `getResultData()` method.) 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)` 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. VirtualColumns 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.

### 3.7.1 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.

## 3.8 JClass DataSource Events and Listeners

The data model fires events that cause a grid, normally `JClass HiGrid`, to redisplay its data based on the changing state of the database. This section describes the methods and constants associated with `JClass DataSource`'s `DataModelEvent` class.

The display grid 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`.

### JClass DataSource - Structure of the Event Classes and Interfaces

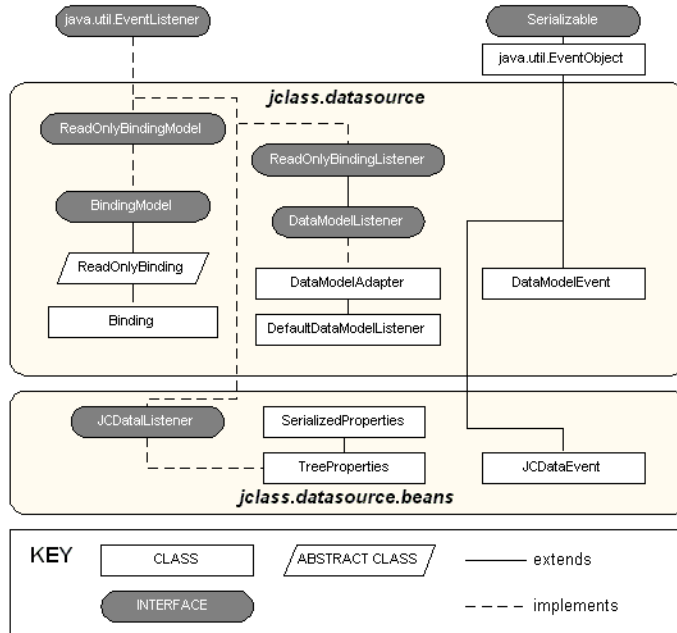


Figure 16 Classes and interfaces related to event handling in JClass DataSource.

#### 3.8.1 The Methods in DataModelEvent

The `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
<code>cancelProposedAction</code>	Cancels the proposed action. This method can be used if the action is cancelable. You may want to test that <code>isCancelable</code> is true before calling <code>cancelProposedAction</code> .
<code>getAncestorBookmarks</code>	Returns a list of the bookmarks which comprise the path from the root to the event node.

Event Method	Description
getBookmark	Returns the bookmark of the changed row.
getCancelled	Sees if proposed action was cancelled by listener.
getColumn	Returns a String indicating which column changed.
getCommand	Gets the command which indicates what action should be taken.
getRowIndex	Returns the row index of the changed row.
getOriginator	Returns the <code>DataModelListener</code> which initiated this event. Allows a listener to determine if it was also the originator of the event.
getTable	Returns the <code>DataTableModel</code> 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. Since 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.

### 3.8.2 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:

<b>DataModel Event Class Constants and Corresponding Listener Method</b>	<b>Description</b>
<p>AFTER_CHANGE_OF_ROW_DATA</p> <p>ReadOnlyBindingListener method: afterChangeOfRowData().</p>	<p>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 required and getRowStatus != INSERTED.</p>
<p>AFTER_DELETE_ROW</p> <p>ReadOnlyBindingListener method: afterDeleteRow().</p>	<p>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 and getRowStatus == DELETED and getShowDeletedRows == true, a row has been required and getRowStatus == INSERTED.</p>
<p>AFTER_INSERT_ROW</p> <p>ReadOnlyBindingListener method: afterInsertRow().</p>	<p>A new row has been added to the datasource. Listeners need to display the row. Rows are always added to the end of DataTableModels.</p>

<b>DataModel Event Class Constants and Corresponding Listener Method</b>	<b>Description</b>
<p>AFTER_MOVE_TO_CURRENT_ROW</p> <p>ReadOnlyBindingListener <b>method:</b> <code>afterMoveToCurrentRow()</code>.</p>	<p>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 <code>DataModel.getCurrentDataTable()</code>, or for field controls, <code>DataModel.getCurrentDataItem()</code>.</p>
<p>AFTER_REQUERY_ROW_AND_DETAILS</p> <p>ReadOnlyBindingListener <b>method:</b> <code>afterRequeryRowAndDetails()</code>.</p>	<p>Re-reads the indicated row and refreshes all open children under this row.</p>
<p>AFTER_REQUERY_TABLE</p> <p>ReadOnlyBindingListener <b>method:</b> <code>afterRequeryTable()</code>.</p>	<p>Re-reads this table and refreshes all open children in the table.</p>
<p>AFTER_RESET</p> <p>ReadOnlyBindingListener <b>method:</b> <code>afterReset()</code></p>	<p>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 <code>DataModel.getDataTableTree().getRoot()</code> for the new root table. Called when the datasource has been reset. See <code>DataModel.requeryAll()</code>.</p>
<p>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</p>	<p>These “BEFORE_” events can be ignored. They are simply to allow applications to cancel the event.</p>
<p>BEFORE_CANCEL_ALL</p> <p>DataModelListener <b>method:</b> <code>beforeCancelAll()</code>.</p>	<p>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 <code>DataModel.cancelAll()</code>.</p>



<b>DataModel Event Class Constants and Corresponding Listener Method</b>	<b>Description</b>
<p>BEFORE_CANCEL_ROW_CHANGES</p> <p>DataModelListener <b>method:</b> beforeCancelRowChanges()</p>	<p>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().</p>
<p>BEFORE_COMMIT_ALL</p> <p>DataModelListener <b>method:</b> beforeCommitAll().</p>	<p>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().</p>
<p>BEFORE_COMMIT_CONDITIONAL</p> <p>DataModelListener <b>method:</b> beforeCommitConditional().</p>	<p>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.</p>
<p>BEFORE_COMMIT_ROW</p> <p>beforeCommitRow().</p>	<p>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 row 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().</p>
<p>BEFORE_DELETE_ROW</p> <p>DataModelListener <b>method:</b> beforeDeleteRow().</p>	<p>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().</p>
<p>BEFORE_DELETE_TABLE</p> <p>DataModelListener <b>method:</b> beforeDeleteTable().</p>	<p>The indicated Data Table will be deleted and flushed from the cache. Can be cancelled.</p>

<b>DataModel Event Class Constants and Corresponding Listener Method</b>	<b>Description</b>
BEFORE_EDIT_CELL  DataModelListener <b>method:</b> beforeEditCell().	Event fired before a cell is edited. Can be cancelled. See DataTableModel.updateCell().
BEFORE_INSERT_ROW  DataModelListener <b>method:</b> 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 <b>method:</b> beforeMoveToCurrentRow().	The global cursor will move to a new row. Can be cancelled.
BEFORE_REQUERY  DataModelListener <b>method:</b> beforeRequery().	Event fired when either DataTableModel.requeryRowAndDetails() or DataTableModel.requeryRow() is called. If not cancelled this event will be followed by an AFTER_REQUERY_ROW_AND_DETAILS event, or 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.requeryRowAndDetails().
BEFORE_RESET  DataModelListener <b>method:</b> 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 <b>method:</b> 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.

<b>DataModel Event Class Constants and Corresponding Listener Method</b>	<b>Description</b>
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.

## 3.9 Handling Data Integrity Violations

### 3.9.1 Exceptions

Many files, including the `JCData`, `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);
}
```

# The JClass DataSource Beans

*Introduction* ■ *Installing JClass DataSource's JAR files*

*The Data Bean* ■ *The Tree Data Bean*

*The Data Navigator and Data Bound Components* ■ *Custom Implementations*

## 4.1 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. Please see [Appendix A](#) for a list of properties for the JClass DataSource JavaBeans.

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<sup>1</sup>, 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.

---

1. Swing components are designated DSdbJCheckbox, and so on.

This chapter describes `JCData` and `JCTreeData`. The navigator and data bound components are described in the following chapter.

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
```

## 4.2 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 `/jdk/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.

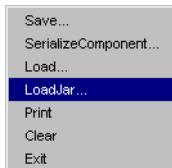


Figure 17 Choosing `LoadJar...` from the `BeanBox`'s File menu.

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.

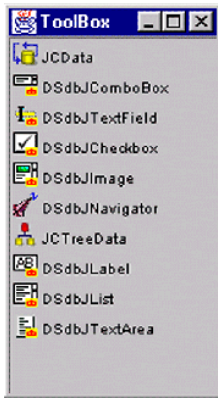


Figure 18 JClass DataSource's Beans, displayed in the ToolBox.

At this point you are ready to add these components to the BeanBox and begin setting them up.

## 4.3 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 one non-expandable grid. The fields in the grid may be chosen from more than one database table.

### 4.3.1 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 `jdbcdatasource.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.

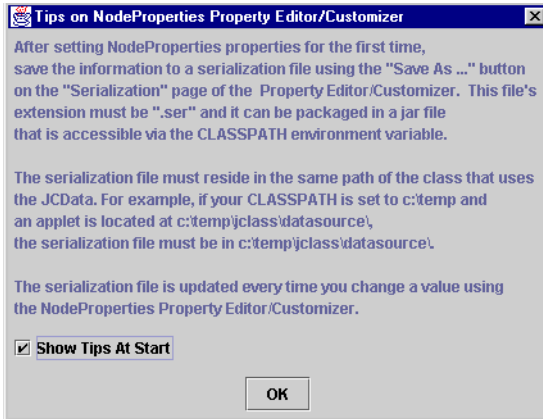


Figure 19 The Data Bean tip window: a reminder to save the serialization file.

For more information on saving the serialization file, see Section 4.3.3, [Saving a Serialization File](#).

### 4.3.2 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 20.

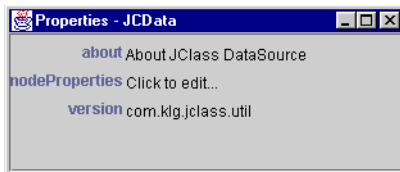


Figure 20 The BeanBox's Properties sheet, showing the properties of the Data Bean.

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.



### 4.3.3 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.

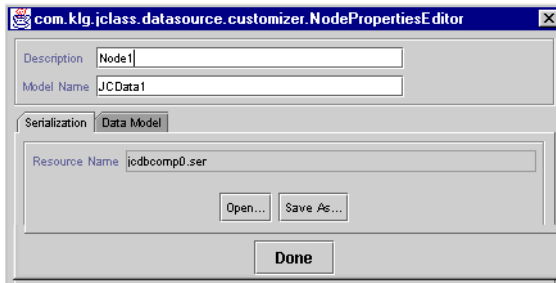


Figure 21 The *Serialization* tab of the *NodePropertiesEditor*.

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.

### 4.3.4 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 22 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.)

The screenshot shows a dialog box with three tabs: 'Connection', 'SQL Statement', and 'Driver Table'. The 'Connection' tab is active. It features a 'Use Parent Connection' checkbox (unchecked) and an 'Auto Commit' checkbox (checked). Below this is a 'Server' section with four fields: 'Server Name' (a dropdown menu showing 'o:ff-sybase://gonzo:5003;dbx:ff-sybase://gonzo:5003'), 'Host or IP Address' (a text box containing 'jdbc:ff-sybase://gonzo'), 'TCP/IP Port' (a text box containing '5003'), and 'Driver' (a dropdown menu showing 'ocnnet.sybase.Sybase Driver'). A 'Login' section follows, containing 'Login Name' (text box with 'myLogin'), 'Password' (text box with '\*\*\*\*\*'), 'Database' (text box with 'r\_and\_d'), and a 'Prompt User For Login' checkbox (checked).

Figure 23 A database connection that requires every field to be filled in.

### 4.3.5 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 25.

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.

### 4.3.6 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 25 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.

### 4.3.7 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 24 The *Join* window.

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.

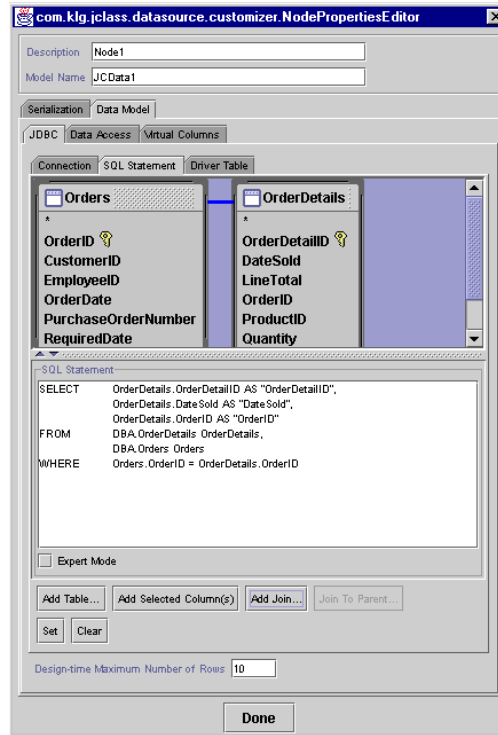


Figure 25 The SQL Statement page, showing a completed query.

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.

### 4.3.8 The Driver Table Tab

One of the possible operations on data tables, once they have been retrieved from the database, is the query 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 query. This is best illustrated with an example.

For the query 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.SalepersonID = Salespeople.SalepersonID
```

For the query 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 *SalepersonID*, 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 query 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 query a single row.

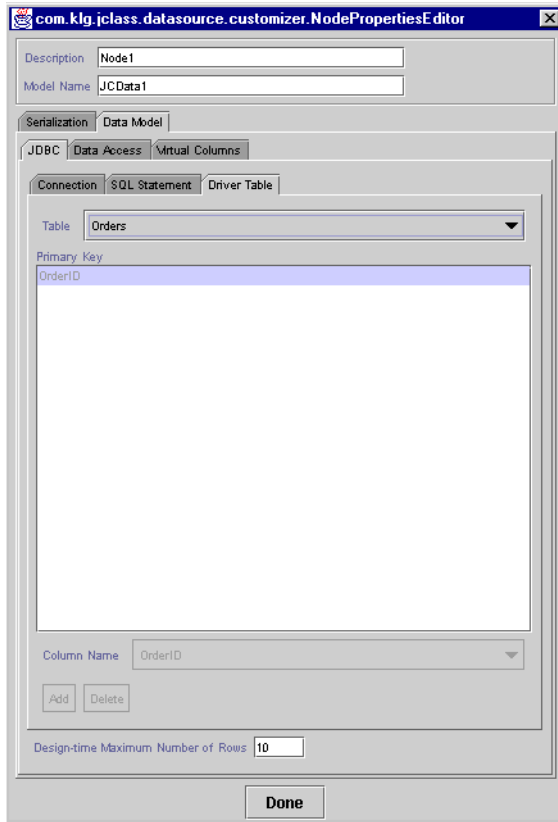


Figure 26 The Driver Table tab.

### 4.3.9 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 policies in [Commit Policy](#), in Chapter 2, 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**.

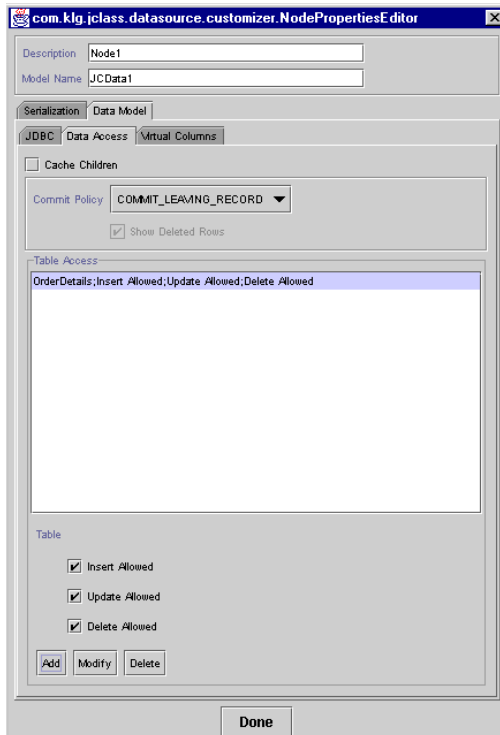


Figure 27 The Data Access tab.

#### 4.3.10 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*.

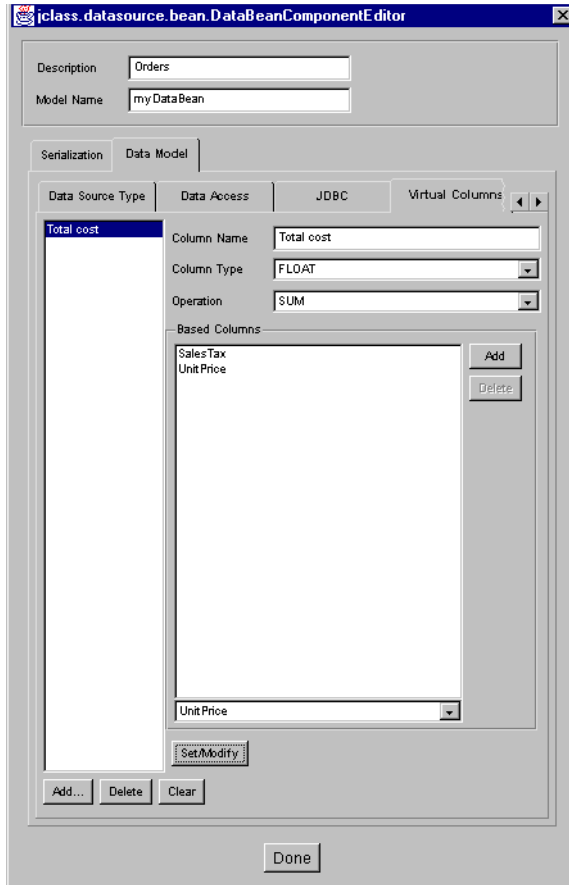
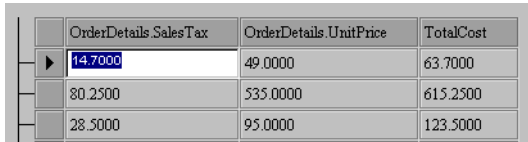


Figure 28 The Virtual Columns tab.

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 28 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 29 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.

### 4.3.11 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.

## 4.4 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 *TreePropertiesEditor* 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 30.

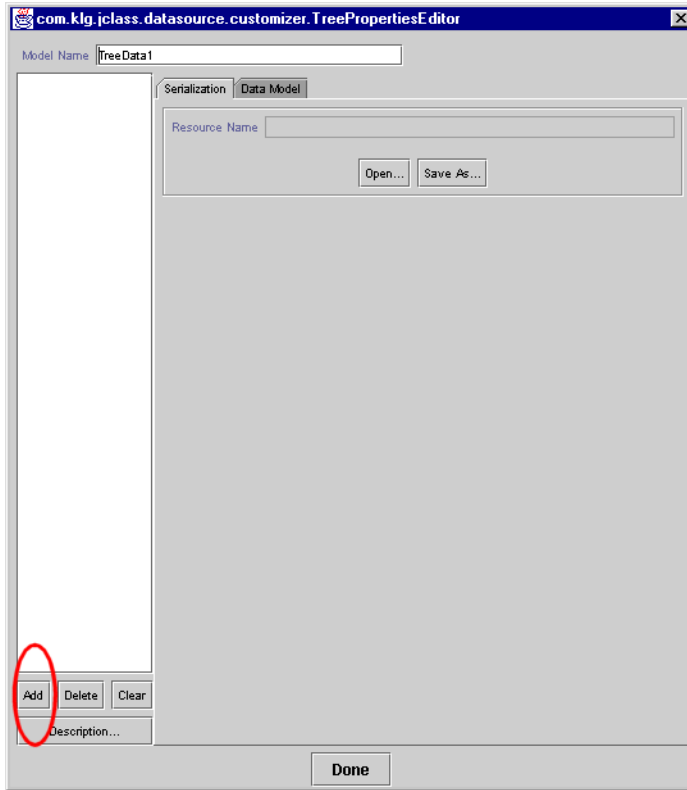


Figure 30 Before a table is added, you are asked to supply a descriptive name.

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 31 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.

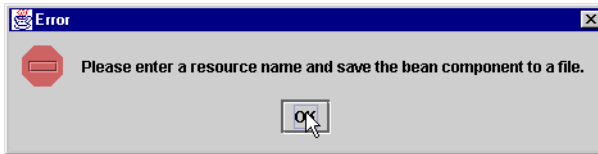


Figure 31 Name the root data table after clicking the Add button in Figure 30.

You have the beginnings of your data design, as shown in Figure 32.

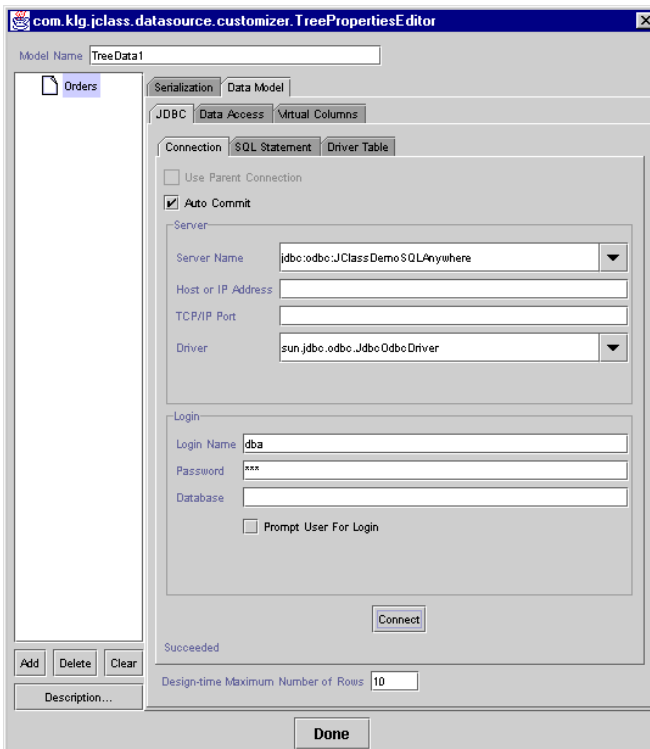


Figure 32 The Connection tab for the Tree Data Bean component.

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 4.3.8, [The Driver Table Tab](#), for the step-by-step process.

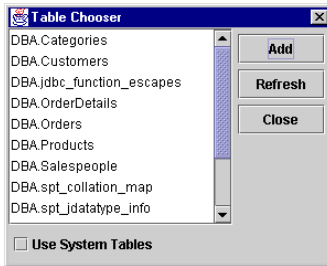


Figure 33 The Table Chooser window.

The completed form is shown in Figure 35. 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.

#### 4.4.1 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.

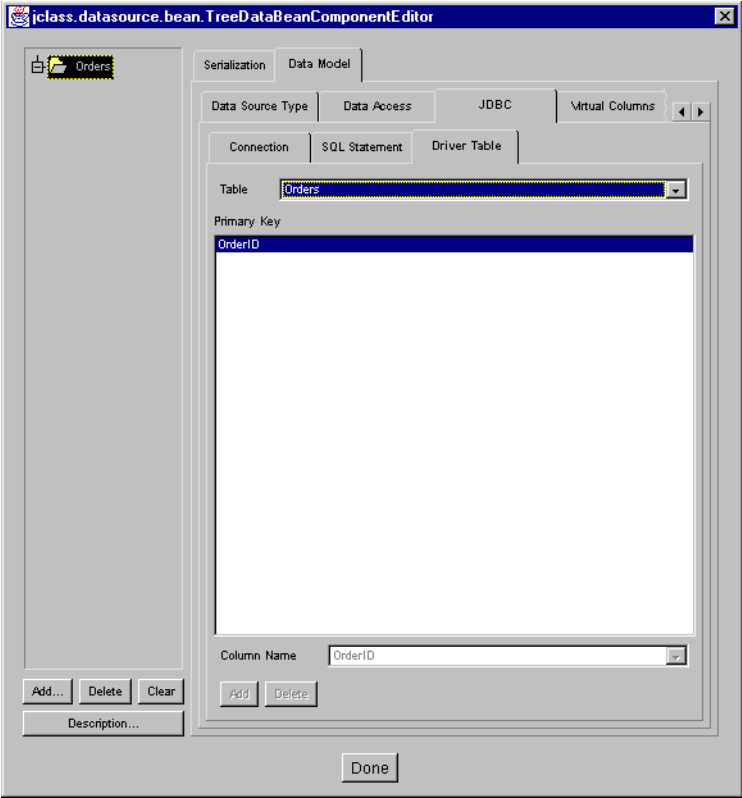


Figure 34 The Driver Table tab.

The next diagram shows a completed *SQL Statement* panel for a sub-table called *OrderDetails*.

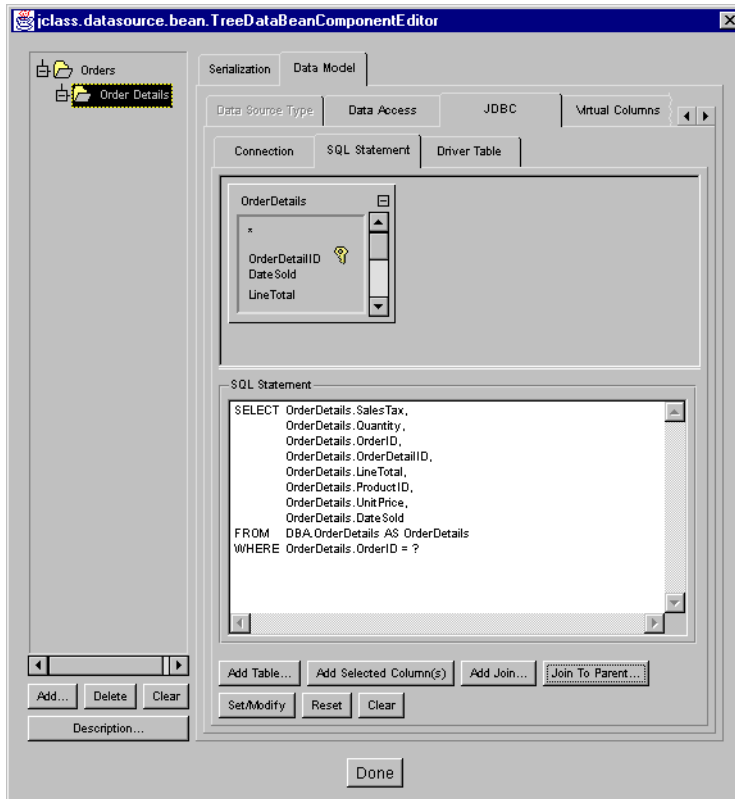


Figure 35 Adding a detail-level table and selecting a query statement.

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.

## 4.5 The Data Navigator and Data Bound Components

There are several JClass DataSource 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.

[DataSource's Data Bound Components](#), in Chapter 5 discusses in detail the navigator and the data bound controls.

## 4.6 Custom Implementations

### 4.6.1 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 section demonstrates the use of such a customizer. It will demonstrate how to add the JAR file to a specific IDE so that you can begin using the JClass DataSource's JavaBeans.

### 4.6.2 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.



# DataSource's Data Bound Components

*Introduction* ■ *The Types of Data Bound Components*  
*The Navigator and its Functions* ■ *Data Binding the Other Components*

## 5.1 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.

## 5.2 The Types of Data Bound Components

JClass DataSource contains Swing-type components. If you are using other JClass DesktopViews products, 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 class, 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 section illustrates how this is done.

JClass DataSource has two capabilities: a data model and data binding. The data model is an API for the management of hierarchical data. Data binding is built on top of the data model and presents convenience classes used to bind JClass Chart, JClass LiveTable, and JClass Field to the data model. JClass HiGrid connects directly to the data model, so it has no need to concern itself with the additional data binding mechanism.

Single-level data binding is possible in JClass DataSource with the help of Binding classes. You can create a data source, connect a field or table to it and add listeners to a single level. The components themselves do not need to know about any level other than the one they are interested in. This means they only receive events which directly affect the data in their level, or cascading events which affect their level. This simplifies the way that data-display components are programmed.

To hide the hierarchical underpinnings of the DataSource, there is now a class called `com/klg/jclass/datasource/Binding`. This class, along with those derived from it like `com/klg/jclass/datasource/jdbc/JDBCBinding`, creates the `DataModel` and `MetaDataModel` objects for the user. Here is an example that creates two levels in JDBC:

```
JDBCBinding orders = new JDBCBinding(c, "select * from Orders");
// pass parent, orders, to new child to establish the hierarchy
JDBCBinding details = new JDBCBinding(c, "select * from OrderDetails",
    orders);
```

You would then create the component by either passing in the instance of Binding, or explicitly setting it as in these examples which bind a TextField component:

```
DSdbJTextField orderId = new DSdbJTextField(orders, "OrderID");
```

or

```
DSdbJTextField orderId = new DSdbJTextField();
orderId.setDataBinding(orders, "OrderID");
```

### 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.

## 5.3 The Navigator and its Functions

### 5.3.1 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 `jclass.datasource.swing.DSdbJNavigator`.

### 5.3.2 The Navigator Binds to any MetaData Level

The navigator binds to any `MetaData` level, just like the other `DataSource` data bound components. It uses the `jclass.datasource.bean.DataBindingEditor` property editor. The property is called `DataBinding`, just like all the other data bound components in `JClass LiveTable`, `JClass Field`, `JClass Chart`, 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")`.

### 5.3.3 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 36 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 <code>DataTable</code> .
Previous	Moves to the previous row in the current <code>DataTable</code> . 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 <code>JClass HiGrid</code> 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 <code>Data Table</code> record number, and total number of records in that <code>Data Table</code> .
Next	Moves to the next row in the current <code>DataTable</code> . If already at the end, no move occurs
Last	Moves to the last row in the current <code>DataTable</code>
Insert	Adds a new record in the current table at the end of the table.

The Swing version of the navigator uses tooltips to show what each of the buttons does. The tooltip'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 37 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.
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.

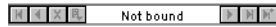
### 5.3.4 Exploring DSdbNavigator's Bean Properties

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.

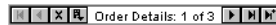
Here are the steps to bind a navigator to a data source:

1. Place a `JCData` or a `JCTreeData` on your form.

- Use the Data Bean's customizer to specify the connection to the database.
- 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.



- Click **Select a Data Source** to launch its data binding editor.
- 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.
- 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 38.

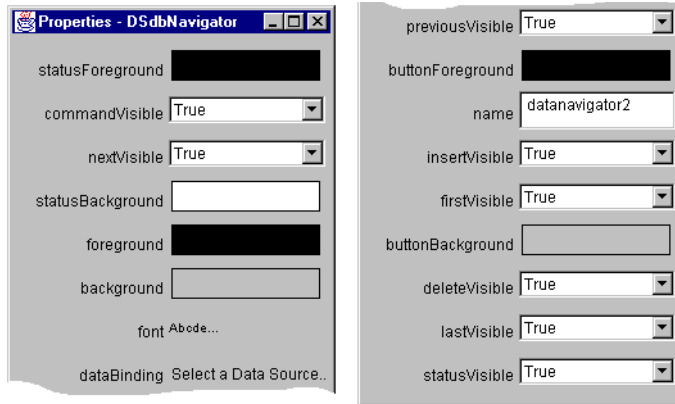


Figure 38 The properties of `DSdbNavigator`.

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**.

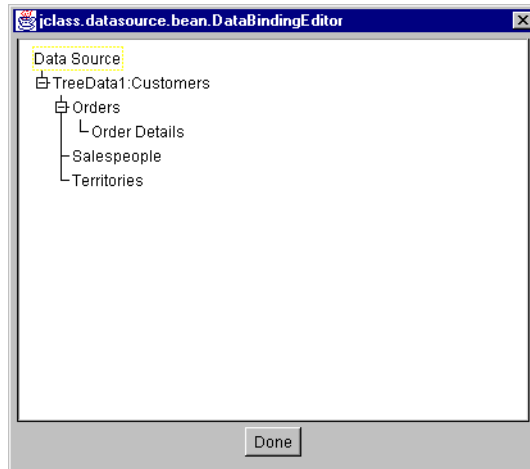


Figure 39 DSdbNavigator's data binding editor.

## 5.4 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.

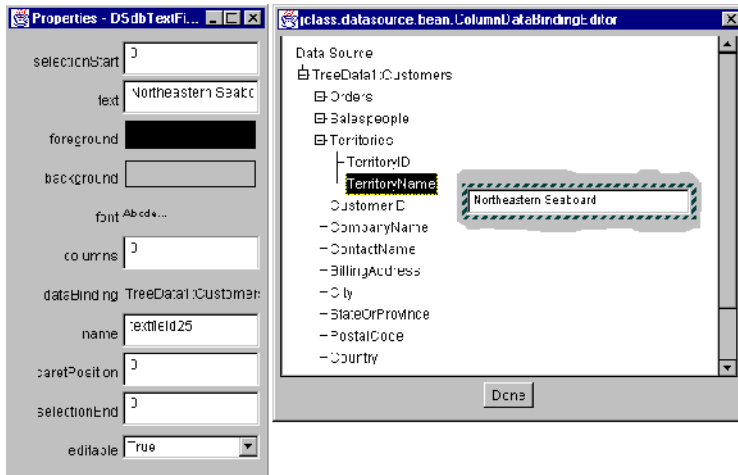


Figure 40 A DSdbTextField, its Properties, and its ColumnDataBindingEditor.



The entity-relationship 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.

## 6.2 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. 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. The `datasource` package 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 `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 `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 Software not be used in
10 * advertising or publicity pertaining to this material without,
11 * the specific prior written permission of an authorized
12 * representative of Quest Software.
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, FITNESS
17 * FOR A PARTICULAR PURPOSE, OR NON-INFRINGEMENT. QUEST SOFTWARE SHALL
18 * NOT BE LIABLE FOR ANY DAMAGES SUFFERED BY USERS AS A RESULT OF USING,
19 * MODIFYING OR DISTRIBUTING THIS SOFTWARE OR ITS DERIVATIVES.
20 */
21
```

```

22 package jclass.datasource.examples.jdbc;
23
24 import java.util.*;
25 import java.sql.*;
26
27 import jclass.datasource.treemodel.*;
28 import jclass.datasource.jdbc.*;
29 import jclass.datasource.*;
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,
110         "select * from Orders order by OrderID asc");
111     Orders.setDescription("Orders");
112
113     // Create the Customer MetaData
114     MetaData Customers = new MetaData(this, Orders, c);
115     Customers.setDescription("Customers");
116     Customers.setStatement(
117         "select * from Customers where CustomerID = ?");
118     Customers.joinOnParentColumn(
119         "CustomerID", "CustomerID");
120     Customers.open();
121
122     // Create the Territory MetaData
123     MetaData Territory = new MetaData(this, Customers, c);
124     Territory.setDescription("Territory");
125     String t = "select TerritoryID,
126         TerritoryName from Territories
127         where TerritoryID = ?";
128     Territory.setStatement(t);
129     Territory.joinOnParentColumn("TerritoryID", "TerritoryID");
130     Territory.open();
131
132     // Create the OrderDetails MetaData
133     // 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 =
143         "select OrderDetailID, OrderID, ProductID, ";
144     detail_query += " DateSold, Quantity, UnitPrice, ";
145     detail_query += " '0.15' AS TaxRate ";
146     detail_query += " from OrderDetails where OrderID = ?";
147     OrderDetails.setStatement(detail_query);
148     OrderDetails.joinOnParentColumn("OrderID","OrderID");
149     OrderDetails.open();
150
151     //Extend the row with some calculated values.
152     BaseVirtualColumn TotalLessTax = new BaseVirtualColumn(
153         "TotalLessTax",
154         java.sql.Types.FLOAT,
155         BaseVirtualColumn.PRODUCT,
156         new String[] {"Quantity", "UnitPrice"});
157     BaseVirtualColumn SalesTax = new BaseVirtualColumn(
158         "SalesTax",java.sql.Types.FLOAT,
159         BaseVirtualColumn.PRODUCT,
160         new String[] {"TotalLessTax", "TaxRate"});
161     BaseVirtualColumn LineTotal = new BaseVirtualColumn(
162         "LineTotal",java.sql.Types.FLOAT,
163         BaseVirtualColumn.SUM,
164         new String[] {"TotalLessTax", "SalesTax"});
165
166     OrderDetails.addColumn(TotalLessTax);
167     OrderDetails.addColumn(SalesTax);
168     OrderDetails.addColumn(LineTotal);
169
170     // Create the Products Metadata
171     Metadata Products = new Metadata(this, OrderDetails, c);
172     Products.setDescription("Products");
173     String query = "select a.ProductID,
174         a.ProductDescription,a.ProductName,";
175     query += " a.CategoryID, a.UnitPrice, a.Picture, ";
176     query += " b.CategoryName";
177     query += " from Products a, Categories b";
178     query += " where a.ProductID = ?";
179     query += " and a.CategoryID = b.CategoryID";
180     Products.setStatement(query);
181     Products.joinOnParentColumn("ProductID","ProductID");
182     Products.open();
183
184     // 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(
197         "DemoData failed to initialize " + e.toString());
198 }
199
200 }

```

## 6.3 Custom Data Binding

### Binding Stock AWT and Swing Components to a Data Source

Binding ordinary AWT or Swing components to a data source involves subclassing the component and having it implement the `DataModelListener` interface. An example is given in `jclass.datasource.examples.components.textfield`. The signature for a data-aware `TextField` is:

```

public class DBTextField extends TextField
    implements DataModelListener, FocusListener, KeyListener

```

Call its constructor to create a new `DBTextField` component and bind it to a particular column in a `MetaDataModel`:

```

public DBTextField(DataModel data_model,
    MetaDataModel meta_data_model,
    String column_name) {
    this();
    setDataBinding(data_model, meta_data_model, column_name);
}

```

Note that the three parameters are the `DataModel`, the `MetaDataModel` for the master-detail level, and the name of the database field. This information is passed to the `setDataBinding` method, which completes the name association and registers the component as a listener for `DataModelEvents`.

See `jclass.datasource.examples.components.textfield.DBTextField` for a full code example.

### **Binding Your Own Components to a Data Source**

A simpler data binding solution exists. This involves extending an AWT or Swing component, then subclassing `FieldDataBinding` from `jclass.datasource.components.FieldDataBinding` in an inner class, and implementing `refreshCell` in that inner class.

```
class FieldDataBinding extends
    jclass.datasource.components.FieldDataBinding
```

The constructor for this class passes the instance of the component defined in the containing class so that if an error is generated an error popup can be presented. After changes have been made to the component, save them by calling either `convertAndSaveItem` or `saveItem`. Use `convertAndSaveItem` to ensure that the data type has been converted to a database type acceptable to `JClass DataSource` and use `saveItem` for a component that returns a `Boolean` and therefore does not need extra conversion into a database-acceptable type. These methods should be called from within the implementation of a listener method. For instance, call `convertAndSaveItem` on a text-type data bound component from the `focusLost` method as part of your `FocusListener` implementation.

*Part* ***II***

*Reference  
Appendices*



# Appendix A

## Bean Properties Reference

[DataBean](#) ■ [DataBeanComponent](#) ■ [DataBeanCustomizer](#) ■ [JCTreeData](#) ■ [TreeDataBeanComponent](#)  
[TreeDataBeanCustomizer](#) ■ [DSdbJNavigator](#) ■ [DSdbJTextField](#) ■ [DSdbJImage](#) ■ [DSdbJCheckbox](#)  
[DSdbJList](#) ■ [DSdbJTextArea](#) ■ [DSdbJLabel](#)

This section contains a listing of the JClass DataSource Bean properties and their default values for the `DataBean`, `JCTreeData`, `DSdbNavigator`, and the data bound components. The properties are arranged alphabetically by property name. The second entry on a 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 DataBean

Property	Type	Default Value
about (G)	<code>java.lang.String</code>	About JClass DataSource
class	<code>java.lang.Class</code>	class <code>jclass.datasource.DataBean</code>
commitPolicy	<code>int</code>	COMMIT_LEAVING_RECORD
currentGlobalBookmark	<code>long</code>	-1
currentGlobalTable	<code>jclass.datasource.DataTableModel</code>	(null)
dataBeanComponent	<code>jclass.datasource.DataBeanComponent</code>	Click to edit.
dataTableTree	<code>jclass.datasource.treemodel.TreeModel</code>	<i>dynamic</i>
description	<code>java.lang.String</code>	Node1
eventsEnabled	<code>boolean</code>	True
listeners	<code>java.lang.Object</code>	(null)

Property	Type	Default Value
metaDataTree	jclass.datasource.treemodel.TreeModel	<i>dynamic</i>
modelName	java.lang.String	DataBean1
modified	boolean	False
version (G)	java.lang.String	JClass DataSource version number for <i>&lt;platform&gt;</i>

## A.2 DataBeanComponent

Property	Type	Default Value
class	java.lang.Class	class jclass.higridd.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	jchigridd0.ser
serializationRequired	boolean	False
structureOnly	boolean	False

## A.3 DataBeanCustomizer

Property	Type	Default Value
alignmentX	float	0.5
alignmentY	float	0.5
background	java.awt.Color	(null)
component	(null)	null
componentCount	int	1

Property	Type	Default Value
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.4 JCTreeData

Property	Type	Default Value
about (G)	java.lang.String	About JClass DataSource
class	java.lang.Class	class jclass.datasource.JCTreeData
currentGlobalBookmark	long	-1
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

Property	Type	Default Value
modified	boolean	(null)
JCTreeDataComponent	jclass.datasource. JCTreeDataComponent	Click to edit.
version (G) (G)	java.lang.String	JClass DataSource version number for <platform>

## A.5 TreeDataBeanComponent

Property	Type	Default Value
class	java.lang.Class	class jclass.datasource. JCTreeDataComponent
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	jcbdtree0.ser
serializationRequired	boolean	False
structureOnly	boolean	False

## A.6 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



Property	Type	Default Value
enabled	<b>boolean</b>	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=105,height=140
name	java.lang.String	pane10
object	java.lang.Object	<b>null</b>
preferredSize	java.awt.Dimension	(null)
visible	<b>boolean</b>	True

## A.7 DSdbJNavigator

Property	Type	Default Value
UIClassID	java.lang.String	<b>not a pluggable look and feel class</b>
accessibleContext	com.sun.java.accessibility.AccessibleContext	<b>dynamic</b>
alignmentX	<b>float</b>	<b>0.5</b>
alignmentY	<b>float</b>	<b>0.5</b>
autoscrolls	<b>boolean</b>	False
background	java.awt.Color	204,204,204
border	com.sun.java.swing.border.Border	<b>null</b>
bounds	java.awt.Rectangle	<b>null</b>
buttonBackground	java.awt.Color	192,192,192
buttonForeground	java.awt.Color	0,0,0

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
commandVisible	<b>boolean</b>	True
component	<b>(null)</b>	null
componentCount	<b>int</b>	7
components	java.awt.Component[]	<b>dynamic</b>
dataBinding	java.lang.String	<b>Select a Data Source.</b>
debugGraphicsOptions	<b>int</b>	<b>0</b>
deleteVisible	<b>boolean</b>	True
doubleBuffered	<b>boolean</b>	True
enabled	<b>boolean</b>	False
firstVisible	<b>boolean</b>	True
focusCycleRoot	<b>boolean</b>	False
focusTraversable	<b>boolean</b>	False
font	java.awt.Font	<b>null</b>
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	<b>null</b>
height	<b>int</b>	0
insertVisible	<b>boolean</b>	True
insets	java.awt.Insets	0, 0, 0, 0
lastVisible	<b>boolean</b>	True
layout	java.awt. LayoutManager	<b>null</b>
managingFocus	<b>boolean</b>	False
maximumSize	java.awt.Dimension	32767, 32767
minimumSize	java.awt.Dimension	width=108,height=17
name	java.lang.String	datanavigator0
nextFocusableComponent	java.awt.Component	<b>null</b>
nextVisible	<b>boolean</b>	True
opaque	<b>boolean</b>	True

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
optimizedDrawingEnabled	<b>boolean</b>	True
paintingTile	<b>boolean</b>	False
preferredSize	java.awt.Dimension	width=239,height=17
previousVisible	<b>boolean</b>	True
registeredKeyStrokes	com.sun.java.swing. KeyStroke[]	<b>dynamic</b>
requestFocusEnabled	<b>boolean</b>	True
rootPane	com.sun.java.swing. JRootPane	<b>null</b>
statusBackground	java.awt.Color	255,255,255
statusForeground	java.awt.Color	0,0,0
statusVisible	<b>boolean</b>	True
toolTipText	java.lang.String	<b>null</b>
topLevelAncestor	java.awt.Container	<b>null</b>
validateRoot	<b>boolean</b>	False
visible	<b>boolean</b>	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	<b>int</b>	0
x	<b>int</b>	0
y	<b>int</b>	0

## **A.8 DSdbJTextField**

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
UI	com.sun.java.swing. plaf.TextUI	<b>dynamic</b>
UIClassID	java.lang.String	TextFieldUI
accessibleContext	com.sun.java. accessibility. AccessibleContext	<b>dynamic</b>

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
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
doubleBuffered	boolean	False
editable	boolean	True
enabled	boolean	True
focusAccelerator	char	
focusCycleRoot	boolean	False
focusTraversable	boolean	True
font	java.awt.Font	null

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null
height	<b>int</b>	0
highlighter	com.sun.java.swing. text.Highlighter	<b>dynamic</b>
horizontalAlignment	<b>int</b>	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	<b>dynamic</b>
layout	java.awt. LayoutManager	<b>null</b>
managingFocus	<b>boolean</b>	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	<b>null</b>
nextFocusableComponent	java.awt.Component	<b>null</b>
opaque	<b>boolean</b>	True
optimizedDrawingEnabled	<b>boolean</b>	True
paintingTile	<b>boolean</b>	False
preferredScrollable ViewportSize	java.awt.Dimension	width=4,height=19
preferredSize	java.awt.Dimension	width=4,height=19
registeredKeyStrokes	com.sun.java.swing. KeyStroke[]	<b>dynamic</b>
requestFocusEnabled	<b>boolean</b>	True
rootPane	com.sun.java.swing. JRootPane	<b>null</b>
scrollOffset	<b>int</b>	0

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
scrollableTracks ViewportHeight	<b>boolean</b>	False
scrollableTracks ViewportWidth	<b>boolean</b>	False
selectedText	java.lang.String	<b>null</b>
selectedTextColor	java.awt.Color	0,0,0
selectionColor	java.awt.Color	204,204,255
selectionEnd	<b>int</b>	0
selectionStart	<b>int</b>	0
text	java.lang.String	
toolTipText	java.lang.String	<b>null</b>
topLevelAncestor	java.awt.Container	<b>null</b>
validateRoot	<b>boolean</b>	True
visible	<b>boolean</b>	True
visibleRect	java.awt.Rectangle	java.awt.Rectangle[x=0,y=0 ,width=0,height=0]
width	<b>int</b>	0
x	<b>int</b>	0
y	<b>int</b>	0

## A.9 DSdbJImage

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
UIClassID	java.lang.String	<b>not a pluggable look and feel class</b>
accessibleContext	com.sun.java.accessibility.AccessibleContext	<b>null</b>
alignmentX	<b>float</b>	0.5
alignmentY	<b>float</b>	0.5

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
autoscrolls	<b>boolean</b>	False
background	java.awt.Color	<b>null</b>
border	com.sun.java.swing. border.Border	<b>null</b>
bounds	java.awt.Rectangle	<b>null</b>
component	<b>(null)</b>	<b>null</b>
componentCount	<b>int</b>	0
components	java.awt.Component[]	<b>dynamic</b>
dataBinding	java.lang.String	<b>Select a Data Source.</b>
debugGraphicsOptions	<b>int</b>	0
doubleBuffered	<b>boolean</b>	False
enabled	<b>boolean</b>	True
focusCycleRoot	<b>boolean</b>	False
focusTraversable	<b>boolean</b>	False
font	java.awt.Font	<b>null</b>
foreground	java.awt.Color	<b>null</b>
graphics	java.awt.Graphics	<b>null</b>
height	<b>int</b>	0
insets	java.awt.Insets	0, 0, 0, 0
layout	java.awt. LayoutManager	<b>null</b>
managingFocus	<b>boolean</b>	False
maximumSize	java.awt.Dimension	[width=32767,height=32767
minimumSize	java.awt.Dimension	0, 0
name	java.lang.String	<b>null</b>
nextFocusableComponent	java.awt.Component	<b>null</b>
opaque	<b>boolean</b>	False
optimizedDrawingEnabled	<b>boolean</b>	True
paintingTile	<b>boolean</b>	False

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
preferredSize	java.awt.Dimension	0, 0
registeredKeyStrokes	com.sun.java.swing. KeyStroke[]	<b>dynamic</b>
requestFocusEnabled	<b>boolean</b>	True
rootPane	com.sun.java.swing. JRootPane	<b>null</b>
toolTipText	java.lang.String	<b>null</b>
topLevelAncestor	java.awt.Container	<b>null</b>
validateRoot	<b>boolean</b>	False
visible	<b>boolean</b>	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	<b>int</b>	0
x	<b>int</b>	0
y	<b>int</b>	0

## **A.10 DSdbJCheckbox**

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
UI	com.sun.java.swing. plaf.ButtonUI	<b>dynamic</b>
UIClassID	java.lang.String	CheckBoxUI
accessibleContext	com.sun.java. accessibility. AccessibleContext	<b>dynamic</b>
actionCommand	java.lang.String	
alignmentX	<b>float</b>	0.0
alignmentY	<b>float</b>	0.0
autoscrolls	<b>boolean</b>	False
background	java.awt.Color	204,204,204



<b>Property</b>	<b>Type</b>	<b>Default Value</b>
border	com.sun.java.swing. border.Border	dynamic
borderPainted	<b>boolean</b>	False
bounds	java.awt.Rectangle	null
component	(null)	null
componentCount	<b>int</b>	0
components	java.awt.Component[]	dynamic
dataBinding	java.lang.String	Select a Data Source.
debugGraphicsOptions	<b>int</b>	0
disabledIcon	com.sun.java.swing. Icon	null
disabledSelectedIcon	com.sun.java.swing. Icon	null
doubleBuffered	<b>boolean</b>	False
enabled	<b>boolean</b>	True
focusCycleRoot	<b>boolean</b>	False
focusPainted	<b>boolean</b>	True
focusTraversable	<b>boolean</b>	True
font	java.awt.Font	null
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	null
height	<b>int</b>	0
horizontalAlignment	<b>int</b>	2
horizontalTextPosition	<b>int</b>	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	<b>boolean</b>	False

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
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	<b>char</b>	<b>null</b>
model	com.sun.java.swing. ButtonModel	<b>dynamic</b>
name	java.lang.String	<b>null</b>
nextFocusableComponent	java.awt.Component	<b>null</b>
opaque	<b>boolean</b>	False
optimizedDrawingEnabled	<b>boolean</b>	True
paintingTile	<b>boolean</b>	False
preferredSize	java.awt.Dimension	width=23,height=23
pressedIcon	com.sun.java.swing. Icon	<b>null</b>
registeredKeyStrokes	com.sun.java.swing.Key Stroke[]	<b>dynamic</b>
requestFocusEnabled	<b>boolean</b>	True
rolloverEnabled	<b>boolean</b>	False
rolloverIcon	com.sun.java.swing. Icon	<b>null</b>
rolloverSelectedIcon	com.sun.java.swing. Icon	<b>null</b>
rootPane	com.sun.java.swing. JRootPane	<b>null</b>
selected	<b>boolean</b>	False
selectedIcon	com.sun.java.swing. Icon	<b>null</b>
selectedObjects	java.lang.Object[]	<b>null</b>
text	java.lang.String	
toolTipText	java.lang.String	<b>null</b>
topLevelAncestor	java.awt.Container	<b>null</b>

Property	Type	Default Value
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.11 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
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

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
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
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

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
nextFocusableComponent	java.awt.Component	<b>null</b>
opaque	<b>boolean</b>	True
optimizedDrawingEnabled	<b>boolean</b>	True
paintingTile	<b>boolean</b>	False
preferredScrollable ViewportSize	java.awt.Dimension	width=256, height=128
preferredSize	java.awt.Dimension	0, 0
prototypeCellValue	java.lang.Object	<b>null</b>
registeredKeyStrokes	com.sun.java.swing. KeyStroke[]	<b>dynamic</b>
requestFocusEnabled	<b>boolean</b>	True
rootPane	com.sun.java.swing. JRootPane	<b>null</b>
scrollableTracks ViewportHeight	<b>boolean</b>	False
scrollableTracks ViewportWidth	<b>boolean</b>	False
selectedIndex	<b>int</b>	-1
selectedIndices	<b>int[]</b>	[I@1f3bf5
selectedValue	java.lang.Object	<b>null</b>
selectedValues	java.lang.Object[]	<b>dynamic</b>
selectionBackground	java.awt.Color	204,204,255
selectionEmpty	<b>boolean</b>	True
selectionForeground	java.awt.Color	0,0,0
selectionInterval	<b>(null)</b>	<b>null</b>
selectionMode	<b>int</b>	2
selectionModel	com.sun.java.swing. ListSelectionMode	<b>dynamic</b>
toolTipText	java.lang.String	<b>null</b>
topLevelAncestor	java.awt.Container	<b>null</b>

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
validateRoot	<b>boolean</b>	False
valueIsAdjusting	<b>boolean</b>	False
visible	<b>boolean</b>	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
visibleRowCount	<b>int</b>	8
width	<b>int</b>	0
x	<b>int</b>	0
y	<b>int</b>	0

## **A.12 DSdbJTextArea**

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
UI	com.sun.java.swing.plaf.TextUI	<b>dynamic</b>
UIClassID	java.lang.String	TextAreaUI
accessibleContext	com.sun.java.accessibility.AccessibleContext	<b>dynamic</b>
actions	com.sun.java.swing.Action[]	<b>dynamic</b>
alignmentX	<b>float</b>	0.5
alignmentY	<b>float</b>	0.5
autoscrolls	<b>boolean</b>	True
background	java.awt.Color	255,255,255
border	com.sun.java.swing.border.Border	<b>dynamic</b>
bounds	java.awt.Rectangle	<b>null</b>
caret	com.sun.java.swing.text.Caret	<b>dynamic</b>
caretColor	java.awt.Color	0,0,0

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
caretPosition	<b>int</b>	0
columns	<b>int</b>	0
component	<b>(null)</b>	<b>null</b>
componentCount	<b>int</b>	0
components	java.awt.Component[]	<b>dynamic</b>
dataBinding	java.lang.String	<b>Select a Data Source.</b>
debugGraphicsOptions	<b>int</b>	0
disabledTextColor	java.awt.Color	153,153,153
document	com.sun.java.swing. text.Document	<b>dynamic</b>
doubleBuffered	<b>boolean</b>	False
editable	<b>boolean</b>	True
enabled	<b>boolean</b>	True
focusAccelerator	<b>char</b>	
focusCycleRoot	<b>boolean</b>	False
focusTraversable	<b>boolean</b>	True
font	java.awt.Font	<b>null</b>
foreground	java.awt.Color	0,0,0
graphics	java.awt.Graphics	<b>null</b>
height	<b>int</b>	0
highlighter	com.sun.java.swing. text.Highlighter	<b>dynamic</b>
insets	java.awt.Insets	2, 2, 2, 2
keymap	com.sun.java.swing. text.Keymap	<b>dynamic</b>
layout	java.awt. LayoutManager	<b>null</b>
lineCount	<b>int</b>	1
lineEndOffset	<b>(null)</b>	<b>null</b>
lineOfOffset	<b>(null)</b>	<b>null</b>

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
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
preferredScrollable ViewportSize	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
rows	int	0
scrollableTracks ViewportHeight	boolean	False
scrollableTracks ViewportWidth	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



Property	Type	Default Value
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.13 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
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

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
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
labelFor	java.awt.Component	null
layout	java.awt.Layout Manager	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

<b>Property</b>	<b>Type</b>	<b>Default Value</b>
paintingTile	<b>boolean</b>	False
preferredSize	java.awt.Dimension	0, 0
registeredKeyStrokes	com.sun.java.swing. KeyStroke[]	<b>dynamic</b>
requestFocusEnabled	<b>boolean</b>	True
rootPane	com.sun.java.swing. JRootPane	<b>null</b>
text	java.lang.String	
toolTipText	java.lang.String	<b>null</b>
topLevelAncestor	java.awt.Container	<b>null</b>
validateRoot	<b>boolean</b>	False
verticalAlignment	<b>int</b>	0
verticalTextPosition	<b>int</b>	0
visible	<b>boolean</b>	True
visibleRect	java.awt.Rectangle	0, 0, 0, 0
width	<b>int</b>	0
x	<b>int</b>	0
y	<b>int</b>	0



# Appendix B

## Distributing Applets and Applications

*Using JarMaster to Customize the Deployment Archive*

### 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 *jdbcdatasource.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 as part of the JClass DesktopViews suite of products. For more details please refer to [Quest Software's Web site](#).



# Index

## A

- abstract relationship 14
- access
  - specify tables and fields 26
- access rights
  - Data Access Tab 79
- accessing a database 48
  - middleware products 50
  - the JDBC-ODBC bridge 50
  - Type 1 driver 49
  - Type driver 49
- advance 56
- ambiguous column names 40
- API 3
- append 54, 55
- applets
  - distributing 133
- applications
  - distributing 133
- assumptions 2
- atBegin 56
- atEnd 56

## B

- BaseDataTable 15, 25, 34
- BaseMetaData 15
- batching HiGrid updates 42
- Bean 69
  - Data Bean 71
  - Data Bean editor 72
  - Data Navigator 40
  - DataSource
    - custom implementation 88
  - DSdbNavigator, properties 94
  - JARs, installing 70
  - properties reference 109
  - serialization file 71
  - saving 73
  - Tree Data 82
- BeanBox 27
- binding
  - a component to a single level 90
  - AWT and SWING components to a data source 105
  - data binding, managing 9

- interface for single-level data binding 9
- specifying path names 91, 92
- your own components to a data source 106

- BindingModel 9
- bookmark 19
- current 15
- navigating 23

- Borland JBuilder
  - data binding 88

## C

- choosing tables in the Data Bean
  - tables
    - choosing 75
- class 13, 43
  - BaseDataTable 34
  - DataModel 34
  - DataModelEvent, constants 62
  - diagram, data model 61
  - TreeModel 34
  - useful, code snippets 37
- clone 56
- code snippet
  - connecting to a database via a JDBC-ODBC bridge 38
  - joining tables 38
  - refreshing tables 39
  - setting permissions 39
  - setting the top-level query 38
  - useful classes 37
- column
  - accessing from a database 57
  - ambiguous names 40
  - excluding from update operations 60
  - property 57
  - virtual 58
    - computation order 59
    - computed columns 80
    - set via a customizer 80
  - virtual, define 81
- comments on product 6
- commit policy 15, 37, 101
  - COMMIT\_LEAVING\_ANCESTOR 37, 53
  - COMMIT\_LEAVING\_RECORD 37, 53
  - COMMIT\_MANUALLY 37, 53
  - Data Access Tab 79

- global cursor 48
  - set 26
  - setting 53
- COMMIT\_LEAVING\_ANCESTOR 53
- COMMIT\_LEAVING\_RECORD 53
- COMMIT\_MANUALLY 53
- component
  - binding
    - IDE 91
    - path 91
    - standard method 91
  - binding programmatically 91
  - binding to a data source 106
  - binding to a meta data level 90
  - data binding, other 97
  - data bound 13, 91
  - data control 40
  - DSdbNavigator 93
  - types of data bound 89
- connection
  - JDBC-ODBC 16
- constructor 14
  - root 101
- createTable 24
- current bookmark 15
  - DataTableModel 48
- current path 21, 22
- cursor
  - global 15
- custom implementations 41
- customizer 51
  - choosing tables 75
  - JDBC connection 43
  - setting a query 76

## D

- data
  - binding, managing 9
  - control components 40
  - multiple data views 13
  - structure 14
  - traversing 54
  - unbound 41
- Data Access tab 30
- Data Bean 71
  - editor 72
- data binding 90
  - a component to a single level 90
  - Borland JBuilder 88
  - component programmatically 91
  - component through an IDE 91
  - component to meta data-level 90
  - constructing 52
  - custom 105
  - JClass components in an IDE 91
  - navigator programmatically 92
  - navigator through an IDE 92
  - other components 97
  - single-level 90
  - via JDBC 39
- data bound components 10, 13, 87, 89
- data integrity
  - handling exceptions 67
- data manipulation language 26
  - DELETE 26
  - INSERT 26
  - SELECT 26
  - UPDATE 26
- data model 44, 47, 90
  - classes and interfaces diagram 61
  - event, methods 61
  - highlights 16
  - instantiating 50
  - methods 44
  - setting 24
  - setting programmatically 24
  - unbound 24
- Data Model tab 28
- data navigator 87
- Data Navigator Bean 40
- data source
  - binding components 106
  - connection 13
  - events 60
  - listeners 60
- data table tree 47
- data tree
  - creating root level 25
- database
  - accessing rows and columns 57
  - accessing with a JDBC-ODBC bridge 50
  - accessing with a Type 1 driver 49
  - accessing with a Type 4 driver 49
  - accessing with Middleware products 50
  - column properties 57
  - connection 73
  - connection, specifying 48
  - JDBC-ODBC connection 16
  - requering 56
  - set commit policy to be used when updating 26
- DataBean 27, 109
  - about 109
  - class 109
  - commitPolicy 109
  - currentGlobalBookmark 109
  - currentGlobalTable 109
  - custom editor 28
  - Data Access Tab 79
  - dataBeanComponent 109
  - dataTableTree 109



- description 109
- Driver Table Tab 77
- eventsEnabled 109
- listeners 109
- metaDataTree 110
- modelName 110
- modified 110
- Set button 32
- version 110
- Virtual Columns Tab 80
- DataBeanComponent 110
  - class 110
  - dataSourceNames 110
  - dataSources 110
  - resourceName 110
  - root 110
  - serializationFile 110
  - serializationRequired 110
  - structureOnly 110
- DataBeanComponentEditor 72
- DataBeanCustomizer 110
  - alignmentX 110
  - alignmentY 110
  - background 110
  - component 110
  - componentCount 110
  - components 111
  - enabled 111
  - font 111
  - foreground 111
  - insets 111
  - layout 111
  - maximumSize 111
  - minimumSize 111
  - name 111
  - object 111
  - preferredSize 111
  - visible 111
- DataModel 15, 24, 34
  - global cursor 48
- DataModelEvent
  - class constants 62
- DataModelListener
  - methods 64
- DataSource Bean
  - custom implementation 88
- DataTable
  - subclass 15
- DataTableAbstractionLayer 34, 35
- DataTableModel 15, 24
  - current bookmark 48
- DataTables 15
- DemoData program 100
- design-time maximum number of rows 74
- DML
  - data manipulation language 26
  - DELETE 26
  - INSERT 26
  - SELECT 26
  - UPDATE 26
- driver
  - manager 100
  - table 76
  - Type 1 49
  - Type 4 49
  - Type 4, definition 49
- Driver Table Tab 85
- DSdbJCheckbox 120
  - accessibleContext 120
  - actionCommand 120
  - alignmentX 120
  - alignmentY 120
  - autoscrolls 120
  - background 120
  - border 121
  - borderPainted 121
  - bounds 121
  - component 121
  - componentCount 121
  - components 121
  - dataBinding 121
  - debugGraphicsOptions 121
  - disabledIcon 121
  - disabledSelectedIcon 121
  - doubleBuffered 121
  - enabled 121
  - focusCycleRoot 121
  - focusPainted 121
  - focusTraversable 121
  - font 121
  - foreground 121
  - graphics 121
  - height 121
  - horizontalAlignment 121
  - horizontalTextPosition 121
  - icon 121
  - insets 121
  - label 121
  - layout 121
  - managingFocus 121
  - margin 122
  - maximumSize 122
  - minimumSize 122
  - mnemonic 122
  - model 122
  - name 122
  - nextFocusableComponent 122
  - opaque 122
  - optimizedDrawingEnabled 122
  - paintingTile 122
  - preferredSize 122
  - pressedIcon 122

- registeredKeyStrokes 122
- requestFocusEnabled 122
- rolloverEnabled 122
- rolloverIcon 122
- rolloverSelectedIcon 122
- rootPane 122
- selected 122
- selectedIcon 122
- selectedObjects 122
- text 122
- toolTipText 122
- topLevelAncestor 122
- UI 120
- UIClassID 120
- validateRoot 123
- verticalAlignment 123
- verticalTextPosition 123
- visible 123
- visibleRect 123
- width 123
- x 123
- y 123
- DSdbjImage 118
  - accessibleContext 118
  - alignmentX 118
  - alignmentY 118
  - autoscrolls 119
  - background 119
  - border 119
  - bounds 119
  - component 119
  - componentCount 119
  - components 119
  - dataBinding 119
  - debugGraphicsOptions 119
  - doubleBuffered 119
  - enabled 119
  - focusCycleRoot 119
  - focusTraversable 119
  - font 119
  - foreground 119
  - graphics 119
  - height 119
  - insets 119
  - layout 119
  - managingFocus 119
  - maximumSize 119
  - minimumSize 119
  - name 119
  - nextFocusableComponent 119
  - opaque 119
  - optimizedDrawingEnabled 119
  - paintingTile 119
  - preferredSize 120
  - registeredKeyStrokes 120
  - requestFocusEnabled 120
- rootPane 120
- toolTipText 120
- topLevelAncestor 120
- UIClassID 118
- validateRoot 120
- visible 120
- visibleRect 120
- width 120
- x 120
- y 120
- DSdbjLabel 129
  - accessibleContext 129
  - alignmentX 129
  - alignmentY 129
  - autoscrolls 129
  - background 129
  - border 129
  - bounds 129
  - component 129
  - componentCount 129
  - components 129
  - dataBinding 130
  - debugGraphicsOptions 130
  - disabledIcon 130
  - displayedMnemonic 130
  - doubleBuffered 130
  - enabled 130
  - focusCycleRoot 130
  - focusTraversable 130
  - font 130
  - foreground 130
  - graphics 130
  - height 130
  - horizontalAlignment 130
  - horizontalTextPosition 130
  - icon 130
  - iconTextGap 130
  - insets 130
  - lableFor 130
  - layout 130
  - managingFocus 130
  - maximumSize 130
  - minimumSize 130
  - name 130
  - nextFocusableComponent 130
  - opaque 130
  - optimizedDrawingEnabled 130
  - paintingTile 131
  - preferredSize 131
  - registeredKeyStrokes 131
  - requestFocusEnabled 131
  - rootPane 131
  - text 131
  - toolTipText 131
  - topLevelAncestor 131
  - UI 129

- UIClassID 129
- validateRoot 131
- verticalAlignment 131
- verticalTextPosition 131
- visible 131
- visibleRect 131
- width 131
  - x 131
  - y 131
- DSdbjList 123
  - accessibleContext 123
  - alignmentX 123
  - alignmentY 123
  - anchorSelectionIndex 123
  - autoscrolls 123
  - background 123
  - border 123
  - bounds 123
  - cellRenderer 123
  - component 123
  - componentCount 124
  - components 124
  - dataBinding 124
  - debugGraphicsOptions 124
  - doubleBuffered 124
  - enabled 124
  - firstVisibleIndex 124
  - fixedCellHeight 124
  - fixedCellWidth 124
  - focusCycleRoot 124
  - focusTraversable 124
  - font 124
  - foreground 124
  - graphics 124
  - height 124
  - insets 124
  - lastVisibleIndex 124
  - layout 124
  - leadSelectionIndex 124
  - listData 124
  - managingFocus 124
  - maximumSize 124
  - maxSelectionIndex 124
  - minimumSize 124
  - minSelectionIndex 124
  - model 124
  - name 124
  - nextFocusableComponent 125
  - opaque 125
  - optimizedDrawingEnabled 125
  - paintingTile 125
  - preferredScrollableViewportSize 125
  - preferredSize 125
  - prototypeCellValue 125
  - registeredKeyStrokes 125
  - requestFocusEnabled 125

- rootPane 125
- scrollableTracksViewportHeight 125
- scrollableTracksViewportWidth 125
- selectedIndex 125
- selectedIndices 125
- selectedValue 125
- selectedValues 125
- selectionBackground 125
- selectionEmpty 125
- selectionForeground 125
- selectionInterval 125
- selectionMode 125
- selectionModel 125
- toolTipText 125
- topLevelAncestor 125
- UI 123
- UIClassIDt 123
- validateRoot 126
- valueIsAdjusting 126
- visible 126
- visibleRect 126
- visibleRowCount 126
- width 126
  - x 126
  - y 126
- DSdbjTextArea 126
  - accessibleContext 126
  - actions 126
  - alignmentX 126
  - alignmentY 126
  - autoscrolls 126
  - background 126
  - border 126
  - bounds 126
  - caret 126
  - caretColor 126
  - caretPosition 127
  - columns 127
  - component 127
  - componentCount 127
  - components 127
  - dataBinding 127
  - debugGraphicsOptions 127
  - disabledTextColor 127
  - document 127
  - doubleBuffered 127
  - editable 127
  - enabled 127
  - focusAccelerator 127
  - focusCycleRoot 127
  - focusTraversable 127
  - font 127
  - foreground 127
  - graphics 127
  - height 127
  - highlighter 127

- insets 127
- keymap 127
- layout 127
- lineCount 127
- lineEndOffset 127
- lineOfOffset 127
- lineStartOffset 128
- lineWrap 128
- managingFocus 128
- margin 128
- maximumSize 128
- minimumSize 128
- name 128
- nextFocusableComponent 128
- opaque 128
- optimizedDrawingEnabled 128
- paintingTile 128
- preferredScrollableViewportSize 128
- preferredSize 128
- registeredKeyStrokes 128
- requestFocusEnabled 128
- rootPane 128
- rows 128
- scrollableTracksViewportHeight 128
- scrollableTracksViewportWidth 128
- selectedText 128
- selectedTextColor 128
- selectionColor 128
- selectionEnd 128
- selectionStart 128
- tabSize 128
- text 129
- toolTipText 129
- topLevelAncestor 129
- UI 126
- UIClassID 126
- validateRoot 129
- visible 129
- visibleRect 129
- width 129
- wrapStyleWord 129
- x 129
- y 129

DsdbjTextField 115

- accessibleContext 115
- actionCommand 116
- actions 116
- alignmentX 116
- alignmentY 116
- autoscrolls 116
- background 116
- border 116
- bounds 116
- caret 116
- caretColor 116
- caretPosition 116
- columns 116
- component 116
- componentCount 116
- components 116
- dataBinding 116
- debugGraphicsOptions 116
- disabledTextColor 116
- document 116
- doubleBuffered 116
- editable 116
- enabled 116
- focusAccelerator 116
- focusCycleRoot 116
- focusTraversable 116
- font 116
- foreground 117
- graphics 117
- height 117
- highlighter 117
- horizontalAlignment 117
- horizontalVisibility 117
- insets 117
- keymap 117
- layout 117
- managingFocus 117
- margin 117
- maximumSize 117
- minimumSize 117
- name 117
- nextFocusableComponent 117
- opaque 117
- optimizedDrawingEnabled 117
- paintingTile 117
- preferredScrollableViewportSize 117
- preferredSize 117
- registeredKeyStrokes 117
- requestFocusEnabled 117
- rootPane 117
- scrollableTracksViewportHeight 118
- scrollableTracksViewportWidth 118
- scrollOffset 117
- selectedText 118
- selectedTextColor 118
- selectionColor 118
- selectionEnd 118
- selectionStart 118
- text 118
- toolTipText 118
- topLevelAncestor 118
- UI 115
- UIClassID 115
- validateRoot 118
- visible 118
- visibleRect 118
- width 118
- x 118

- y 118
- DSdbNavigator 69, 113
  - accessibleContext 113
  - alignmentX 113
  - alignmentY 113
  - autoscrolls 113
  - background 113
  - Bean properties 94
  - border 113
  - bounds 113
  - buttonBackground 113
  - buttonForeground 113
  - commandVisible 114
  - component 114
  - componentCount 114
  - components 114
  - dataBinding 114
  - debugGraphicsOptions 114
  - deleteVisible 114
  - doubleBuffered 114
  - enabled 114
  - firstVisible 114
  - focusCycleRoot 114
  - focusTraversable 114
  - font 114
  - foreground 114
  - functions 93
  - graphics 114
  - height 114
  - insertVisible 114
  - insets 114
  - lastVisible 114
  - layout 114
  - managingFocus 114
  - maximumSize 114
  - minimumSize 114
  - name 114
  - nextFocusableComponent 114
  - nextVisible 114
  - opaque 114
  - optimizedDrawingEnabled 115
  - paintingTile 115
  - preferredSize 115
  - previousVisible 115
  - registeredKeyStrokes 115
  - requestFocusEnabled 115
  - rootPane 115
  - statusBackground 115
  - statusForeground 115
  - statusVisible 115
  - ToolTipText 115
  - topLevelAncestor 115
  - UIClassID 113
  - validateRoot 115
  - visible 115
  - visibleRect 115

- width 115
- x 115
- y 115
- dynamic bookmark 14

## E

- entity-relationship diagram
  - for sample database 99
- events
  - data source 60
  - DataModelEvent 61
- exceptions 67
  - handling 67

## F

- FAQs 5
- feature overview 1
- fields
  - specify for access 26
  - specifying 51

## G

- get 56
- getAncestorBookmark 24
- getAncestors 24
- getChildren 54, 55
- getCurrentBookmark 15
- getCurrentGlobalBookmark 15
- getData 54
- getFirstChild 54, 55
- getIterator 54, 55
- getLastChild 54, 55
- getMetaData 24
- getMetaDataTree 24
- getNextChild 54, 55
- getNextSibling 54
- getParent 54, 55
- getParentBookmark 24
- getPreviousChild 54, 55
- getPreviousSibling 55
- getRowIdentifier 24
- getRowIndex 24
- getRows 24
- getTableName 39
  - global cursor 15, 20, 22
    - commit 48
    - commits 35
    - DataModel 48
- GUI
  - JClass HiGrid 10

## H

- hasChildren 55
- hasMoreElements 56
- hierarchical relationships
  - meta data objects 19

## I

- IDE
  - binding a component 91
  - binding a navigator 92
- insert 55
- interfaces
  - DataTableAbstractionLayer 34
  - diagram, data model 61
  - MetaDataModel 43
  - VirtualColumnModel 58
- internationalization 11
- introducing JClass DataSource 1
- isChildOf(TreeNode) 55

## J

- JAR
  - installing 70
  - optimizing 133
- JarMaster 133
- JBuilder
  - data binding 88
- JCData 10, 28
- JClass Chart 10
- JClass Field 10
- JClass HiGrid 10, 13
- JClass JarMaster 133
- JClass LiveTable 10
- JClass technical support 5
  - contacting 5
- JCTreeData 10, 111
  - about 111
  - class 111
  - currentGlobalBookmark 111
  - currentGlobalTable 111
  - dataTableTree 111
  - eventsEnabled 111
  - JCTreeData 112
  - listeners 111
  - metaDataTable 111
  - modelName 111
  - modified 112
  - version 112
- JDBC 13
  - binding data to the source 39
  - using customizers to specify the connection 43

- JDBC-ODBC 13
  - bridge
    - making a database connection 16
  - bridge for a database connection 16
  - bridge, database access 50
  - bridge, Type 1 driver 49
- join 101
  - code example 38
  - joining tables 38
  - setting a join in a Data Bean 76

## L

- license 3
- licensing 3
- listeners
  - data source 60

## M

- managing data binding 9
- manual overview 2
- meta data
  - abstract model 47
  - binding a component 90
  - defining its structure 14
  - model 13, 14, 17
  - object
    - bare 24
    - hierarchical relationships 19
    - hierarchy 24
  - specifying 51
  - storing 48
  - structure 47
  - tree 47
  - tree structure 19
- MetaData 14, 26
  - subclass 15
- MetaDataModel 15, 24, 25, 43
  - interface 48
- methods 13, 43
  - data model 44
  - data model event 61
  - for traversing data 54
    - advance, TreeIteratorModel 56
    - append, TreeModel 54
    - append, TreeNodeModel 55
    - atBegin, TreeIteratorModel 56
    - atEnd, TreeIteratorModel 56
    - clone, TreeIteratorModel 56
    - get, TreeIteratorModel 56
    - getChildren, TreeModel 54
    - getChildren, TreeNodeModel 55
    - getData, TreeModel 54

- getFirstChild, Tree Model 54
- getFirstChild, TreeNodeModel 55
- getIterator, TreeModel 54
- getIterator, TreeNodeModel 55
- getLastChild, TreeModel 54
- getLastChild, TreeNodeModel 55
- getNextChild, TreeModel 54
- getNextChild, TreeNodeModel 55
- getNextSibling, TreeModel 54
- getParent, TreeModel 54
- getParent, TreeNodeModel 55
- getPreviousChild, TreeModel 54
- getPreviousChild, TreeNodeModel 55
- getPreviousSibling, TreeModel 55
- hasChildren, TreeModel 55
- hasChildren, TreeNodeModel 55
- hasMoreElements, TreeIteratorModel 56
- insert, TreeModel 55
- insert, TreeNodeModel 55
- isChildOf(TreeNode), TreeModel 55
- nextElement, TreeIteratorModel 56
- remove, TreeModel 55
- remove, TreeNodeModel 55
- removeChildren, TreeModel 55
- removeChildren, TreeNodeModel 55
- TreeIteratorModel 56
- TreeModel 54
- TreeNodeModel 55
- getTableNames 39
- moveToRow 35
- setColumnTableRelations 40
- setDataBinding 91, 92
- middleware
  - accessing a database 50
- model-view-controller 14
- moveToRow 35
- multiple data views 13

## N

- names
  - table 39
- navigator 92
  - binding
    - IDE 92
    - path 92
    - standard method 92
  - binding programmatically 92
  - binding to MetaData level 92
  - functions 92
  - Swing support 92
- Navigator Bean 40
- nextElement 56
- Node Properties Editor 28

## O

- organization
  - of JClass DataSource 14
- overview
  - JClass DataSource 13
  - of the manual 2

## P

- permissions
  - setting 39
- placeholder
  - question mark 26
- policy 48
- prepared statements 26
- product feedback 6
- property
  - Bean, reference 109
  - column 57
  - DataBean 109
    - about 109
    - class 109
    - commitPolicy 109
    - currentGlobalBookmark 109
    - currentGlobalTable 109
    - dataBeanComponent 109
    - dataTableTree 109
    - description 109
    - eventsEnabled 109
    - listeners 109
    - metaDataTree 110
    - modelName 110
    - modified 110
    - version 110
  - DataBeanComponent 110
    - class 110
    - dataSourceNames 110
    - dataSources 110
    - resourceName 110
    - root 110
    - serializationFile 110
    - serializationRequired 110
    - structureOnly 110
  - DataBeanCustomizer 110
    - alignmentX 110
    - alignmentY 110
    - background 110
    - component 110
    - componentCount 110
    - components 111
    - enabled 111
    - font 111
    - foreground 111
    - insets 111

- layout 111
- maximumSize 111
- minimumSize 111
- name 111
- object 111
- preferredSize 111
- visible 111
- DSdbJCheckbox 120
  - accessibleContext 120
  - actionCommand 120
  - alignmentX 120
  - alignmentY 120
  - autoscrolls 120
  - background 120
  - border 121
  - borderPainted 121
  - bounds 121
  - component 121
  - componentCount 121
  - components 121
  - dataBinding 121
  - debugGraphicsOptions 121
  - disabledIcon 121
  - disabledSelectedIcon 121
  - doubleBuffered 121
  - enabled 121
  - focusCycleRoot 121
  - focusPainted 121
  - focusTraversable 121
  - font 121
  - foreground 121
  - graphics 121
  - height 121
  - horizontalAlignment 121
  - horizontalTextPosition 121
  - icon 121
  - insets 121
  - label 121
  - layout 121
  - managingFocus 121
  - margin 122
  - maximumSize 122
  - minimumSize 122
  - mnemonic 122
  - model 122
  - name 122
  - nextFocusableComponent 122
  - opaque 122
  - optimizedDrawingEnabled 122
  - paintingTile 122
  - preferredSize 122
  - pressedIcon 122
  - registeredKeyStrokes 122
  - requestFocusEnabled 122
  - rolloverEnabled 122
  - rolloverIcon 122
  - rolloverSelectedIcon 122
  - rootPane 122
  - selected 122
  - selectedIcon 122
  - selectedObjects 122
  - text 122
  - toolTipText 122
  - topLevelAncestor 122
  - UI 120
  - UIClassID 120
  - validateRoot 123
  - verticalAlignment 123
  - verticalTextPosition 123
  - visible 123
  - visibleRect 123
  - width 123
  - x 123
  - y 123
- DSdbJImage 118
  - accessibleContext 118
  - alignmentX 118
  - alignmentY 118
  - autoscrolls 119
  - background 119
  - border 119
  - bounds 119
  - component 119
  - componentCount 119
  - components 119
  - dataBinding 119
  - debugGraphicsOptions 119
  - doubleBuffered 119
  - enabled 119
  - focusCycleRoot 119
  - focusTraversable 119
  - font 119
  - foreground 119
  - graphics 119
  - height 119
  - insets 119
  - layout 119
  - managingFocus 119
  - maximumSize 119
  - minimumSize 119
  - name 119
  - nextFocusableComponent 119
  - opaque 119
  - optimizedDrawingEnabled 119
  - paintingTile 119
  - preferredSize 120
  - registeredKeyStrokes 120
  - requestFocusEnabled 120
  - rootPane 120
  - toolTipText 120
  - topLevelAncestor 120
  - UIClassID 118



- validateRoot 120
- visible 120
- visibleRect 120
- width 120
- x 120
- y 120
- DSdbjLabel 129
  - accessibleContext 129
  - alignmentX 129
  - alignmentY 129
  - autoscrolls 129
  - background 129
  - border 129
  - bounds 129
  - component 129
  - componentCount 129
  - components 129
  - dataBinding 130
  - debugGraphicsOptions 130
  - disabledIcon 130
  - displayedMnemonic 130
  - doubleBuffered 130
  - enabled 130
  - focusCycleRoot 130
  - focusTraversable 130
  - font 130
  - foreground 130
  - graphics 130
  - height 130
  - horizontalAlignment 130
  - horizontalTextPosition 130
  - icon 130
  - iconTextGap 130
  - insets 130
  - labelFor 130
  - layout 130
  - managingFocus 130
  - maximumSize 130
  - minimumSize 130
  - name 130
  - nextFocusableComponent 130
  - opaque 130
  - optimizedDrawingEnabled 130
  - paintingTile 131
  - preferredSize 131
  - registeredKeyStrokes 131
  - requestFocusEnabled 131
  - rootPane 131
  - text 131
  - toolTipText 131
  - topLevelAncestor 131
  - UI 129
  - UIClassID 129
  - validateRoot 131
  - verticalAlignment 131
  - verticalTextPosition 131
- visible 131
- visibleRect 131
- width 131
- x 131
- y 131
- DSdbjList 123
  - accessibleContext 123
  - alignmentX 123
  - alignmentY 123
  - anchorSelectionIndex 123
  - autoscrolls 123
  - background 123
  - border 123
  - bounds 123
  - cellRenderer 123
  - component 123
  - componentCount 124
  - components 124
  - dataBinding 124
  - debugGraphicsOptions 124
  - doubleBuffered 124
  - enabled 124
  - firstVisibleIndex 124
  - fixedCellHeight 124
  - fixedCellWidth 124
  - focusCycleRoot 124
  - focusTraversable 124
  - font 124
  - foreground 124
  - graphics 124
  - height 124
  - insets 124
  - lastVisibleIndex 124
  - layout 124
  - leadSelectionIndex 124
  - listData 124
  - managingFocus 124
  - maximumSize 124
  - maxSelectionIndex 124
  - minimumSize 124
  - minSelectionIndex 124
  - model 124
  - name 124
  - nextFocusableComponent 125
  - opaque 125
  - optimizedDrawingEnabled 125
  - paintingTile 125
  - preferredScrollableViewportSize 125
  - preferredSize 125
  - prototypeCellValue 125
  - registeredKeyStrokes 125
  - requestFocusEnabled 125
  - rootPane 125
  - scrollableTracksViewportHeight 125
  - scrollableTracksViewportWidth 125
  - selectedIndex 125

- selectedIndices 125
- selectedValue 125
- selectedValues 125
- selectionBackground 125
- selectionEmpty 125
- selectionForeground 125
- selectionInterval 125
- selectionMode 125
- selectionModel 125
- toolTipText 125
- topLevelAncestor 125
- UI 123
- UIClassID 123
- validateRoot 126
- valuesAdjusting 126
- visible 126
- visibleRect 126
- visibleRowCount 126
- width 126
  - x 126
  - y 126
- DSdbjTextArea 126
  - accessibleContext 126
  - actions 126
  - alignmentX 126
  - alignmentY 126
  - autoscrolls 126
  - background 126
  - border 126
  - bounds 126
  - caret 126
  - caretColor 126
  - caretPosition 127
  - columns 127
  - component 127
  - componentCount 127
  - components 127
  - dataBinding 127
  - debugGraphicsOptions 127
  - disabledTextColor 127
  - document 127
  - doubleBuffered 127
  - editable 127
  - enabled 127
  - focusAccelerator 127
  - focusCycleRoot 127
  - focusTraversable 127
  - font 127
  - foreground 127
  - graphics 127
  - height 127
  - highlighter 127
  - insets 127
  - keymap 127
  - layout 127
  - lineCount 127
  - lineEndOffset 127
  - lineOfOffset 127
  - lineStartOffset 128
  - lineWrap 128
  - managingFocus 128
  - margin 128
  - maximumSize 128
  - minimumSize 128
  - name 128
  - nextFocusableComponent 128
  - opaque 128
  - optimizedDrawingEnabled 128
  - paintingTile 128
  - preferredScrollableViewportSize 128
  - preferredSize 128
  - registeredKeyStrokes 128
  - requestFocusEnabled 128
  - rootPane 128
  - rows 128
  - scrollableTracksViewportHeight 128
  - scrollableTracksViewportWidth 128
  - selectedText 128
  - selectedTextColor 128
  - selectionColor 128
  - selectionEnd 128
  - selectionStart 128
  - tabSize 128
  - text 129
  - toolTipText 129
  - topLevelAncestor 129
  - UI 126
  - UIClassID 126
  - validateRoot 129
  - visible 129
  - visibleRect 129
  - width 129
  - wrapStyleWord 129
  - x 129
  - y 129
- DSdbjTextField 115
  - accessibleContext 115
  - actionCommand 116
  - actions 116
  - alignmentX 116
  - alignmentY 116
  - autoscrolls 116
  - background 116
  - border 116
  - bounds 116
  - caret 116
  - caretColor 116
  - caretPosition 116
  - columns 116
  - component 116
  - componentCount 116
  - components 116

- dataBinding 116
- debugGraphicsOptions 116
- disabledTextColor 116
- document 116
- doubleBuffered 116
- editable 116
- enabled 116
- focusAccelerator 116
- focusCycleRoot 116
- focusTraversable 116
- font 116
- foreground 117
- graphics 117
- height 117
- highlighter 117
- horizontalAlignment 117
- horizontalVisibility 117
- insets 117
- keymap 117
- layout 117
- managingFocus 117
- margin 117
- maximumSize 117
- minimumSize 117
- name 117
- nextFocusableComponent 117
- opaque 117
- optimizedDrawingEnabled 117
- paintingTile 117
- preferredScrollableViewportSize 117
- preferredSize 117
- registeredKeyStrokes 117
- requestFocusEnabled 117
- rootPane 117
- scrollableTracksViewportHeight 118
- scrollableTracksViewportWidth 118
- scrollOffset 117
- selectedText 118
- selectedTextColor 118
- selectionColor 118
- selectionEnd 118
- selectionStart 118
- text 118
- toolTipText 118
- topLevelAncestor 118
- UI 115
- UIClassID 115
- validateRoot 118
- visible 118
- visibleRect 118
- width 118
- x 118
- y 118
- DSdbNavigator 113
  - accessibleContext 113
  - alignmentX 113
  - alignmentY 113
  - autoscrolls 113
  - background 113
  - border 113
  - bounds 113
  - buttonBackground 113
  - buttonForeground 113
  - commandVisible 114
  - component 114
  - componentCount 114
  - components 114
  - dataBinding 114
  - debugGraphicsOptions 114
  - deleteVisible 114
  - doubleBuffered 114
  - enabled 114
  - firstVisible 114
  - focusCycleRoot 114
  - focusTraversable 114
  - font 114
  - foreground 114
  - graphics 114
  - height 114
  - insertVisible 114
  - insets 114
  - lastVisible 114
  - layout 114
  - managingFocus 114
  - maximumSize 114
  - minimumSize 114
  - name 114
  - nextFocusableComponent 114
  - nextVisible 114
  - opaque 114
  - optimizedDrawingEnabled 115
  - paintingTile 115
  - preferredSize 115
  - previousVisible 115
  - registeredKeyStrokes 115
  - requestFocusEnabled 115
  - rootPane 115
  - statusBackground 115
  - statusForeground 115
  - statusVisible 115
  - toolTipText 115
  - topLevelAncestor 115
  - UIClassID 113
  - validateRoot 115
  - visible 115
  - visibleRect 115
  - width 115
  - x 115
  - y 115
- JCTreeData 111
  - about 111
  - class 111

- currentGlobalBookmark 111
- currentGlobalTable 111
- dataTableTree 111
- eventsEnabled 111
- JCTreeDataComponent 112
- listeners 111
- metaDataTree 111
- modelName 111
- modified 112
- version 112
- TreeDataBeanComponent 112
  - class 112
  - dataSourceNames 112
  - dataSources 112
  - resourceName 112
  - root 112
  - serializationFile 112
  - serializationRequired 112
  - structureOnly 112
- TreeDataBeanCustomizer 112
  - alignmentX 112
  - alignmentY 112
  - background 112
  - component 112
  - componentCount 112
  - components 112
  - enabled 113
  - font 113
  - foreground 113
  - insets 113
  - layout 113
  - maximumSize 113
  - minimumSize 113
  - name 113
  - object 113
  - preferredSize 113
  - visible 113

## Q

- query
  - basics 26
  - requering a database 56
  - setting a query in the Data Bean customizer 76
  - SQL, specifying 51
  - store results 27
- Quest Software technical support
  - contacting 5
- question mark parameter 26

## R

- ReadOnlyBindingListener
  - methods 63

- ReadOnlyBindingModel
  - in data binding 9
  - refreshing tables 39
  - related documents 3
  - relationship
    - abstract 14
  - remove 55
  - removeChildren 55
  - result set 56
    - adding rows programatically 57
    - displaying 82
    - performing updates 56
  - results
    - store query results 27
  - root constructor 101
  - root level
    - creating 25
  - root table 14, 19
  - row index 19
  - Row Nodes
    - naming 37
  - rows
    - accessing from a database 57
    - adding to a result set 57
    - keeping track of 19
    - set maximum number at design-time 74

## S

- sample database
  - E-R diagram 99
- sample programs 99
- serialization file 43, 71
  - saving 73
- Set
  - DataBean 32
- setColumnTableRelations 40
- setting permissions 39
- single level data binding 9
- SQL
  - data manipulation language 26
  - query 14, 100
    - prepared statement 26
    - specifying 51
  - Statement tab 30, 84
  - structure of the sample database 99
  - subprotocol 100
  - sub-tables 14
  - support 5
    - contacting 5
    - FAQs 5

## T

- table
  - access
    - Data Access Tab 79
  - driver tables 76
  - getting names 39
  - joining 76
    - code snippet 38
  - refreshing 39
  - specifying 51
- Table chooser dialog 32
- tables
  - specify for access 26
- technical support 5
  - contacting 5
  - FAQs 5
- text field
  - display 22
- traversing data
  - methods for 54
- TreeData Bean 82
- TreeDataBeanComponent 112
  - class 112
  - dataSourceNames 112
  - dataSources 112
  - resourceName 112
  - root 112
  - serializationFile 112
  - serializationRequired 112
  - structureOnly 112
- TreeDataBeanCustomizer 112
  - alignmentX 112
  - alignmentY 112
  - background 112
  - component 112
  - componentCount 112
  - components 112
  - enabled 113
  - font 113
  - foreground 113
  - insets 113
  - layout 113
  - maximumSize 113
  - minimumSize 113
  - name 113
  - object 113
  - preferredSize 113
  - visible 113
- TreeModel 34, 47
- Type 1 driver 49
- Type 4 driver 49
  - definition 49
- types
  - of data bound components 89
- typographical conventions 2

## U

- unbound data 41
  - definition 41
  - generating a table's data programmatically 24
- updates
  - batching 42
  - database, set commit policy 26
  - performing 56
- useful classes
  - code snippets 37
- using JarMaster to customize the deployment archive 133

## V

- VectorData 24
- view 14
  - multiple data views 13
- virtual column 58
  - computation order 59
  - computed columns 80
  - define 81
  - set via a customizer 80
- visual components
  - using 10

## W

- Windows
  - database driver installation 16

